Proceedings of the GCC Developers' Summit

June 28th–30th, 2006
Ottawa, Ontario
Canada

Contents

Conference Organizers

Andrew J. Hutton, *Steamballoon Incorporated*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
Ian Lance Taylor, *Google*
C. Craig Ross, *Linux Symposium*
Andrew J. Hutton, *Steamballoon Incorporated*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

An interblock VLIW-targeted instruction scheduler for GCC

Andrey Belevantsev
ISP RAS
abel@ispras.ru

Maxim Kuvyrkov
ISP RAS
mkuvyrkov@ispras.ru

Vladimir Makarov
Red Hat
vmakarov@redhat.com

Dmitry Melnik
ISP RAS
dm@ispras.ru

Dmitry Zhurikhin
ISP RAS
zhur@ispras.ru

Abstract

Modern VLIW architectures (e.g. IA-64) require instruction level parallelism (ILP) to be explicitly exposed by a compiler. An instruction scheduler is a key compiler component for utilizing ILP. The current GCC scheduler has a number of pitfalls in approaching this goal, including the oldest interblock scheduling algorithm (whose weakness is in prevention of instruction cloning), non-optimal region formation, a traditional two-pass execution scheme (before and after register allocation), and lack of transformations for eliminating false dependencies (e.g. register renaming and forward substitution).

In this paper, we present an approach for implementing new aggressive instruction scheduler for GCC. The scheduling algorithm is inspired with selective scheduling and resource-constrained software pipelining approaches. It is mainly targeted for VLIW-like platforms, but the framework being implemented is general enough and it can be used for other targets in the future. The key features of the approach are as follows: works with DAG regions, supports code motion with adding bookkeeping insns, supports register renaming and forward substitution, and integrates with software pipelining. We discuss the algorithm and its adaptation to GCC, implementation issues, and the current state of the project.

1 Introduction

Current GCC instruction scheduler has a long history. Integration of the Haifa scheduler has started in August 1997, when the first version of `haifa-sched.c` appeared in CVS repository. Since then, the scheduler's source code was significantly cleaned up[1], and new features (such as scheduling of extended basic blocks and DFA scheduling) were added. Nevertheless, the scheduler doesn't work well for the modern architectures, which require exposing instruction-level parallelism during compilation. These are the EPIC architecture [EPIC] and various VLIW embedded systems. The

[1] Still, some bits of the scheduler were integrated in GCC only recently. For example, using GCC edge probabilities instead of scheduler's own evaluation is supported only since February 2006.

major pitfalls of the current scheduler are as follows:

- The oldest interblock scheduling approach. The Haifa scheduler is a variant of dominator-path based scheduling, which appeared first in 1992. It tends to improve performance on the critical path sacrificing other paths in a region. Conceptually better approach should try to improve performance on all paths through the region. The second problem with the approach is that it doesn't support instruction cloning, i.e. code motion is allowed only when creating bookkeeping copies to preserve program correctness is not necessary. Moving up branches is also impossible without instruction cloning.

- Interblock scheduling is placed before register allocator and reload passes. This leads to the following disadvantages: firstly, scheduled insns could be later significantly changed (especially by reload); second, a single instruction before reload could be later splitted into several target insns, so before reload the scheduler works with inaccurate model of pipeline hazards.

- Lack of instruction transformations that eliminate nontrue dependencies. The most important such transformations are partial register renaming and forward substitution. Other examples are predication (which turns control dependencies into data dependencies) and instruction mutation (which turns e.g. shift insn into multiply insn when this allows to execute more insns on the current cycle). It should be noted that register renaming and forward substitution are essential to achieve good results when a scheduler is placed after register allocation.

Since the Haifa scheduler implements code motion without copies, it doesn't support creation of basic blocks or instructions during scheduling. On the contrary, adding support for instruction cloning and other transformations requires the infrastructure for manipulation of instructions and basic blocks. During our project on implementing speculation support in the scheduler we have found that creating such an infrastructure inside the Haifa scheduler is hard and requires a lot of hacking. Besides this, it is desirable to have a framework for evaluating instruction transformations, which should allow adding/removing and enabling/disabling transformations. This is also not easy to achieve with the current implementation.

Based on the above considerations, the project on implementing new aggressive interblock instruction scheduler for GCC was started. The ultimate goal of the project is to create an infrastructure capable of scheduling arbitrary DAGs and supporting various instruction transformations, among which are instruction cloning, partial register renaming, forward substitution, code motion of branches, and unification. The scheduler infrastructure should support fine-grained control over transformations to allow e.g. enabling speculation support on ia64 and disabling instruction cloning on embedded targets. The other goal is to create an infrastructure for manipulating instructions and control flow graph during scheduling. Some of required functionality could be borrowed from the previous project on speculation support.

We have chosen selective scheduling [Moon97] as a basis, also using thoughts from resource-constrained software pipelining (RCSP) [Aiken95] and percolation scheduling [Nicolau85]. Selective scheduling provides a thorough summary of scheduling ideas also mentioned in RCSP and percolation scheduling approaches, so it serves as a great starting point. Its basic algorithms are significantly

reworked by us for the GCC implementation.

We are trying to follow evolutional approach in the implementation process. Basic pieces of the infrastructure that provide the simplest code motion features are implemented first, and then more instruction transformations are added. Our goal is to get the working scheduler as early as possible, and then keep it working after addition of every single transformation.

The project is an ongoing work started in September 2005. At the moment of this writing we are in the middle of the way. This paper describes the current status of the project focusing on the improvements we made to the basic algorithms. The rest of the paper is organized as follows. Section 2 provides an overview of the basic scheduling algorithms of the selective scheduling approach focusing on the key ideas we use for the GCC scheduler. Section 3 covers the most important implementation challenges and solutions that we designed for them. Our current progress is sketched in Section 4, while Section 5 concludes.

2 The basic algorithm

Selective scheduling approach contains the following key ideas of the interblock scheduler design: separation of the available insn computation and the actual code motion stages, formulation of the computation stage through simple propagation routines, incremental recomputation of the available insns, *partial* register renaming through scheduling *right-hand sides* (RHSes) instead of whole insns, representation of a VLIW insn as a tree for moving up branches. We use all these concepts in our implementation and explain them in the below subsections.

2.1 The main scheduling routines

The scheduler takes as an input an arbitrary part of the control flow graph that forms a DAG. The driver routine breaks the CFG onto several DAGs and schedules each of them separately via `schedule_region` routine, which contains the main scheduling loop. Each iteration of this loop tries to gather a *parallel group* of instructions at the currently scheduling point, and then to advance the point. A parallel group corresponds to a single VLIW instruction when targeting to a VLIW architecture, or to a regular instruction in other cases. The loop terminates when no more instructions are left for scheduling.

Filling a parallel group is handled by the `fill_group` routine. Its driver loop adds new instructions to the current parallel group until target resources and data dependencies permit. First, the set of available operations, or *av set*, is computed for the current scheduling point. Then the best operation is chosen and scheduled. Finally, the best operation is moved up to the scheduling point from its original location, possibly creating bookkeeping copies and updating av sets along its moving path (see Fig. 1).

The basic routines do not change when more transformations are added to the scheduler. Instead, the new functionality is incorporated into the computation, choosing, and code motion stages. For example, partial register renaming is done through scheduling RHSes instead of whole instructions. An instruction is eligible for register renaming when it is a store to a register, i.e. of the form `(set (reg) (rhs))`. For such an instruction, only its RHS participates in the scheduling process. The RHS is added to the av set, the choosing routine besides the best operation also finds a target register for the RHS, and the operation is scheduled

```
fill_group(insn) {
  group = create_empty_group ();

  while (1)
  {
    av_set = compute_av_set (insn);

    best_op = choose_best_op (av_set);
    if (best_op == NULL)
      break;

    schedule_op (best_op);
    move_op (insn, best_op);
    advance_scheduling_point (&insn);
  }

  return group;
}
```

Figure 1: The `fill_group` routine

as `best_reg = best_rhs`. Corresponding bookkeeping copies for this instruction are created during code motion in `move_op`. We will consider that some insns are scheduled as RHSes in the below subsections and will use a term *operation* to denote either an insn or an RHS.

2.2 Computation stage

The task of the computation stage is to gather all instructions available for scheduling along all execution paths. The simple way to do this is to traverse the DAG starting from current scheduling point in reverse topological order. When visiting an instruction (a graph node n), first a set of the insns available immediately after n is computed as a union of n's successors' av sets. Then this set is propagated through n by filtering out its elements that could not be moved up past n. Finally, n's operation is collected and added to the set:

$$avset(n) = moveup\_set(\bigcup_{x \in Succ(n)} avset(x)) \bigcup av\_op(n)$$

As the code motion stage invalidates the av set found, it should be recomputed after scheduling each single insn. The recomputation should be done incrementally to avoid high overhead. It can be noticed that after code motion av sets become invalid only along the moving path and could be restored using the valid sets from other basic blocks. Hence the key idea of the computation stage is to save the intermediate av sets at the beginning of each basic block to avoid recomputating the sets from scratch.

```
moveup_op(insn, op) {

  /* Ok to move if no dependence.  */
  if (!data_dep_between (insn, op))
    return op;

  /* Try substitution.  */
  if (true_dep (insn, op) && rhs_p (op)
      && copy_insn_p (insn))
    {
      dst = SET_DEST (insn);
      src = SET_SRC (insn);

      if (dst_is_in (op, dst))
        return substitute (op,
                           dst, src);
    }

  /* Can't do anything.  */
  return NULL;
}
```

Figure 2: The `moveup_op` propagation helper

The `moveup_set` routine filters its input set with the `moveup_op` helper, which determines whether the given operation could be propagated through the current insn. The `moveup_op` logic depends on the type of the code motion we want to support. When scheduling whole insns, it is enough to check for the data dependence between the two insns. However, when forward substitution is supported through RHS scheduling, we can do better. For example, `x+y` RHS could be moved before the `y=z` copy as `x+z`, i.e. true dependence

```
//current scheduling point
//best_op: z = b + c

if (...) {
  a = b;
} else {
  a = c;



}

z = b + c;
```

(a) Before the traversal

```
//current scheduling point
z = b + c;

if (...) {
  a = b;
} else {
  a = c;
  //bookkeeping copy
  z = b + c;
}
//found and deleted:
//z = b + c;
```

(b) After traversing 'then' path

```
//current scheduling point
z = b + c;

if (...) {
  a = b;
} else {
  a = c;
  //found and deleted:
  //z = b + c;
}
```

(c) After traversing 'then' and 'else' paths

Figure 3: Creating bookkeeping code

between these two operations can be eliminated with substitution. In this case the propagation helper will return the modified operation (see Fig. 2).

2.3 Code motion stage

When the av set is calculated, the scheduler chooses the best element of the set (either an instruction or an RHS) for moving into the current group. The task of choosing the best operation from the av set is orthogonal to the rest of the scheduler and usually is driven by implementation-dependent heuristics, so it is covered in the next section. Here we assume that the best operation `best_op` is chosen and now the task of the code motion stage is to actually move it up to the current scheduling point.

The code motion process is driven by the `move_op` routine. It traverses the DAG starting at the current scheduling point in search of the original operations (from which `best_op` could be derived). Let's assume first that we're scheduling only instructions, then it's enough to search just for `best_op`. When the operation is found, it is deleted from its original place and moved to the parallel group. Then the routine backtracks and continues the traversal. When backtracking along the already traversed code motion path, bookkeeping copies of `best_op` are inserted on edges that join the current moving path from outside. When the traversal explores other code motion path and sees already created bookkeeping copy, it is recognized as original operation and deleted in the same way (see Fig. 3). This allows to create only necessary bookkeeping code.

The process is more complicated when RHSes are also scheduled. It is not enough to search for the `best_op` because it could change through substitution. Hence when traversing through a copy instruction, we should "unsubstitute" `best_op` to reproduce its original form and add the resulting operation to the set of operations we're searching for. Back to our previous example, when `x+z` is the best operation and we're traversing through `y=z`, we don't know whether this operation was moved up earlier as `x+y` (through substitution) or as `x+z` (unchanged), thus we should search for both forms below the copy. To reduce the number of operations we should search for, the set of these operations is intersected with the av sets saved in basic blocks. This is possible because the available operations which can be found below a DDG node should be in its av set.

When the original operation is found, it is either removed or changed to a copy `old_dest=new_dest`, when register renaming is used. Then during backtracking the current form of the `best_op` at the node being tra-

versed should always be retained to allow correct creation of bookkeeping code.

2.4 Tree instruction

The convenient form for representing a parallel group (i.e. instructions that could be emitted at the same cycle) is a *tree* instruction (introduced in [Ebcioğlu88]). Each node of the tree instruction corresponds to a branch test, and leaf nodes correspond to instruction labels where their parent can branch to. Each edge of the tree instruction corresponds to sequential operations. Tree correctness and its execution semantics is determined by the target. For example, when the target is not capable of multiway branching, the tree would not contain any branch test nodes.

The concept of tree instruction is necessary for doing code motion of branches. In this case the computation stage takes branches into account allowing them to be propagated up to the next branch. The code motion stage analogously searches also for branches and creates bookkeeping copies of them. The `fill_group` routine is changed to emit the insns into the tree instruction. When a branch is scheduled and there's still place in the current parallel group, an extra scheduling point (or *boundary*) is created. The `compute_av` and `move_op` routines are called for each group boundary, thus allowing to schedule instructions at both targets of the branch.

3 The GCC implementation

The basic scheduling algorithms sketched in the previous section are reworked and somewhat improved for the GCC implementation. The key improvement that we make in our implementation is the use of data dependencies to avoid unnecessary computations during the computation stage. The idea here is that we don't want to collect operations that would anyway be filtered out during later traversal. Removing such operations as early as possible reduces the time needed for each update of the av sets, which happens on each scheduling iteration.

The second major implementation feature is handling a number of instruction transformations through annotating dependencies with an extra data. The data is used to decide whether the scheduler is able to break the particular dependence using the given transformation. For example, data speculation is handled by annotating dependencies with a *spec* flag (what kind of speculation should be used) and a *weakness* (how "probable" is this dependence); forward substitution is handled by placing substitutable and non-substitutable dependencies to different lists. The advantage of this approach is that when adding new transformation, it allows natural extension of the existing framework. Namely, the basic infrastructure remains untouched, and the only changes go to the dependence data structures and the propagation routines of computation and code motion stages.

Other implementation changes include computation of register liveness sets needed for register renaming and CFG/insn handling infrastructure for code motion. These changes are necessary to implement the concepts of the previous chapter in any real compiler. Besides those, we discuss implementation and target specific portions of the scheduler that are not covered in the basic approach, namely choosing the best instruction for scheduling and the best register for renaming, and region formation.

3.1 Input regions

The scheduling approach we use permits any kind of DAG regions as an input. The needed

functionality is to find whether a basic block or an edge belongs to the current region and the possibility to extend a region when new basic blocks are created. The former is used during region traversal, whereas the latter should be done during the code motion stage. At the moment of writing we use the region infrastructure from the Haifa scheduler extended with the patches for the ia64 speculation support [IA64Speculation]. In the future we consider to use the region finder implemented for GCC by Kenneth Zadeck [ZadeckRegions].

3.2 Data structures and their computation

There is a number of data structures needed by the scheduler that can be represented as linked lists: a path in a DAG, parallel group boundaries, fences[2], and av sets. These structures are backed up by a single linked list interface. Custom list types are implemented through this interface, and their data is accessed through a union contained in a list node analogously to `struct rtx_def`. This provides a uniform interface for accessing, manipulating, and iterating over the list-like data structures. All per-instruction data is stored in the `s_i_d` array indexed by `INSN_UID` analogously to the Haifa scheduler.

The most important role is played by the availability and liveness sets (av and lv sets). This is because the contents of av and lv sets should be valid on each scheduling iteration, and their initial computation and update is not trivial. The basic approach for computing av sets suggests to traverse a DAG collecting all operations on the way and filtering out those that can't be propagated through currently visiting node. This solution is not optimal: an operation collected from the bottom of the DAG can be moved up somewhere to the middle before it would be filtered out, wasting time for unnecessary propagation between the two nodes. An ideal solution here would be to traverse a dependence graph[3] instead of a DAG. However, this is not possible due to: a) control dependencies should be represented explicitly, which leads to the quadratic complexity of the algorithm, and b) a copy of av set should be left at the beginning of each basic block to allow quicker code motion. We are implementing the combined approach, i.e. the algorithm still traverses a DAG, but uses dependencies to decrease the number of computations needed when visiting a node.

3.2.1 Instruction dependencies and av sets

Instruction dependencies are split into two types—*hard* and *weak* ones—and placed to different lists accordingly.[4] Weak dependencies are those that can be broken by the scheduler, while hard ones are those that can't. Only true dependencies can be weak, while hard ones are {anti,output}-dependencies and all others (for example, created from `ASM`s or `SCHED_GROUP_P` insns). Weak dependencies that require different scheduling transformations to be broken (for example, either with substitution, if the insn is scheduled as a RHS, or with data speculation) can further be splitted onto several lists. The Haifa scheduler will see all dependencies as hard ones.

Maintaining several dependence lists helps to distinguish weak dependencies (which are of the most interest to the scheduler) among others. We can easily check whether an instruction has hard dependencies (not traversing the

[2] A *fence* is a point at which the next parallel group will be created and filled.

[3] Given that it contains both data and control dependencies.

[4] On the current GCC mainline this separation is implicit: the whole dependence list is maintained sorted, and weak dependencies are located after hard ones.

whole dependence list), or traverse and sort only weak ones. However, this comes at the expense of writing additional code to handle all dependencies at once.

Creation of dependencies and their separation by types happens in `sched-deps.c`. During the computation stage the node visiting algorithm looks as follows:

- When entering a node, first a union of node's successors' av sets is computed the same way as in the original approach, i.e. the visiting routine is called recursively.
- Let's denote node's operation as *op*. If *op* has any hard dependencies, it is not added to the av set. If *op* has only substitutable weak dependencies, then it's added to the set, and all *op*'s producers are notified that when they are visited, *op* should be substituted. This is achieved by maintaining additional list of operations that need substitution for each copy instruction. If *op* has no dependencies (or has weak speculative dependencies), it's also added to the list.
- When a copy node is being visited, we should check whether any pending substitutions should be performed on operations from the av set. As we maintain a separate list of such operations, we don't need to traverse the whole set searching for them. Instead, a substitution is performed for each *pend_op* from node's pending list, and all weak dependencies of node's *op* are added to *pend_op*'s dependencies. These dependencies also need to be checked the same way as in the previous step: if *op* has any hard dependencies, then all *pend_ops* should be removed from the av set.
- When the node visited is a basic block head, a copy of the intermediate av set is left at the node analogously to the original approach.

Consider Fig. 4 as an example. Node 4 doesn't have any hard dependencies, but has one weak substitutable dependence, so its operation can be scheduled as an RHS: `y+1` is added to the av set and to the pending ops of Node 2. Node 2 has no dependencies so it's safe to add it and all its pending ops to the av set. Before adding pending operations they should be substituted, so `y+1` becomes `z+1`.

```
(1) if cc0          av(1) = {z, z+1}
     /   \
      (2) y = z     pend_ops(2) = {y+1}
           \
      (3) if cc1    av(3) = {y+1}
           /
    (4) u = y+1     weak_dep(4) = {2},
                    is substitutable
```

Figure 4: Using dependencies during `compute_av`

3.2.2 Liveness sets

Liveness information is stored as `regsets` and computed using the scheme similar to the computation of av sets. The `compute_lv` driver also traverses a DAG and updates the intermediate lv sets. First, the intermediate lv set is calculated as a union of node's successors' lv sets. Then all registers that are set or clobbered by the visiting insn are subtracted from the lv set, and all registers that are used by the insn are added to the set. A copy of the intermediate lv set is saved at the beginning of each basic block to allow faster updating.

The sets of registers that are used, set, or clobbered by an instruction are calculated as a side effect of dependence analysis in `sched_deps.c`. The sets are the part of per-instruction dependence data structure `deps_`

`insn_data` and are stored in `d_i_d` array indexed by `INSN_LUID`. Initial lv sets are not computed from scratch but rather taken from `global_live_at_start` sets.

3.3 Choosing the best instruction

When the av set is computed, it's time to choose the best available operation from the set. This task is usually based completely on heuristics. The basic set of heuristics to use is taken from the Haifa scheduler, i.e. critical path, register pressure, control flow probability, etc. Data speculation adds to those the weakness degree of speculative dependence. The point that should be made here is that the role and weight of each heuristic is to be determined during evaluation and tuning of our implementation, which are yet in the future. As of now, we'd like to make several points regarding the difficulties in choosing the best operation that come from forward substitution and register renaming transformations.

When scheduling an RHS, there's a number of problems to be solved. First, a target register for storing the result of RHS should be chosen. This register should not be live along the moving path of the RHS from its original location(s) to the current scheduling point. This condition is not easy to check because during the choosing stage we don't know this moving path. The path is implicitly constructed during the code motion stage for the chosen operation, but before this we need to know this path for *all* available RHSes. Current implementation of this step is slow and actually performs the traversal similar to that of `move_op` for all members of the av set. The better way would be to calculate the sets of available registers for each RHS during the computation stage.

If original register of an RHS is available, then it's always the best choice. If not, then the best register is chosen in the way analogous to that of `regrename.c`. When no registers are available, the RHS cannot be scheduled. In the future, the logic for spilling a register in favor of using it for renaming should be implemented, namely using function saved registers for renaming.

The second problem is using the DFA pipeline descriptions for modeling the processor state. When scheduling an insn, DFA is used analogously to the Haifa scheduler, i.e. to check whether target resources permit scheduling of the insn on the current cycle. The first cycle multipass scheduling [Makarov03] will also be adapted from the Haifa scheduler to work directly on av sets. When scheduling an RHS, a valid insn should be formed first and then passed to the DFA interface. The possible optimization would be to extend the DFA interface to be able to handle RHSes at least for some cases.

The last problem is scheduling of multilatency operations. Fortunately, it could be solved with adding an extra dependence attribute called *tick*. A tick holds the simulated cycle number on which producer's result will be ready[5]. An instruction cannot be considered for scheduling until the current simulated cycle reaches a maximum from ticks of instruction's dependencies.

3.4 Code motion

The basic thing that is needed for the implementation of the code motion with support for instruction cloning is an interface for manipulating instructions and basic blocks. Two typical problems to be solved are creating bookkeeping insns and creating basic blocks for bookkeeping code. The code which solves those tasks and extends global data structures

[5] In the Haifa scheduler this information is represented as a per-instruction attribute called `INSN_TICK`.

of GCC is already in place. The extra code to be written is the proper extension of the data structures specific to the scheduler. This task for creating basic blocks for recovery code was solved in the ia64 speculation patch and would be taken from there.

The other point to make here is that one of the expensive parts of `move_op` is substitution/unsubstitution that should be done on the operations we are searching for. Fortunately, it is possible to make use of dependencies analogously to the computation stage. Substitution should be tried only for those copy insns that are found in the weak dependencies of the searched operations. It is also possible to save the original form of the operation during the computation stage and use it without performing actual substitution.

3.5 Target-specific details

A target affects the scheduling process through a number of hooks. Analogously to the Haifa scheduler the scheduler hooks can be used to override the choosing decisions and instruction costs. The new hooks we add are the hook to determine dependence type (for example, forbid creating of weak dependencies), the hook to disable scheduling an insn as an RHS, the hook to affect register choosing decisions, and the hooks for controlling speculation support similar to those of the ia64 speculation patch. The features of `cc0` targets (for example, scheduling the `cc0` user right after the setter) would be handled via dependence attributes. Target-specific reorganization passes such as bundling on ia64 would remain untouched. For this purpose, we will preserve the feature of marking the insns that start a new cycle with TImode. In the future, it is possible to consider integrating the bundling with the scheduler, but this doesn't seem to worth the trouble.

4 Current progress

We planned the implementation process in a number of stages. As noted above, we've tried to follow the evolutional approach and keep the working scheduler after each stage. The stages are as follows:

- Implement the basic infrastructure, i.e. the driver, computation and code motion routines. Do not allow any interblock code motions or any transformations.
- Add interblock motions without copies and adopt the DFA interface to the new infrastructure. The resulting scheduler should be similar to the Haifa's.
- Allow instruction cloning during scheduling. This step requires support for creating bookkeeping code and liveness checking analogous to the Haifa one.
- Allow forward substitution and register renaming. After completion of this step and some cleanup the code can be placed in the FSF branch.
- Support creation of hard and weak dependencies in `sched-deps.c`. Use created dependencies in computation and code motion stages.
- Implement a tree instruction support, which would allow implementing code motion of branches.

The current status of the project is exactly in the middle of this way. At the moment of writing (April 2006) support for instruction cloning is implemented (i.e., checking of liveness information and bookkeeping code creation), and all the pieces are being tested together. The scheduler works on x86 in place of the old sched2 pass. However, benchmarks were not yet run on any platform.

5 Conclusions

In this paper, we present an effort of implementing new aggressive interblock instruction scheduler for GCC. The project goal is to design and implement a scheduling framework that can be easily extended to support a number of instruction transformations, such as register renaming, forward substitution, instruction mutation. Implementing those transformations allows to place the scheduler after register allocation passes. Following newer approach than the Haifa scheduler would improve scheduling for modern architectures such as EPIC.

Implementing a new scheduler for GCC has to be a long project. Taking ideas of selective scheduling and RCSP as a start, we have added better use of data dependencies, additional transformations such as speculation, and more complicated register renaming in our implementation. Now we are in the middle of the way, and there's still a lot of things to be done. The major features that are yet to be implemented are tree instruction support, code motion of branches, better use of dependencies, and using function saved registers for renaming. The future work would be to implement software pipelining on top of the scheduler, which is quite natural with our approach.

6 Acknowledgments

We'd like to thank Vladimir Makarov from Red Hat for extensive consulting on this project. The insights to instruction scheduling and GCC world provided by Vlad are invaluable. We thank Diego Novillo, Daniel Berlin, and James Wilson for giving helpful comments to this and other GCC work that we're doing. And we'd like to thank HP Company for sponsoring this project and the Gelato Federation for their attention to Itanium and GCC.

References

[Aiken95] Alexander Aiken, Alexandru Nicolau, and Steven Novack. Resource-Contrained Software Pipelining. IEEE Transactions on Parallel and Distributed Systems, 6(12), pp. 1248–1270, December 1995.

[GCCInternals] `http://gcc.gnu.org/onlinedocs/gccint`

[Ebcioğlu88] Kemal Ebcioğlu. Some design ideas for a VLIW architecture for sequential natured software. In *Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing)*, North Holland, Amsterdam, 3–21, 1988.

[EPIC] Michael S. Schlansker, B. Ramakrishna Rau. EPIC: An Architecture for Instruction-Level Parallel Processors. HP Laboratories Palo Alto Technical Report HPL-1999-111, February 2000. `http://www.hpl.hp.com/techreports/1999/HPL-1999-111.pdf`

[IA64Speculation] `http://gcc.gnu.org/ml/gcc-patches/2005-12/msg01924.html`

[Makarov03] Vladimir Makarov. The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC. In *Proceedings of GCC Developers' Summit*, Ottawa, Canada, June 2003.

[Moon97] Soo-Mook Moon and Kemal Ebcioğlu. Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining. ACM TOPLAS, Vol 19, No. 6, pages 853–898, November 1997.

[Nicolau85] Alexandre Nicolau. Percolation Scheduling: a Parallel Compilation Technique. Technical Report. UMI Order

Number: TR85-678., Cornell University, 1985.

[ZadeckRegions] `http://gcc.gnu.org/ml/gcc-patches/2005-09/msg01888.html`

Replacement special loop form by a call of built-in function

Tomáš Bílý
SUSE CR
tbily@suse.cz

Abstract

There are some patterns in computer programs (especially loops) which could be successfully replaced by a call of some built-in function and letting the GCC expanders decide which implementation is the best. As the built-in functions are coded specifically for a target platform such replacement can improve the runtime performance.

In this report, we present an implementation of this problem for some special cases of loops which can be transformed to `memset` or `memcpy` call. We also present performance results and a discussion limits of a current implementation.

1 Introduction

One of frequently used optimization methods is a loop transformation. There are some loop constructions with special statement patterns that could be transformed into a built-in function call and an expander infrastructure in GCC chooses the best possible implementation of this function. This is a way how a compiler could possibly take the best implementation of the program constructions for a specific platform and thus a runtime performance can be improved.

Many of program constructions initialize some arrays or copy the content of one array into another array. We were inspired by this phenomenon and we tried to search loops constructions and patterns for matching that could be used for transformation into `memset` or `memcpy` built-in call.

We can see that it is not necessary to transform the whole loop. This case is a particular instance of loop distribution (see [4]) where one of the loops is transformed to built-in call.

Finding these patterns is very similar to a vectorization approach. Some of the constructions could be optimized by use of current auto-vectorization infrastructure. But this could be done only on architectures that can support vector instructions. It is possible that the current auto-vectorization optimization could not be the best way to use.

Basic approach to handle the searching of these patterns is based on the theory of data-dependence analysis (see [1]). It must be examined that reordering the statements could not produce incorrect results.

Current GCC contains implementation of data dependence analyzer, scalar evolution infrastructure (see [2]) and auto-vectorization infrastructure (see [5]). We have heavily used all those infrastructures in our implementation of

outline problem.

Rest of paper is organized as follows. In Section 2 we describe our implementation. In Section 3 we present current measurements of runtime performance. In Section 4 we present summary and we describe some future plans for enhancement.

2 Implementation

We will call *builtinizer* an algorithm implementation that solves pattern matching and loop transformation into built-in call.

Our implementation of builtinizer is developed at the IR level of GIMPLE trees in SSA form (see [3]). Builtinizer are trying to handle

- array references (see Figure 1a or Figure 3a)
- indirect (pointer) references (see Figure 1b or Figure 3b)
- multidimensional arrays
- patterns on every level of nested loops

Patterns that builtinizer currently recognizes are

- `x[]..[] = 0`
- `*x = 0`
- `x[]..[i] = y[]..[i]` or `r = y[]..[i]; x[]..[i] = r;` (in GIMPLE)
- `*p = *q` or `r = *q; *p = r;` (in GIMPLE)

(a) array ref

```
for (i = 0; i < N; i++)
 {
  ...
  x[i] = 0;
  ...
 }
```

(b) pointer ref

```
for (i = 0; i < N; x++, i++)
 {
  ...
  *x = 0;
  ...
 }
```

Figure 1: `memset` built-in call pattern before transformation

```
memset (x, 0, N * sizeof (*x));
for (i = 0; i < N; i++)
 {
  ...
 }
```

Figure 2: `memset` built-in call pattern after transformation

Type of items must be one of the types `char`, `short`, `int`, `float`, `double`.

Current implementation handles an multidimensional arrays in inner most component only.

2.1 builtinizer structure

Builtinizer applies a set of analysis on each loop, followed by the built-in call transformation for the loops that had successfully passed the analysis phase. Examples such transformations you may see in Figure 1 and Figure 2 or Figure 3 and Figure 4.

(a) array refs

```
for (i = 0; i < N; i++)
 {
  ...
  x[i] = y [i];
  ...
 }
```

(b) pointer refs

```
for (i = 0; i < N; x++, y++, i++)
 {
  ...
  *x = *y;
  ...
 }
```

Figure 3: `memcpy` built-in call pattern before transformation

```
memcpy (x, y, N * sizeof (*x));
for (i = 0; i < N; i++)
 {
  ...
 }
```

Figure 4: `memcpy` built-in call pattern after transformation

```
for (i = 0; x [i] == -1; i++)
 {
  x [i] = 0;
 }
```

Figure 5: uncountable loop

2.2 builtinizer analysis

The first analysis phase probes the loop exit condition and number of iterations. Than examine some control-flow attributes (for example nesting level). One major restriction is required for a loop that can be builtinized. This is that loop is countable (an expression that calculates the loop bound could be constructed and evaluated at compile time or at runtime). For example the loop in Figure 5 is not countable loop. The loop bound analysis is done by scalar evolution analyzer.

Next step finds all memory references in the loop and checks if an access function that describes their modification in the loop can be constructed. This informations are required for the memory dependence tester and the access pattern analysis. Dependences that do not cover up memory operations are analyzed directly from SSA representation.

The final analysis phase scans all the statements in the loop and determines if they match pattern rules for transformation to built-in call.

2.3 builtinizer transformation

The analysis phases gather useful information about the loop, the statements and the data references. These informations are used during transformation phase. Data structures that store this informations are used from the vectorizer and the data dependence analyzer. They are

- `loop_vect_info` – holds information at the loop level
- `stmt_vect_info` – holds information at the statement level
- `data_reference` – holds information at the memory reference level

The loop transformation phase scans all accepted statements from the analysis phase and these statements grouped to patterns. The statement groups remove from the loop and insert relevant built-in call statement before the loop. The memory reference statements are implicitly removed by builtinizer but remaining scalar statements (that have some relevance to the removed statements) are expected to be removed by dead code elimination.

Figure 6 illustrates the transformation process. First, statements are grouped (by a pattern recognition) to $G_1 = \{S1\}$ and $G_2 = \{S3, S4\}$. Then group G_1 is transformed to `memset` built-in call (see Figure 6b) and group G_2 is transformed to `memcpy` built-in call (see Figure 6c).

(a) before builtinization

```
for (i = 0; i < N; i++)
 {
S1:  x[i] = 0;
S2:  y[2*i] = 0;
S3:  a = z[i];
S4:  w[i] = a;
     ...
 }
```

(b) after builtinization of S1

```
memset (x, 0, N * sizeof (*x));
for (i = 0; i < N; i++)
 {
S2:  y[2*i] = 0;
S3:  x = z[i];
S4:  w [i] = x;
     ...
 }
```

(c) after builtinization of S3 and S4

```
memset (x, 0, N * sizeof (*x))
memcpy (w, z, N * sizeof (*z))
for (i = 0; i < N; i++)
 {
S2:  y[2*i] = 0;
     ....
 }
```

Figure 6: The transformation process

3 Experimental results

We have gathered current experimental results. Testing systems were Pentium-M 1.6 GHz and AMD Athlon 64 X2 Dual Core Processor 2200. List of Pentium runtime performance results is in Table 1. List of Athlon runtime performance results is in Table 2. Numbers of `memset` and `memcpy` patterns recognized are in Table 3.

4 Conclusion and future plans

As could be seen in experimental results the current implementation improve runtime performance slightly in some cases. Almost all

loops in test-cases have small number of iterations then they are handled well by current loop optimizations.

In the future we will improve handling of multidimensional arrays, we will expand variety of `memset` filling constant and we will try to extend number of patterns for matching.

5 Acknowledgments

We would like to thank Andrew Pinski because this work is based on his patch (see [6]) and to Honza Hubička for his helpful advice.

References

[1] Randy Allen and Ken Kennedy. *Optimizing Compiler for Modern Architectures: A dependence based approach*. Morgan Kaufmann, 2001.

[2] Sebastian Pop Daniel Berlin, David Edelsohn. High-level loop optimization for gcc. In *Proceedings of the 2004 GCC Developer's Summit*, 2004.

[3] Free Software Foundation. Gcc internals manual. `http://gcc.gnu.org/onlinedocs/gccint/`.

[4] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[5] Dorit Naishlos. Autovectorization in gcc. In *Proceedings of the 2004 GCC Developer's Summit*, 2004.

[6] Andrew Pinski. [patch] [improvements branch?] loops to memset. `http://gcc.gnu.org/ml/gcc-patches/2004-09/msg00873.html`.

GCC options	without builtinizer (average result)	with builtinizer (average result)
tramp3d -n 50		
`-O2 -ffast-math`	3m03.58s	3m03.09s
`-O3 -ffast-math`	2m44.90s	2m44.88s
SPEC2000 164.gzip		
`-O2 -ffast-math`	188 (745)	186 (751)
`-O3 -ffast-math`	186 (753)	184 (762)
SPEC2000 175.vpr		
`-O2 -ffast-math`	146 (958)	146 (960)
`-O3 -ffast-math`	142 (986)	141 (993)
SPEC2000 181.mcf		
`-O2 -ffast-math`	209 (862)	209 (862)
`-O3 -ffast-math`	198 (910)	198 (910)
SPEC2000 253.perlbmk		
`-O2 -ffast-math`	156 (1151)	163 (1104)
`-O3 -ffast-math`	157 (1148)	163 (1105)
SPEC2000 300.twolf		
`-O2 -ffast-math`	226 (1325)	220 (1358)
`-O3 -ffast-math`	213 (1406)	218 (1373)
SPEC2000 177.mesa		
`-O2 -ffast-math`	186 (751)	184 (762)
`-O3 -ffast-math`	175 (802)	173 (814)
SPEC2000 179.art		
`-O2 -ffast-math`	159 (1639)	158 (1645)
`-O3 -ffast-math`	150 (1737)	149 (1740)
SPEC2000 188.ammp		
`-O2 -ffast-math`	323 (681)	322 (684)
`-O3 -ffast-math`	323 (681)	321 (686)

Table 1: Pentium-M experimental results

GCC options	without builtinizer (average result)	with builtinizer (average result)
tramp3d -n 100		
`-O2 -ffast-math`	1m51.04s	1m50.80s
SPEC2000 164.gzip		
`-O2 -ffast-math`	122 (1146)	121 (1150)
`-O3 -ffast-math`	123 (1141)	120 (1166)
SPEC2000 175.vpr		
`-O2 -ffast-math`	137 (1020)	135 (1034)
`-O3 -ffast-math`	134 (1044)	133 (1054)
SPEC2000 181.mcf		
`-O2 -ffast-math`	283 (636)	283 (636)
`-O3 -ffast-math`	281 (640)	281 (640)
SPEC2000 252.eon		
`-O2 -ffast-math`	74.2 (1752)	70.6 (1842)
`-O3 -ffast-math`	58.0 (2240)	58.0 (2241)
SPEC2000 253.perlbmk		
`-O2 -ffast-math`	129 (1390)	133 (1358)
`-O3 -ffast-math`	135 (1114)	136 (1103)
SPEC2000 300.twolf		
`-O2 -ffast-math`	259 (1159)	255 (1175)
`-O3 -ffast-math`	256 (1172)	255 (1177)
SPEC2000 177.mesa		
`-O2 -ffast-math`	106 (1325)	99.9 (1402)
`-O3 -ffast-math`	96.7 (1448)	96.6 (1450)
SPEC2000 179.art		
`-O2 -ffast-math`	191 (1360)	188 (1379)
`-O3 -ffast-math`	191 (1358)	192 (1357)
SPEC2000 188.ammp		
`-O2 -ffast-math`	165 (1333)	164 (1338)
`-O3 -ffast-math`	166 (1329)	164 (1339)

Table 2: Athlon experimental results

program name	# `memset` occurrences	# `memcpy` occurrences
tramp3d	102	125
SPEC2000 164.gzip	10	3
SPEC2000 175.vrp	4	0
SPEC2000 181.mcf	0	0
SPEC2000 186.crafty	47	3
SPEC2000 252.eon	6	23
SPEC2000 256.bzip	8	2
SPEC2000 300.twolf	5	3
SPEC2000 177.mesa	21	84
SPEC2000 179.art	1	3
SPEC2000 183.equake	15	9
SPEC2000 188.ammp	13	5

Table 3: number of `memset` and `memcpy` pattern occurrences

Call path profiling for unmodified, optimized binaries

Nathan Froyd, Nathan Tallent, John Mellor-Crummey, and Rob Fowler
Department of Computer Science
Rice University
{froydnj,tallent,johnmc,rjf}@cs.rice.edu

Abstract

Understanding the performance of today's large and complex programs requires a new generation of profiling tools that attribute costs to the full calling contexts in which they are incurred. We describe a new call path profiler for optimized code and the changes needed in GCC and other parts of the GNU tool chain to support such tools.

Although `gprof` has long been the standard for call graph profiling in the GNU toolchain, it suffers from several shortcomings. First, in modern object-oriented programs, costs must be attributed to full calling contexts because the cost of each call can be context dependent; `gprof` ignores this issue. Second, `gprof` relies upon instrumentation in procedure prologues to collect performance data. This imposes four costs: (1) recompilation is required; (2) nested instrumentation distorts the measurements; (3) instrumentation overhead can be significant; and (4) the instrumentation interferes with aggressive compiler optimization. We have developed a call-path profiler that avoids all of these shortcomings. To measure the performance of unmodified, fully-optimized binaries, we use stack unwinding and sampling to attribute costs to calling contexts and to collect frequency counts for call graph edges. Unlike `gprof`, `csprof` accurately attributes context-dependent costs. Furthermore, experiments with the SPEC CPU2000 benchmarks show that our new profiling strategy is significantly more efficient than `gprof` as well.

Profilers and other tools that rely on stack unwinding must correctly handle events that occur at any point in the execution. This requires more comprehensive unwinding information than what compilers already record to support C++ exceptions. In this paper, we describe changes made to GCC to record the additional information necessary. We present experimental results on the x86-64 platform that show our profiler has low overhead and that the additional unwinding information it requires is of modest size. To cope with compilers that don't record comprehensive unwind information, it is possible to recover this information using binary analysis of executables. We discuss modifications to `binutils` needed to support this and related analyses.

1 Introduction

The gap between peak and typical performance on modern microprocessor architectures has been growing with each new generation of processors. Today, only carefully tuned applications achieve a substantial fraction of peak

performance. While optimizing compilers improve application performance, compiler optimization often fails to deliver much of the improvement possible. As a result, much of the burden of performance tuning falls to application developers. Good performance tools are essential to help developers determine where they should invest their tuning effort.

Understanding where an application spends its time is only the first step toward tuning. The next steps are determining whether this is a symptom of inefficiency and understanding how inefficiency arises. Sometimes the answers are local, e.g., a loop iterating over a large amount of data doesn't utilize the memory hierarchy effectively. Often, answers are elusive. Is the time spent in a procedure the result of inefficiency in the procedure, or the result of the procedure being invoked frequently? An understanding of the contexts in which costs are incurred is vital for analyzing object-oriented abstractions, component-based software, and instantiations of templates for data and computation. Can the costs associated with using a particular abstraction, e.g., a set, be reduced by picking a different implementation? Questions such as this may have multiple answers: different choices may be appropriate for different instances of an abstraction within a program. For instance, a bit vector can be a good implementation where a dense set is needed, but other representations are preferable for sparse sets. To tune a program effectively, developers must know the contexts in which costs are incurred to choose among the myriad possibilities for tuning.

For over two decades, the approach pioneered in `gprof` [7] has been the standard for acquiring contextual information (in the form of call graphs) to interpret costs incurred. `gprof` inserts instrumentation in procedure prologues to log procedure entry and increment a count associated with (callsite, callee) pairs. A critical shortcoming of `gprof` is that it assumes that the cost of a function call is independent of its calling context. The cost of a function or method call can vary widely between object, component, or template instances and `gprof` lacks the ability to help pinpoint such differences to guide application tuning.

Using `gprof` has four additional shortcomings. *First,* `gprof` relies upon information collected by instrumentation in procedure prologues. With the GNU toolchain, adding this instrumentation requires recompilation of the program and all the non-standard libraries that it uses.

Second, executing instrumentation code in each procedure call dilates execution time. This can preclude the use of `gprof` on large production runs. We have observed a 3-14x slowdown on different systems of an execution of a synthetic call-intensive "torture test" written to showcase this problem [6]. Execution time dilation is not just a theoretical concern. Section 4 describes experiments with the SPEC CPU2000 integer benchmarks [12], which show that `gprof` instrumentation increases execution time by 82% on average. Object-oriented code with small methods is especially sensitive to dilation.

Third, the instrumentation in nested procedure calls dilates the measured cost of each procedure, thus introducing a systematic measurement error. This dilation disproportionately inflates costs attributed to small procedures. An analysis of how the fraction of time attributed to each function by `gprof` differs from that attributed by DCPI [2]—a well known, flat profiler with very low overhead—showed that for the SPEC CPU2000 integer benchmarks `gprof`'s measurements were distorted by 23% on average [6].

Fourth, `gprof`'s instrumentation-based approach precludes some compiler optimization.

For instance, in the GNU toolchain, `gprof` instrumentation requires frame pointers and thus, code measured with `gprof` cannot be fully optimized; this contributes to measurement distortion. Instrumentation-based approaches also inhibit inlining and/or post-inlining optimizations.

To avoid the shortcomings of `gprof`, we built a new profiler `csprof` [6]. Rather than relying on instrumentation in procedure prologues, `csprof` uses call stack unwinding to attribute samples to calling contexts and associate frequency counts with call graph edges. This approach has several benefits. First, `csprof` can be used to profile unmodified, fully-optimized programs without recompilation. Second, `csprof` records information about the full calling context rather than just call graph edges. Third, `csprof` does not incur overhead of instrumentation on every function call and thus neither incurs high overheads nor does it systematically distort measurements.

For `csprof` (or any other call-path profiler based on stack unwinding) to work properly on fully-optimized code, it must be able to unwind the call stack at *any* point in a program's execution, even when no frame pointer is used. Successfully unwinding from an event that occurred during a procedure epilogue requires precise information about the machine state at each point in the epilogue. This requires more information than is needed, *e.g.,* to unwind the stack in for C++ exception handling. In this paper, we describe the changes we made to GCC to emit this extra information, the size of this additional information, and we present results of experiments with `csprof` using this information on the x86-64 platform. In addition to changes to GCC, we believe that modernization of `binutils` to better support performance tools is in order.

The rest of this paper is structured as follows. Section 2 presents the high-level design of our call-path profiler `csprof`. Section 3 describes modifications we made to GCC to enable call stack unwinding of unmodified, fully-optimized code at any point in a program's execution. Section 4 compares the overhead and accuracy of `csprof` with that of `gprof` and reports how our changes to GCC affect the size of the unwind information it records. Section 5 briefly describes related work on open-source tools for call stack profiling. Section 6 describes shortcomings of the `binutils` library for performance tools and offers some suggestions for improving it. Section 7 presents our conclusions, ongoing work, and plans for the future.

2 Profiler Design

Our requirement for `csprof` was that it be easy to use with large, modern applications. Hence, it works on dynamically linked, optimized, unmodified binaries.[1] To initiate profiling, `csprof` instructs the dynamic loader (via the `LD_PRELOAD` environment variable or equivalent) to pre-load `csprof`'s profiling library. The library's initialization routine allocates and initializes profiler state and then initiates profiling. The library's finalization routine halts profiling and writes the profiler state to disk for later analysis.

`csprof` works with both asynchronous and synchronous events. Asynchronous events are not initiated by direct program action. They arise from interrupts triggered by the UN*X interval timer and/or hardware performance counter traps. Asynchronous events are monitored by setting up a signal handler to log each event and associate it with its call-path context.

[1] Profiling statically-linked applications is possible, but outside the scope of this paper.

Synchronous events are generated via direct program action. Examples of interesting events for synchronous profiling are memory allocation, I/O, and interprocessor communication. For such events, one might record bytes allocated, written, or communicated, respectively. Monitoring synchronous events typically involves having `csprof` use dynamic loading to override the relevant library routines and then to log information as appropriate when a monitored routine is called.

When synchronous or asynchronous events occur, `csprof` records the *full calling context* for each event. A calling context collected by `csprof` is a list of instruction pointers, one for each procedure frame active at the time the event occurred. The first instruction pointer in the list is the program counter location at which the event occurred. The rest of the list contains return addresses for each of the active procedure frames. We retain stack pointers as well to distinguish between recursive invocations. We have not observed excessive space requirements when retaining entire call stacks; if the storage of samples were to become a concern, we could collapse calling contexts for recursive procedure invocations [1] or record only a suffix of full contexts.

We store samples and their calling contexts in a *calling context tree* (CCT) [1]. In a CCT, the path from each node to the root of the tree represents a distinct calling context. Counts associated with events (e.g. cache misses, microseconds, bytes allocated) are attached to each node in the tree to associate the metrics with the calling context in which they were recorded.

Using a sentinel to limit unwinding. To ensure good statistical coverage of profiled code, one must collect a large number of samples, either by measuring over a long interval, or by using a high sampling rate. In either case, it is desirable for the unwinding and sample recording process be as efficient as possible. Performing a full unwind of the call stack at each sample event has the potential to be costly. To avoid this, we use a sentinel to mark the stack frames in existence when a sample event occurs. When processing the next sample event, we don't need to unwind frames below the sentinel since they have already been recorded.

We use dynamic execution state modification ("stack surgery") to implement stack sentinels in a general way for systems on which we control neither the compilers nor the calling conventions and stack frame layout. We mark a procedure frame with a sentinel by replacing its return address with the address of a *trampoline* function. `csprof` stops unwinding when it finds the trampoline as the return address of a stack frame. After the context of each sample event is recorded, if the frame currently marked by the sentinel is no longer the top stack frame, `csprof` unmarks that frame and marks the top stack frame instead. When a marked procedure returns through the trampoline, the trampoline moves the sentinel to mark the caller's frame before transferring control back into the caller.

In addition to using the sentinel to limit the depth of stack unwinding, `csprof` also memoizes previously inspected calling context by keeping a stack of pointers to the corresponding nodes in the CCT. Inserting a new sample in the CCT thus begins at the node corresponding to the sentinel frame rather than the root of the CCT. This reduces the number of memory references needed to record each sample.

Exposing calling patterns. Besides knowing the full calling context for each sample event, it is useful to know how many unique calls are represented by the samples recorded in a calling context tree. This information enables

a developer interpreting a profile to determine whether a procedure in which many samples were taken was doing a lot of work in a few calls or a little work in each of many calls. This knowledge in turn determines where optimizations should be sought: in a function itself or its call chain. To collect edge frequency counts, we increment an edge traversal count as the program returns from each stack frame active when a sample event occurred. We do this by having the trampoline increment a "return count" for the procedure frame marked by the sentinel as it returns. A more detailed description of this strategy can be found elsewhere [6].

Handling the complexity of real programs. The high-level design for `csprof` that we have described thus far suffices only for profiling simple programs with a single stack in memory that is modified only by procedure calls and returns. Real programs are often more complicated, featuring dynamic loading and unloading of code, register frame procedures, exception handling (including `longjmp`), and multiple threads of control. Many compilers, including GCC, generate tail calls for certain classes of function calls; these need to be handled specially by `csprof`. In addition, it is possible to receive events during execution of the trampoline or the sampler. The details of dealing with these complexities are outside the scope of this paper and are described elsewhere [6].

3 Adding Support to GCC

Our profiler design requires that the stack can be unwound from arbitrary PC locations during a program execution. This is a stricter requirement than the previous client for stack unwinding, namely, C++ exception handling. Exception handling requires only that the stack can be unwound from within a procedure body and never requires unwinds from within a procedure's prologue or epilogue(s). Most modern implementations of C++ use table-driven exception handling [5], where a small table describes the effects of instructions within a range of PCs. The effects recorded are only those that are necessary to unwind the stack properly, such as adjustments to the stack pointer, movement of the return address, in addition to register saves and restores. For example, GCC uses the DWARF2 format [13] for unwind tables. In practice, providing support for unwinding from within prologues is no more difficult than providing support for unwinding from within the body of a procedure.

Supporting unwinding from within epilogues is more complicated due to the way DWARF2 unwind information is interpreted. Conceptually, DWARF2 unwind information forms a table. For every instruction in a procedure, this table encodes rules for obtaining 1) the VMA of the "canonical frame address" (the stack pointer upon entrance to the procedure) and 2) the value on procedure entry of (callee-saved) registers. The rules are encoded as a byte-coded instruction stream. If rules for every instruction were included, the space consumed by the unwind information would be considerable. However, the matrix is generally sparse: the rules for many of the table rows are identical to those for the previous row. Therefore, only a limited number of instructions must have their effects encoded and `advance_loc` byte codes can be used to skip redundant rows. To compute the unwind information for any given instruction, the cumulative effects of all previous instructions in the enclosing procedure must be considered. Thus, to unwind the activation record from a given instruction, one iterates through the sequence of byte codes, applying effect rules until an address greater than the current instruction's VMA is found.

Because of branch and jump instructions, there

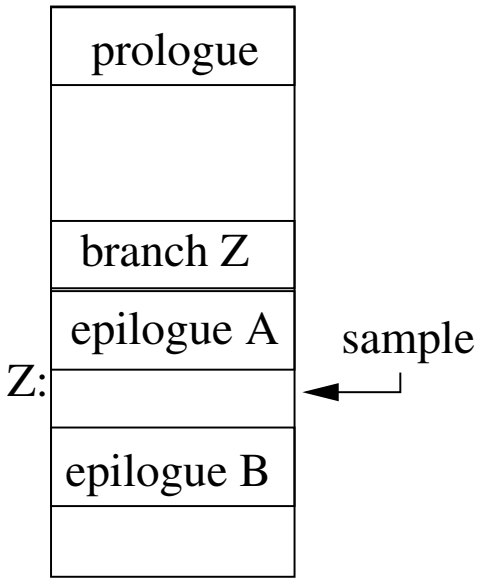


Figure 1: A routine with two epilogues, one of which is interior. Before the interior epilogue (A) there is a conditional branch instruction. We wish to begin unwinding from the sample point.

may be an inconsistency between this linear accumulation of instruction effects and the instructions that are actually executed. Consider an interior epilogue as depicted in Figure 1.[2] A linear scan of instruction effects prior to the sample point "ignores" the conditional branch before epilogue A and leads to the erroneous conclusion that all callee-saved registers have been restored and the current stack frame de-allocated as the scan progresses through the effects of instructions in epilogue A! DWARF2 provides two byte codes — `remember_state` and `restore_state` — that should "parenthesize" interior epilogues. When a DWARF2 unwinder sees a `remember_state` byte code, it conceptually pushes a copy of the current table onto a stack; when it sees a `restore_state`, it restores the table's state with the top of the stack and issues a pop.

[2] A compiler may generate an interior epilogue to improve instruction cache locality.

Our modifications to properly handle unwinding from within epilogues fall into three categories:

- Tagging frame-related epilogue instructions as such in the x86-64 back end;
- Support in the DWARF2 emitter for epilogues;
- Handling multiple epilogues with care.

3.1 Tagging instructions

The DWARF2 unwinding process is based on knowing the effects of "frame-related" instructions, namely, instructions that modify the stack pointer and/or save/restore registers. GCC's back end tags individual RTL pieces with the `FRAME_RELATED_P` tag to indicate their frame-relatedness. Adding the necessary `FRAME_RELATED_P` bits to the x86-64 back end was straightforward. As we worked on the DWARF2 emitter, however, we discovered two additional things that needed to be done.

When the DWARF2 emitter finds a `PARALLEL` RTX that is marked as `FRAME_RELATED_P`, it always processes the first of the child RTXs as if it is a `SET` and then processes subsequent RTXs if they are `SET`s and have the `FRAME_RELATED_P` flag set as well. In our initial attempt tagging, we only set `FRAME_RELATED_P` on the outer `PARALLEL` RTX, but not on any of the children of the node. This caused problems with instructions such as `pop`, which are naturally represented as `PARALLEL` RTXs. When correcting this, we also found that we had to modify the RTL definition of the `leave` instruction to be amenable to the DWARF2 emitter.

3.2 DWARF2 support

We implemented DWARF2 support in two steps. The first step was to fix GCC's DWARF2 emitter to handle instruction patterns commonly found in epilogues. The DWARF2 emitter knew how to handle register-to-memory moves, for example, but was ignorant about how to process memory-to-register moves. When processing the latter, we not only recorded effects on the stack pointer, but also encoded the DWARF2 byte codes indicating which registers had been restored. Extensions to the emitter to handle negative adjustments to the stack pointer and restoring the stack pointer from the frame pointer were also necessary.

The second step was to provide support for emitting the `remember_state` and `restore_state` byte codes. At first blush, this seemed simple enough: when processing a block, if an `EPILOGUE_BEG` note is encountered, emit a `remember_state` byte code. At the end of the same block, emit a `restore_state` byte code. However, we optimized this process so that when an epilogue block is the last block in the function, no `restore_state` byte code is necessary. This approach is similar to the strategy used by the IA-64 back end.

3.3 Multiple epilogues

In GCC's internal representation, epilogues are implicit, beginning with an `EPILOGUE_BEG` note and lasting until the end of the basic block. However, correctly handling epilogues requires that the relevant notes are actually inserted, which was not happening in two cases. The first was when a tail call, a.k.a. "sibcall," epilogue was generated, which was simple to fix. The second was when an epilogue block was duplicated; the `EPILOGUE_BEG` note was not duplicated as one might expect. Furthermore, when the initial non-sibcall epilogue is generated for a function, its instructions are inserted into a special `epilogue` array for use by the instruction scheduler. We found it necessary to insert the instructions of duplicated epilogues into this array to correctly handle repositioning of `EPILOGUE_BEG` notes. Additionally, when a duplicated epilogue was removed (e.g. by cross-jumping), `can_delete_insn_p` failed to indicate that the `EPILOGUE_BEG` note should be removed as well, presumably because multiple `EPILOGUE_BEG` notes were not a concern prior to our modifications.

Even when we ensured that `EPILOGUE_BEG` notes were consistently generated and duplicated as necessary, we found that certain passes of the compiler, particularly the instruction scheduler, assumed that only a single `EPILOGUE_BEG` note existed. To avoid littering the instruction scheduler with special cases for notes, any notes other than basic block boundary markers are removed prior to instruction scheduling and replaced afterwards. Since epilogue instructions may move across basic block boundaries, the instruction scheduler replaces `EPILOGUE_BEG` notes (`reposition_prologue_and_epilogue_notes`) by looking for the first instruction that is located in the `epilogue` array and positioning the note prior to that instruction. This is a flawed strategy when multiple epilogues are present.

We re-wrote the repositioning of the epilogue notes to correctly handle multiple epilogues. Our revised algorithm works as follows: We scan forward through all instructions,[3] looking for either an `EPILOGUE_BEG` note or an instruction that is contained in the "epi-

[3] Comments in `reposition_prologue_and_epilogue_notes` indicated that we cannot depend on the basic block structures maintained by GCC at this point.

logue" array mentioned earlier. When we find an `EPILOGUE_BEG` note, we move it to immediately before the next epilogue instruction that we find and skip to the end of that basic block. If instead we find an instruction that was in an epilogue, we search for the matching `EPILOGUE_BEG` note and move the note prior to the instruction. After doing so, we skip to the end of the basic block in which the note was located. By doing this, we ensure that all instructions that were in epilogues are "covered" by `EPILOGUE_BEG` notes and and therefore "covered" by the DWARF2 `remember_state` and `restore_state` byte codes described above.

3.4 Miscellanea

For our profiler to work properly, the application and all the libraries it requires must provide correct unwind information. It is preferable to have the compiler generate the necessary information, but we also plan to build a binary analysis tool to recover unwind information for legacy compilers that do not provide the necessary information. However, we have found that it is not simply sufficient to augment the compiler. While conducting the experiments described in Section 4, we found that assembly routines in `libc` were not properly annotated with unwind information. These routines required workarounds in the profiler similar to those already in place to detect samples taken within the trampoline. While these routines could have had their unwind information synthesized via binary analysis, it might be necessary for the assembly programmer to provide unwind information since binary analysis tools are not always successful at understanding the structure and semantics of machine code.

4 Experiments

Previous experiments [6] with `csprof` on Alpha/OSF1 demonstrated the viability of `csprof`'s approach for low-overhead profiling of unmodified, optimized binaries. However, the structure of the unwind information on x86-64 is very different from the unwind information on the Alpha. The Alpha uses a simpler, more compact unwind descriptor format that enables the unwinder to unwind using constant work per stack frame. We found, however, that the Compaq compilers did not always follow the published interface necessary for their unwind process to work correctly. Making `csprof` work correctly on a wide range of code required adding several work-arounds on the Alpha—in some cases as drastic as interpreting the instruction stream—to avoid problems caused by deviations from specifications. The workarounds for handling inaccurate unwind information contributed to profiler overhead, though overhead remained low despite them. Even though interpreting DWARF2 at run time would be more expensive than using Compaq's unwind information, we were optimistic that profiling overhead would be low on the x86-64 platform.

This section presents the results of experiments performed on the SPEC CPU2000 benchmarks [12] to gauge the impact of our changes on the size of compiled code as well as the effectiveness of `csprof` on the x86-64 platform. We ran our experiments on a 1.6GHz dual-processor Opteron with 8GB RAM using Gentoo Linux with the 2006.0 no-multilib profile.[4] We used GCC 4.1.0 with Gentoo's standard patches with options `-O3 -fomit-frame-pointer`

[4] We used Gentoo because it made it convenient to set up a `chroot` environment in which we had full control over how the system libraries were compiled. That the entire system was compiled with our modified GCC is a testament to its robustness.

`-funroll-all-loops -fno-tree-salias`. For unwinding call stacks within `csprof`, we used `libunwind` [10] version 0.98.5 with patches for x86-64 and an unwind cache for DWARF2.[5]

We encountered difficulties with several of the SPEC benchmarks. `255.vortex` would compile with both modified and unmodified versions of GCC 4.1.0, but would not run; we attribute this to some patch that Gentoo applies. `186.crafty` would not run consistently when compiled for profiling with `gprof` and we omitted it from Table 1. `csprof` was unable to profile `253.perlbmk` due to that benchmark's use of `longjmp`; `csprof`'s handling of `longjmp` on x86-64 is incomplete at present. Finally, `178.galgel` would not compile under the standard Gentoo compiler, which we again attribute to some patches Gentoo applies.

4.1 Profiling Overhead

This section reports measurements of `csprof`'s profiling overhead and compares it to that of `gprof`. We compiled each benchmark twice, with and without `-pg`.[6] To match `gprof`, `csprof`'s sampling frequency was set at 1000 samples/second. Results from the three runs are shown in Table 1. The runtimes in Table 1 are the average of five runs for each benchmark.

`csprof`'s overheads, shown in column three, are consistently lower than `gprof`'s overheads displayed in column two. Benchmarks such as `252.eon`, `253.perlbmk`, `168.wupwise`, and `177.mesa` have a high number of calls and a number of short procedures.[7] These sorts of programs demonstrate one of the primary problems with `gprof`: the overhead due

Integer programs

Benchmark	Runtime (seconds)	`gprof` overhead (percent)	`csprof` overhead (percent)
164.gzip	163	34	1.8
175.vpr	167	26	2.4
176.gcc	107	46	1.9
181.mcf	335	10	1.2
186.crafty	72	N/A	4.2
197.parser	281	50	1.8
252.eon	80	193	5.0
253.perlbmk	177	167	N/A
254.gap	128	174	1.5
255.vortex	N/A	N/A	N/A
256.bzip2	174	76	4.0
300.twolf	320	43	3.4
Average		82	2.7

Floating-point programs

168.wupwise	155	111	7.7
171.swim	260	15	3.1
172.mgrid	203	10	1.0
173.applu	256	25	1.6
177.mesa	124	74	1.6
179.art	210	3.8	1.4
183.equake	137	30	8.8
187.facerec	256	16	1.5
188.ammp	214	12	1.8
189.lucas	176	0	1.7
191.fma3d	264	28	1.1
200.sixtrack	235	1.7	0.9
301.apsi	259	16	3.5
Average		31	3.2

Table 1: Execution time overhead when profiling the SPEC CPU2000 benchmarks with `gprof` and `csprof`. A overhead of 100% indicates the monitored execution took twice as long.

[5] Available through a Mercurial repository at `http://www.serpentine.com/~arun/`

[6] Using the `-pg` flag required omitting `-fomit-frame-pointer`.

[7] We would expect `gprof`'s overhead on `255.vortex` to be high as well.

to instrumentation in prologues is unacceptably high in many cases. In contrast, `csprof` consistently has low overhead. Furthermore, `csprof`'s overhead can be reduced if necessary by lowering its sample frequency. The lone case where `gprof` has lower overhead than `csprof` is on `189.lucas`, which makes an extremely low number of calls. Even so, in this case `csprof`'s overhead is only 1.7%.

4.2 Space Overhead

`csprof`'s need for extra DWARF2 information for epilogues can increase the size of executables. Here we study this effect. For these tests, we compiled the CPU2000 benchmarks with Gentoo's vanilla GCC 4.1.0 package and our modified version of GCC 4.1.0 that records more complete unwind information. After doing so, we examined the size of the `.eh_frame` section, which contains the information for the DWARF2 unwinder. The results are shown in Table 2.

Benchmark	`.eh_frame` section size (bytes)	% increase
Integer programs		
164.gzip	2604	51.3
175.vpr	6324	51.4
176.gcc	60812	62.6
181.mcf	956	45.2
186.crafty	4004	45.4
197.parser	12284	55.3
252.eon	34940	28.3
253.perlbmk	31724	56.2
254.gap	30140	78.1
255.vortex	32580	42.3
256.bzip2	2524	41.2
300.twolf	7732	52.9
Floating-point programs		
168.wupwise	980	35.9
171.swim	420	38.1
172.mgrid	756	48.7
173.applu	916	50.4
177.mesa	30804	83.1
179.art	1100	53.8
183.equake	1020	36.9
187.facerec	1604	49.4
188.ammp	7268	59.3
189.lucas	364	44.0
191.fma3d	17676	45.7
200.sixtrack	9900	57.9
301.apsi	4804	47.0

Table 2: Sizes in bytes of the `.eh_frame` section of applications when compiled with "vanilla" GCC 4.1.0 and the percentage increase with our modifications.

Table 2 shows that increases in the unwind information hover around 50%, with a few outliers such as `177.mesa` and `254.gap`. One might expect that the size of unwind information might double since we now record information about epilogues in addition to the information about prologues already recorded. We see less than a 2x space increase for two reasons. First, the encoding of epilogue unwind information using `DW_CFA_restore`, which specifies when a callee-saved register is restored, takes two bytes (opcode and register). This requires less space than its counterpart in prologues, `DW_CFA_offset`, which uses three bytes (opcode, offset, and register) to define a stack position for a register. Second, epilogue information occasionally fills space that would otherwise be filled by `DW_CFA_nops` inserted to satisfy alignment restrictions.

It is important to note that even though the relative increase in the size of the unwind information is large, this change was a small impact on the total size of the binary file on disk. Across all applications in the SPEC benchmark, our modifications increase the size of the binary file by less than two percent. We feel that this is a small cost to pay for the ability to effectively and informatively profile any fully optimized binary with low overhead. Furthermore,

any application that might possibly need to unwind from within procedure epilogues (such as GDB) requires these modifications for correctness.

5 Related Work

Several call stack profilers are in wide use today. Apple's Shark [3] is a statistical call path profiler based on stack sampling. Like `csprof`, Shark provides full calling context for profiled costs. `csprof` goes one step further than Shark by recording return counts along edges, enabling more precise analysis of performance problems. Two well-known Linux call stack profilers, OProfile [8] and Sysprof [11], are system-wide profilers that require kernel-level support. Both do their call stack unwinding in the kernel and are limited to unwinding code compiled with frame pointers. This restriction almost certainly requires a recompile on x86-64, as the ABI for that platform does not require a frame pointer to be used. `csprof` has no such limitation.

Arnold and Sweeney [4] implemented a call stack profiler similar to `csprof` in Java. Since they controlled the run time environment and the compiler they implemented their sentinels by setting the low-order bit of the return address in every stack frame that their unwinder visited to indicate it had been seen. In subsequent unwinds only frames with their low-order bit cleared needed to be unwound. Their technique did not require any explicit compiler support, since the Java environment ignored the low-order bit of the return address when returning from a procedure. In Arnold and Sweeney's formulation, return counting could be done concurrently with the unwinding of the call stack during sampling—a "lazy" approach. Our technique gives us the freedom to choose between a lazy approach or an eager approach. We have chosen the eager approach because we already maintain a node into the CCT—to support efficient insertion of collected samples—and it is a simple matter to increment its return count in the trampoline.

6 A Call to Modernize `binutils`

We envision an overhaul of `binutils` shaped and informed by the needs of *performance tools*. We are *not* advocating a fundamental shift in the basic purpose of `binutils`; it currently is a collection of binary and performance tools. Rather we are advocating changes that would both improve the existing functionality and improve performance.

As a motivating example, consider the fact that we are not aware of a compiler that generates the complete unwind information required by `csprof`, even if the information is needed for correctness in other applications. While we would not object if other compilers generated such information—several non-GNU compilers generate `gprof` instrumentation—making `csprof` independent from compiler support and potential recompilation is an important goal. Therefore, we are interested in inferring unwind information using static binary analysis. Enhancing the `binutils` library in several ways would help us do this in a portable fashion.

6.1 Interface and Functionality

First, to infer what instructions affect the stack frame and register state, the `binutils` API must offer improved support for examining a binary's instruction stream and determining where procedures begin and end. Even though `binutils` supports the disassembly of binaries on a wide range of platforms, the interface

it exposes is specifically designed for printing disassembled instructions, not for examining or querying properties about all of the instructions in a procedure. A more general interface would not negatively affect the operation of `binutils` consumers such as `objdump` or GDB and could be exploited by a wide range of binary analysis tools.

Another example that motivates the examination of binaries is the `bloop` binary analysis tool, which is part of our HPCTOOLKIT [9] performance analysis tools. `bloop` presently uses a modified version of `binutils` that we locally maintain. Given an executable, `bloop` constructs the control flow graph (CFG) for each procedure, performs interval analysis to recover loop nests, and consults the executable's line maps to map VMAs to source line numbers. `bloop` then correlates the recovered program structure information with profile data to compute loop-based performance metrics in addition to the traditional procedure and line based metrics.[8] To reconstruct the CFG, `bloop` must 1) classify instructions (conditional branch, unconditional jump, return, non-control-flow) and 2) compute the target address of PC-relative conditional branches. Since we knew that `objdump`'s disassembler regurgitated nearly all of this information, we wondered if `binutils`' opcode library could help us. However, because `binutils` does not make this information accessible to its clients, we nearly abandoned it after being initially discouraged by its print-biased interface. However, after a clever suggestion from a former `binutils` maintainer, we were able to extract the information we needed with a small amount of precise surgery. The small modifications we made to the interface and to the relevant decoders made a significant difference in `binutils`' utility. We understand that `binutils` was designed to meet a specific need; our argument is that a careful and modest redesign would continue to meet the needs of its current clients while providing the support needed for sophisticated binary analysis tools.

[8] By performing loop analysis within a binary analyzer instead of a source-level tool, `bloop` is able to analyze binaries from multi-language code bases (e.g. Fortran95, C, C++) without multiple front-ends. More interestingly, with careful use of accurate debugging information, a binary analyzer provides information about the actual *optimized* code, not just the source code, enabling us to understand the effects due to loop transformations, software pipelining and loop fusion.

6.2 Performance Issues

A second major concern with `binutils` is algorithmic efficiency. Since a typical binary analyzer must examine every instruction in every procedure of a load module, algorithmic complexity is an important consideration. For example, consider our proposed tool for synthesizing unwind information from an executable. If `csprof` dynamically invoked such a tool in response to a runtime call to `dlopen`, the tool must run very quickly. While performing a linear search through the line table to map an instruction address back to a source line might be reasonable for a client like GDB when execution stops at a breakpoint, such an approach would be unsuitable for a tool that uses this interface to map each instruction in a large executable back to its source line. These are not theoretical concerns since executables can easily have hundreds of thousands of instructions.

To reduce the execution time of `bloop`, we had to replace `binutils`'s linear searches through the DWARF2 and ECOFF line tables with binary searches. Since `bloop` maps every instruction back to its source line, linear search of the line table caused execution time to grow quadratically with procedure size. For a similar reason, we added a one-element cache to the ELF function name lookup since sorting the symbols was a difficult solution. Attention to

algorithmic efficiency should also benefit current clients of `binutils`.

6.3 `binutils` vs. Custom Solutions

Performance analysis tools require access to a richer set of the symbolic and system information contained in typical binaries than currently exposed by `binutils`. For example, `bloop`'s analysis is complicated by nested procedures (e.g. Fortran 90) and by multiply instantiated statement instances (e.g. inlining). Since such information can be useful for debuggers, DWARF[9]—the nearly universal standard in the UN*X world—contains constructs for representing this information. Since most of this information was already read by the `binutils`'s DWARF reader, we wrote our own call-back routines to expose it in the `binutils` interface. In addition to symbolic information, binary analysis tools require access to the binary's system information, commonly represented as ELF in the UN*X world. We continue to need an easier way to access a binary's list of dependencies (the information retrieved by `ldd`) and to find the begin addresses of the segments that are mapped to memory during run time.

The references to DWARF and ELF raise the issue of portability, one of our initial goals for `csprof`. After all, the *raison d'etre* for `binutils` is to provide a common interface for a multitude of different ABIs. It also revisits the question of scope: what is the purpose of `binutils`? Why not use `libelf` and `libdwarf` to access specific ELF and DWARF information? Other performance tools authors have abandoned `binutils` in favor of creating their own binary interfaces for reasons related to the issues we have raised. We think, however, that a judicious redesign of `binutils` can reasonably accommodate the competing demands of, on one hand, generality and portability and on the other, improved and efficient support for the specialized and richer set of information encoded in ELF and DWARF.[10] Moreover, we argue that this is desirable because ELF and DWARF are such widely used standards, in fact *the* standard on Linux/GNU systems. A modernized and portable `binutils` that provides greater access to the instruction stream, emphasizes efficiency, and exposes more of the symbolic and system information (with a particular bias towards DWARF and ELF), would both improve `binutils` and provide an excellent platform for the development of a first-class suite of performance tools.

7 Conclusions and Future Work

We have presented `csprof`, a low-overhead call stack profiler and the modifications necessary to make it work on GCC. Experiments on the x86-64 platform have confirmed our initial results from experiments on the Alpha platform and have given us confidence that `csprof`'s approach offers a portable method for low-overhead context-sensitive profiling. Our methods are easily enabled by modest compiler support; however, to our knowledge no other compiler generates the necessary DWARF2 byte codes for `csprof` to unwind fully optimized code at arbitrary points, which is necessary with an asynchronous sample source. Binary analysis can provide the necessary information, but is not foolproof. Therefore, we would very much like our modifications to GCC to be integrated into the mainline compiler as both sup-

[9] We use 'DWARF' to include both DWARF2 and DWARF3.

[10] Note that we are not addressing the issue of binary modification. While this is an interesting area, support for these techniques would genuinely be a new concern for `binutils`.

port for our methods and to motivate other compiler groups to provide similar DWARF2 information.

Distributed in the `binutils` package, `gprof` has for many years offered portable, context-sensitive profiling for performance analysis and naturally complements GCC. To better support profiling of fully optimized modern object-oriented applications, we have designed `csprof` to deliver low-overhead profiles with full calling context for costs. Just as `gprof` required compiler support to make profiling easy to use, we have extended GCC to generate the unwind information required by `csprof`. We believe `csprof` should be a key component of a modernized `binutils`.

However, there remains work to be done before our work could be considered ready for GCC mainline. As Andrew Haley pointed out on the `gcc-patches` list,[11] we would need to ensure that our modifications worked with the x86 back end as well. We are in the process of satisfying this requirement, as our changes cause issues while performing the register-to-stack conversion pass necessary for the x86. In addition, our modifications prevent GCC's code reordering from working, as generating the DWARF2 `advance_offset` byte code requires taking the difference of two assembly code labels—an operation that fails when those labels reside in different sections. We are investigating options for alternate approaches for the necessary byte code generation. Finally, Haley also noted that with our patches, only the x86 and x86-64 back ends are capable of generating this extra information for unwinding. He felt that this sort of information should be generated consistently across all of GCC's back ends, as appropriate. While we agree with this point in principle, we feel that integrating our work is merely an enhancement to the x86 and x86-64 back ends and the necessary work for other back ends can be done as time permits.

[11] `http://gcc.gnu.org/ml/gcc-patches/2006-03/msg00426.html`

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, New York, NY, USA, 1997. ACM Press.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[3] Apple Computer. Shark. `http://developer.apple.com/performance/`. 14 April 2006.

[4] M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. Technical Report 21789, IBM, 1999.

[5] C. de Dinechin. C++ exception handling. *IEEE Concurrency*, 8(4):72–79, 2000.

[6] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *ICS '05: Proceedings of the 19th annual International Conference on Supercomputing*, pages 81–90, New York, NY, USA, 2005. ACM Press.

[7] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN*

Symposium on Compiler Construction, pages 120–126, New York, NY, USA, 1982. ACM Press.

[8] John Levon et al. OProfile. `http://oprofile.sf.net/`. 14 April 2006.

[9] J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–101, 2002.

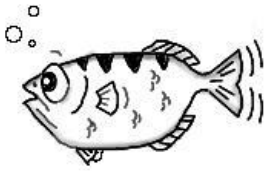
[10] D. Mosberger-Tang. `libunwind`. `http://www.hpl.hp.com/research/linux/libunwind/`. 14 April 2006.

[11] S. Sandmann. Sysprof. `http://www.daimi.au.dk/~sandmann/sysprof/`. 14 April 2006.

[12] SPEC Corporation. SPEC CPU2000 benchmark suite. `http://www.spec.org/cpu2000/`. 29 April 2005.

[13] UNIX International. DWARF debugging information format. `http://www.eagercon.com/dwarf/dwarf-2.0.0.pdf`. 29 April 2005.

Recent Developments in GDB



Paul J. Gilliam
IBM Linux Technology Center
pgilliam@ibm.com

The views and opinions expressed in this paper are those of the author, not of his employer, nor of his colleagues who comprise the GDB community. Any errors of fact or judgment are his alone.

Abstract

Many programmers on many platforms depend on GDB to help them find and fix bugs in their programs. Some of these programmers use GDB directly and others indirectly through one of several available graphical front-ends.

This paper summarizes major changes in GDB over the last few years, starting with the release of GDB 6.0, which was released 2003-10-06.

These changes were made in a number of areas, including:

- Support for additional architectures.
- Internal changes and reorganizations aimed at better supporting current and future architectures.
- New features to help programmers debug their programs.
- Enhancements to make GDB easier to use, both for those who use it directly and for those who write and maintain front ends.

Also included is an overview of the GDB community and how it operates, touching on a recent overhaul in the way the community interrelates to improve GDB.

1 Introduction

Many programmers on many platforms depend on GDB to help them find and fix bugs in their programs. Some of these programmers use GDB directly, while other use it indirectly through one of several available graphical front-ends, such as DDD from GNU or Xcode from Apple.

These programmers have one thing in common with respect to GDB: they all wish it were better. Of course, each of them has a different idea of what constitutes *better*, and what's good for

one platform may not be good for *all* platforms. These basic conflicts could have led to a warring and bitter group of users. Instead, it has led to a vibrant community that maintains and extends GDB.

One measure of the vibrancy of the GDB community is the topic of this paper. What recent developments have been made? The answer is "lots!" There will be more about the GDB user community later in this paper.

For the purposes of this paper, GDB activity will be divided into five parts:

1. **Platforms** This part covers target architectures and host environments (operating system and processor). When these two are connected as part of GDB, it is called a *native* debugger. Otherwise, the two are connected by some communication protocol and GDB is called a *remote* debugger.
2. **User-Level Features** These are features that are primarily aimed at the user. These will directly help the user debug their program. An example is support for a particular computer language.
3. **Under-the-Hood Features** These are features that are primarily aimed at some part of GDB itself. Hopefully, these will help the user, but in a less direct way. For example, doing a better job of understanding a particular format for debug information will help GDB do a better job, but may not be reflected in GDB's user interfaces.
4. **Deleted Features** These are features that are no more. They are ex features. Sometimes they are features that were meant to ease a migration from one internal scheme to another and the migration has been completed. Sometimes they are components of GDB that have lost their usefulness. The most natural (in the Darwinian sense) reason for a feature to be deleted is a lack of interest in the GDB community to maintain it.
5. **Commands** These are the most visible part of GDB to most users. If GDB is accessed through a GUI, these may not be visible at all. In either case, they are the basic building blocks of GDB's functionality. Commands are usually added to GDB, but occasionally they are deleted.

2 Platforms

It's kind of hard to know what a *platform* is when talking about GDB. This comes from the fact that GDB is used in different ways by different communities. For example, the imbedded community sees GDB as running on a particular host, debugging their code on a particular target. A typical developer will see GDB as running in the same environment as the program they are developing on and for.

To keep things simple for this section, I'm going to ignore all this and hope that the description used is enough to figure out these details.

When indicating the affected version of GDB, >6.4 means it will be in the 6.5 release of GDB, which had not yet happened when this paper was written, but *should* have happened by the time it is presented.

2.1 Added

Table 1 shows the platforms that have been added to the list supported by GDB.

2.2 Removed

The GDB community uses a two-step process to remove a platform. First it is made obsolete,

platform	GDB
Morpho Technologies ms2 (target)	>6.4
OpenBSD arm	6.4
OpenBDS mips64	6.4
GNU/Linux m32r	6.3
GNU/Linux hppa	6.2
OpenBSD m68k	6.2
OpenBSD m88k	6.2
OpenBSD PowerPC	6.2
NetBSD VAX	6.2
OpenBSD VAX	6.2
NetBSD amd64	6.1
OpenBSD amd64	6.1
OpenBSD alpha	6.1
OpenBSD sparc	6.1
OpenBSD sparc64	6.1

Table 1: Added Platforms

which means the code to support that platform is 'commented out.' Then after time has passed and no one has come forward to maintain it, the platform is completely removed from the source tree. (Of course, it's still in the CVS repository if someone wants to resurrect it). For the purposes of this section, I'll list the version of GDB where the platform was removed, or if it hasn't been removed yet, where it was obsoleted.

Table 2 shows the platforms that have been removed.

3 New or Improved User-level Features

It seems that the life-blood of many commercial software packages is new features or newly improved ones. This is driven by the need for new sales of the same product to the same customers. GDB does not feel this force because it is Free (as in Freedom). The force that drives new or improved user-level features in GDB is provided by the programmers willing to do the work (or by people paying someone to do the work) to make GDB better.

A note of caution: Not all the features discussed in this section or the next are universally available. Some are limited to particular operating systems and/or particular processors and only native or remote. I have tried to indicate where that is the case, but I may not have been completely successful.[1]

3.1 Checkpoints

This is a really cool feature: it lets a user select a point in time during a debugging session and mark it as one that can be gone back to later. The concept of checkpoints has been around for a long time and still has the same underlaying motivation: "If things go bad, I don't want to have to repeat every step I took to get there." Ideally, we would like to halt on an error and 'undo' back to a good state, but we can't do that.[2]

When you reach a point in your debugging that you might want to go back to, use the command `checkpoint`. GDB will tell you the number of the new checkpoint, just like it tells you the number of a new breakpoint. Now, after some more debugging, you want to go back to the way things were: use the command `restart` *id*, where *id* is the id number GDB assigned when you used the `checkpoint` command. With the `restart` command, the state of the target program is restored to what it was at the time of the corresponding `checkpoint` command. Then you can try something else.

[1] See Section 7.

[2] Yet: see Section 7.

platform	GDB	platform	GDB
Motorola MCORE	6.4	National Semiconductor NS32000	6.4
VxWorks and XDR protocol	6.4		
Ah8300	6.3	mn10300	6.3
sh64	6.3	v850	6.3
Sun 2, running SunOS 3	6.2	Sun 2, running SunOS 4	6.2
Sun 3, running SunOS 3	6.2	Sun 3, running SunOS 4	6.2
T386BSD	6.1	AT&T 3b1/Unix pc	6.1
Bull DPX2 (68k, SVR3)	6.1	decstation	6.1
Fujitsu SPARClite	6.1	H8/500 simulator	6.1
HP/PA Pro target	6.1	HP/PA running BSD	6.1
HP/PA running OSF/1	6.1	LynxOS	6.1
Matsushita MN10200 w/simulator	6.1	Motorola 680x0 running LynxOS	6.1
PMAX (MIPS) running Mach 3.0	6.1	riscos	6.1
Sequent family	6.1	SGI Iris (MIPS) run. Irix V3:	6.1
SGI Irix-4.x	6.1	sonymips	6.1
SPARC running LynxOS	6.1	SPARC running SunOS 4	6.1
SunOS 4	6.1	sysv	6.1
sqware Sparclet	6.1	Z8000 simulator	6.1
Argonaut Risc Chip (ARC)	6.0	Fujitsu FR30	6.0
HP/Apollo 68k Family	6.0	i386 running Mach	6.0
i386 running Mach 3.0	6.0	i386 running OSF/1	6.0
I960 with MON960	6.0	IBM AIX PS/2	6.0
Mitsubishi D30V	6.0	Motorola Delta 88000 run. Sys V	6.0
OS/9000	6.0	V850EA ISA	6.0

Table 2: Deleted Platforms

Other commands allow you to manage checkpoints. The `infocheckpoints` command tells you what checkpoints have been saved, listing for each checkpoint: its number, `pid`, and source line number or label. The `delete-checkpoint` *id* command will delete the checkpoint numbered *id*.

There are some restrictions, of course. I/O can not be 'taken back.' Data written to the disk will not be erased, for example. File pointers are, however, 'rewound' to their values at the time the checkpoint was taken. Another potential problem is that each checkpoint will have its own `pid`. When the `restart` command is used, the restored program will not have the same `pid` as before.

This only works for a native GDB on a GNU/Linux system.[3]

3.2 Internationalization

When supported, GDB will be built with internationalization (libintl). Not all the necessary mark up is complete, but it's getting done and such mark up is now the way *things are done*. So all that's needed now are translations: any volunteers?

[3]For now. See Section 7.

3.3 Ada

Support has been added for debugging `ada` programs compiled with the GNAT compiler. Currently, support is limited to expression evaluation, but it's a start.

3.4 Pending Breakpoints

A *pending* breakpoint is one for which a valid address can't be found right now, but will be in the future. A straightforward example would be setting a breakpoint in a shared library that has yet to be loaded. When the library *is* loaded, GDB will be able to find the valid address for the breakpoint. Then the pending breakpoint is removed and a *real* breakpoint is created.

3.5 Objective-C

GDB fully supports the Objective-C language. This should be no surprise because the 'Next' system, from back in the late 1980's, used the GNU toolchain to build its software, most of which was in Objective-C. Why then is this a 'recent' development? Back then there was no GDB community as we know it today. 'Next' had their own GDB, Tektronix had their own GDB (I ported it to the M88000/SysV environment while at Tek), and the FSF had the 'real' version.

3.6 Processes

GDB did not deal with multiple processes very well: if your program did a fork, you had to make sure that the new process would sleep long enough to get to another terminal, start another GDB and `attach` to the new process. The next step was to introduce the `set|show follow-fork` commands so that the user could decide which process, the parent or the child, would be the one GDB continued to debug. The one not being debugged would simply keep running. This only works on a couple of operating systems: HP-UX and GNU/Linux.

Now the fate of the fork not being debugged can be decided using the new `set detach-on-fork` command. If set to `yes`, then the fork will detach from GDB and run independently. But if set to `no`, then the fork will stay in the stopped state and join the list of forks being debugged. The command `info forks` shows you the `id`s of the forks currently being debugged. The command `fork` *id* tells GDB to stop debugging the current fork, leaving it stopped, and start debugging form *id*. To remove a fork from the list being debugged, either use `detach-fork` to let the fork run on its own after removing it or use `delete-fork` to kill the fork after removing it. This only works on GNU/Linux.[4]

3.7 Text-mode User Interface

The GDB Text User Interface, TUI for short, is a terminal interface which uses the `curses` library to show the source file, the assembly output, the program registers and GDB commands in separate text windows.[5] What's new is that this is now a run-time option where it used to be a build-time option.

3.8 Convenience Variables/User Defined Functions

Convenience variables are used in GDB either for the user to save things they are interested in or for GDB to make some value available.

[4] So far, see Section 7.

[5] Cut-n-pasted from `gdb.textinfo`.

For example, the user may want to remember a particular index in an array while stepping through a loop: `set $foo = bar` where `$foo` is the convenience variable and `bar` is the index variable for the user's program. You could use a value in a processor register in an expression like this `x12/g $r2*8+base` which tells GDB to print the twelve 64-bit integers in the array `base` indexed by register `r2`.

So what's new? Two things are new, both designed to help users who write their own GDB commands.

The first one is a new convenience variable, `$argc`. If you guessed that this variable contains the number of arguments to the user-defined command, you would be correct. Using this, it is now possible (easier?) to write user-defined commands that take a variable number of arguments.

The other new thing provides a way to initialize a convenience variable if it does not already have a value. Normally, the first time a convenience value is used, it has a special 'void' value. Using the `init-if-undefined` *$variable = expression* command, you can now create a convenience variable that starts with a value of your choice.

4 New or Improved Under-the-Hood Features

The features in this section are not the ones the sales force cares about. They are not glaringly visible to users, even though they may have a big impact. From a user's point of view, these features are more behind the scenes.

4.1 Threads

Support for threads has gotten a lot better. For example, GDB used to get confused by programs that do a lot of thread creation and deletions. Several flavors of threads are now supported: NPTL threads, linux threads (on GNU/Linux), and BSD user-level threads (on OpenBSD and FreeBSD). Per-thread variables (aka thread-local storage) is now supported on GNU/Linux.

4.2 GDB/mi

The GDB/mi interface (interpreter in GDB parlance) is the new way for GDB to be used as part of a larger system, such as a GUI or IDE, DDD and XCode. The GDB/mi was introduced with GDB 5.0, so it's not new. But it is vastly improved and as more experience is gained using it, it gets better and better. There have been three versions of this interface, but only the two most recent are still available. MI3, the most recent, is now the default.

4.3 BSD libkvm Interface

Using `set target kvm`, when running native on a BSD flavored OS, allows debugging of kernel core dumps and even live kernels, though only on a few processors: i386, amd64, m68k, and sparc. For GNU/linux, another approach was taken. There, a stand-alone variant of GDB, called KDB, is used for kernel debugging.

4.4 Windows Host Support[6]

GDB runs on MS Windows, either with Cygwin or MinGW. GDB support for both of these environments has had improvements. For Cygwin, see Section 4.6 below.

[6] Lifted whole from GDB's NEWS file.

GDB now builds as a cross debugger hosted on i686-mingw32, including native console support, and remote communications using either network sockets or serial ports

4.5 Remote Debugging

GDB has had a number of improvements in this area. Below are a few examples.

It has been possible for a while to set the communication time-out using the `set remotetimeout` *n* command, where *n* is the number of seconds to wait before giving up on reading from a target. Now, the time-out value can also be set when GDB is started using the new `-l` *n* command line option.

Before the new `p` packet was introduced as part of the remote protocol, target registers had to be read from the target in groups. Now using the `p` packet, GDB can read a single register from the target. Combine this improvement with the register cache and communication with the target is much more efficient and hence faster.

Another change to the protocol allows hosted file I/O. This is where target programs access files in GDB's filesystem via the remote protocol.

4.6 Improvements in Dwarf Support

The DWARF 2 standard for debugging information has had a profound impact on GDB, but it didn't happen all at once. In fact, GDB's support for DWARF 2 is getting better all the time. One example is that GDB built for Cygwin now supports DWARF 2.

Another improvement is support for DWARF 2 location expressions. These are used by DWARF to tell the debugger where to find the value of a given variable. This used to be easy, but now with different optimizations performed by the compiler, the location of a variable's value can change during the execution of the target program. Location expressions allow the compiler to tell the debugger how to keep track.

One complaint lodged against DWARF is that even though great care was taken by the DWARF committee to keep DWARF's 'footprint' small, DWARF can add tremendously to the size of an executable. One answer to this is the new `-feliminate-dwarf2-dups` flag to GCC and GDB's new support for it. Using this, GCC will try and pare down the duplicate information it puts out in the DWARF sections. GDB was modified to deal with the new DIEs[7] GCC uses under this flag.

Another answer to the "debugger information bloat" problem works with other debugger information formats as well. In conjunction with BINUTILS, GDB now supports debug information in separate files. Now, for example, library packages can be distributed without debug info, making them much smaller. The debug information can be in a separate package which is only installed if needed.

4.7 Improved C++ Support

Support for debugging programs written in C++ has been improved in a couple of areas. For one, GDB has a new C++ name demangler. Not only does it do a better job demangling the names produced by newer versions of GCC, but it does so faster than the old one. This can substantially reduce the start-up time when debugging a large C++ program.

Another improvement is with support for C++ nested types and namespaces. GDB now understands that the namespace and/or outer type

[7] Debug Information Entries

must be included, using the scope operator, in the type or function's name.

4.8 Strictly Internal Improvements

In order to increase GDB's reliability, maintainability, and all the other goodabilities, major sections of the code have been overhauled or re-architected.

The code that deals with signal trampolines has been overhauled. This has fixed many problems GDB was having in this area. A couple of these problems were that GDB had trouble showing a correct backtrace from inside a signal handler and had trouble single stepping through a signal trampolines.

Speaking of backtracing, a whole new mechanism was created to help GDB do that. One major improvement that this made possible was to use DWARF 2's call frame information. It also makes it easier to separate out target dependent heuristics and makes the whole approach to backtracing more robust and modular.

In the same vain, a new framework for supporting different architectures was added to GDB. This helps GDB keep track of all the different target architectures it supports. It also paves the way for two trends GDB is taking: object oriented design and muli-arch support.[8]

5 Deleted Features

The features in this section, whether user-level or under-the-hood, are no more. They are ex features. All that's left to say about them is what they were and why they got the axe.

[8] See Section 7.

ARM `rdi-share` module[9] RDI is an ABI standard for ARM hardware debuggers and simulators, giving you access to the full processor state. In rdi-share, there was support for building a gdb that could talk to an RDI DLL and use that as a target.

IIRC it was removed because it was outdated and its legal status was unclear, given that ARM ltd. gives you the necessary headers under NDA only.

Netware NLM debug server It's sometimes hard to remember that Novell had the corner on networking PCs: Netware was the best (only?) way to do it. Now other networking code has pretty much taken over and the need to debug Netware Loadable Modules (NLM's) has gone away.

command line options `-async` and `-noasync|` Once upon a time, GDB had no "event loop": it was totally synchronous, you typed a command and GDB would go away and do it. These command options were an attempt to be able to specify that behavior, even after GDB became more event driven. But now, that's just not possible any more and these options have been removed.

registers and frame compatibility modules When there is a major internal change to the way GDB does something, like access registers, or deal with the stack, a 'bridge' is provided so that all the different configurations that GDB supports don't have to all change at once. But there comes a time to burn the bridge and move on.

[9] Taken from an e-mail from Simon Richter. Thanks Simon, I didn't have a clue.

6 Commands

Some of these commands have been discussed above: they are included here as well for completeness.

6.1 New Commands

checkpoint This command creates a checkpoint and tells you its id. At some later point you can use that id to restore your program's state back to what it was when you issued this command.

restart *id* Restore your program's state back to what it was when the checkpoint with the given *id* was created.

info checkpoints Show information, including id, about all the checkpoints that have been taken and not deleted or detached.

delete-checkpoint *id* Forget about *id*.

set detach-on-fork [on or off] If set to `on` then GDB will detach from the parent or child after a fork, depending on what **follow-fork** is set to.

show detach-on-fork Show if `detach-on-fork` is set to `on` or `off`.

info forks Tell what forks are available for debugging.

fork *id* Switch from debugging the current fork, leaving it in the stopped state, and start debugging the fork with the given *id*.

delete-fork *id* Delete the given fork and kill the associated process.

detach-fork *n* Detach from the given fork, letting its associated process run independently.

init-if-undefined *$variable* **=** *expression* Set the given variable to the given value, but only if the variable has not been used before.

set print array-indexes When array element values are displayed, their index will also be shown.

set logging [on or off] If logging is set `on`, then GDB's output will be written to a log file, as well as displayed.

set logging file *name* Set the logging file name to *name*. the default is `gdb.txt`.

set logging redirect [on or off] Normally, when logging is on, GDB output will both be displayed and written to the log file. If this is set to `on`, then the output is sent only to the log file.

set logging overwrite [on or off] When logging is turned `on`, the log file will be be overwritten if this value is `on`, otherwise, it is appended to.

show logging Show if logging is set `on` or `off`. Also show if the log file is, or would be, overwritten or append to. Also show if GDB's output is to be displayed or just written to the log file. And throw in the name of the log file too.

start *arguments* This is the same as setting a temporary breakpoint at "main" and then issuing the "run *arguments*" command.

disconnect When `detach` is used to stop debugging and disconnect from the target, the target is allowed to run independent of GDB. `disconnect` works the same way except that the target is not allowed to run. It just sits there, waiting for someone to attach to it, and start debugging it. This would let you debug the child and parent with different GDBs after a fork. Or

maybe even debug with two altogether different debuggers.

`maint set profile [on or off]`
GDB has code built into it so that it can be profiled. This is used to turn `on` or `off` that code and is used to study the behavior of GDB itself and has nothing to do with the target.

6.2 Deleted Commands

Table 3 shows commands that have been deleted along with the commands that replace them.

7 The GDB Community and Future Trends

Why does this section cover two different things? Because they are so tightly linked. The future of GDB depends on the community of software freedom fighters that are involved with it. In a simple view: no community, no GDB development.

A more complex view: the impetus for change comes from two different sources. One source is the individual who wants something fixed, changed, or added and is willing to do the work, (or pay someone else to). The attitude of the community is that if you want something and are willing to do the work to get it, then go for it. Of course there are limits. Those limits are determined and enforced by a set of official 'maintainers.'

There are two types of maintainers and just recently, a third type was added. There are 'global' maintainers who are free to OK patches to any part of GDB. There are also what might be called 'areas of interest' maintainers. These maintainers can reject or OK patches to particular areas of GDB. The new type of maintainers are patch champions. Their role is to make sure that no patch gets lost. Patches can be rejected, but thanks to the patch champions, not by default.

So what if you have a good idea for some specific, but can't do the work yourself, or you see a need for a bigger change then one person could do? This is where the "group mind" aspect of the GDB community comes into play.

Someone tries to start a dialog in the GDB mailing list, maybe just to call attention to something that needs some. If the message catches someone else's interest, then they reply to the mailing list, and the discussion has begun.

The discussion can end in one of three ways:

1. the community reaches consensus that it should be done, now who has the time to do the work?
2. the community reaches consensus that it should not be done. Sorry.
3. the community is leaning toward option one above, but before they reach consensus, someone gets tired of all the discussion and says "I'll just do it, OK?" and the community replies "Fine, but if we don't like it when you're done, it won't become a part of GDB."

Sometimes an idea or request can stay in option one for a long time before someone has time for it. Or sometimes it is such a big or far reaching idea that it needs to be done in steps. It is mostly by this path that the community develops trends for the future.

So what are some of the future trends?

deleted command	replacement
`set|show arm dissembly-favor`	`set|show arm disassembler`
`othernames`	`set arm disassembler`
`set|show remotedebug`	`set|show debug remote`
`set|show archdebug`	`set|show debug arch`
`set|show eventdebug`	`set|show debug event`
`regs`	`info registers`
`set prompt-escape-char`	- none -

Table 3: Deleted Commands

propagate Several really cool things only work on one OS or with a small number of processors and need to be propagated to *all* configurations. This will not always be possible due to hardware and/or software incompatibilities. But things like kernel bugs can be fixed and should be if that's the problem.

separate Some features that could logically be independent, like the MI and the CLI, are not quite. They should be separated and made independent. The next trend should help with that.

objectify GDB is slowly evolving into an object oriented design. Thanks to the "observer" mechanism, "Catch and throw" and a bunch of other stuff Cagney and others did, GDB looks a lot like a "coarse grain" object oriented design. More is needed. A more complete object orientation will make GDB much easier to deal with as a **large** piece of software.[10] Generalizations like "multi-arch" will be easier to do. A "multi-arch" GDB will work with any architecture GDB knows about, not just the one it was built for.

it's a given There will be new processors, new operating systems, new languages (natural and computer), new compilers, etc.

optimized code GDB has been getting better in dealing with optimized code but still has a ways to go. The mailing list often gets messages asking "why does the line number keep jumping around when I step through my code?"

blue sky Some ideas start life as "blue sky" ideas. They are real cool, but they will never be practical. Then some new piece of technology comes along and, just like that, the idea isn't so far fetched anymore.

A current Blue Sky idea for GDB is "reverse execution." This would be like having a target with a reverse gear. Every instruction could go forward or backward, performing a computation or undoing the results of one. Some believe that a processor is coming "soon" that will make this possible. In the meantime, we can talk about how GDB would deal with such a processor and maybe even hack one of the simulators so that the idea can be experimented with.

And if we can't go back instruction by instruction, maybe we can go back by some larger amount: maybe back to some checkpoint. Oh right: already did that. So now, what's the next step? Better send a query to the GDB mailing list and see if it catches anyone's eye.

[10] About 1.75 million lines of code

Profile driven loop transformations

Richard Günther
SUSE Labs
rguenther@suse.de

Abstract

Today scientific computing applications are developed using modern principles of software design. Among others, this leaves specialization and optimization of loop kernels to the compiler. In particular, loops which run for a known low number of iterations in one of the dimensions, such as loops handling boundary condition computation, usually produce inferior code using F95 array expressions or C++ template library utility functions. The same holds true for strides of multidimensional arrays which usually are the same for all arrays that participate in a loop kernel, but still are not known so at compile time.

GCC has developed a rich infrastructure for both loop analysis and transformation[1] as well as supporting profile guided optimizations[3]. Using this infrastructure we present the results of developing loop optimizations that rely on the use of loop versioning and profiling of iteration counts and access evolution. The goal is to reduce the numbers of induction variables to consider during induction variable optimization and improve the generated code by requiring a less overall number of registers. This is done by providing loop specializations for both the above mentioned cases. We present the implementation of the instrumentation and the optimizing phase discussing current limitations of the framework GCC provides. A case study involving the TraMP3d benchmark to gather statistical data for the transformations is presented as well as performance results for applying the transformations on a standard loop kernel.

1 Introduction

There are two different species of profiles, *CFG profiles*, which profile for instance edge execution counts, and *value profiles*, which profile for instance the value of the dividend in a division instruction.

A CFG profile can be used to direct inlining decisions such as emphasizing inlining into hot sections of a program and keep cold sections optimized for size. It also is used to estimate branch probabilities to guide partitioning of hot and cold code sections and basic block reordering to improve code locality and instruction cache efficiency. A CFG profile can to some extent also be used to estimate loop iteration counts to guide optimizations such as loop unrolling and peeling, though in general CFG profile info is too imprecise here.

Value profiles on the other hand are used to decide whether specializations of computations are worthwhile, such as specializing a division instruction for a constant divisor or dividend. In general value profile using optimizations increase code size for introducing a new common

faster path based on knowledge of certain values used in the following computation.

A simple example for a value profile transformation is the instruction

```
c = a/b
```

which can be transformed into

```
if (b == 1) c = a;
else c = a/b;
```

based on profile information that `b == 1` is indeed common.

In the following, we will use value profiles of variables and predicates composed from quantities used in nested loop code to create specializations of these loops which can then be optimized by later passes.

1.1 Interesting Loops

We are interested in a certain type of loops. This is not necessarily a requirement of the instrumentations and transformations we outline, but we will restrict further discussion on nested loops of the form

```
for (iN = i0N; iN < i1N; ++iN)
 ...
  for (i0 = i00; i0 < i10; ++i0)
    f (mem0[(i0 + C00) * stride00
            + ... + (iN + CN0) * strideN0],
       ...,
       memM[(i0 + C0M) * stride0M
            + ... + (iN + CNM) * strideNM]);
```

That is, loops of the nest N which have an inner loop body that is a function of M memory reads or writes with possibly different access patterns specified by the strides stride00 to strideNM and the constant offsets C00 to CNM. Interesting loops are required to have loop invariant strides, loop bounds and memory base addresses.

GCC has infrastructure to analyze and identify loops of the above form and canonicalize them so that the above constraints can be verified by looking at the SSA web. In particular, the scalar evolution infrastructure[1] can be used to verify loop invariantness and to query the number of iterations of each loop of the nest. It also provides a way to decompose memory access patterns to the form outlined above.

1.2 Transformations

The set of transformations we are after is inspired by the TraMP3d[2] hydrodynamics code. TraMP3d is based on the FreePOOMA[4] library which provides data-parallel array operations similar to Fortran90+. As those go through a common loop expander template function, common degenerate cases are worth to optimize. The loop expander template for three dimensions looks like the following

```
template <class LHS, class Op, class RHS,
          class Domain>
inline static void __attribute__((flatten))
evaluate(const LHS& lhs, const Op& op,
         const RHS& rhs, const Domain& domain)
{
  int e0 = domain[0].length();
  int e1 = domain[1].length();
  int e2 = domain[2].length();
#pragma omp parallel for
  for (int i2=0; i2<e2; ++i2)
    for (int i1=0; i1<e1; ++i1)
      for (int i0=0; i0<e0; ++i0)
        op(lhs(i0,i1,i2), rhs.read(i0,i1,i2));
}
```

Here memory access patterns such as strides and constant offsets are encapsulated in the `lhs` and `rhs` expression template[5] objects.

Let `op` do a simple assignment of the `rhs` object to the `lhs` object. Further be `lhs.strides` an array specifying the strides used to access the `lhs` memory, and `rhs.strides` for the `rhs` memory. `domain.size` should be an array specifying the size of the domain to iterate over. We are then interested in the following specializations being done:

(a) lhs.strides[0] == rhs.strides[0] == 1

(b) lhs.strides[0] == rhs.strides[0] == 1 and domain.size[0] == 2

(c) lhs.strides[0] == rhs.strides[0] == 1 and domain.size[1] == 2

(d) lhs.strides[0] == rhs.strides[0] == 1 and domain.size[2] == 2

Here (a) covers copying of unit stride array regions, and (b) to (d) cover periodic boundary updates. In addition to these specializations, reflecting boundary condition updates would be optimized by versioning for lhs.strides[0] == 1 and rhs.strides[0] == -1 and the respective domain size conditions.

By providing specializations for these cases, we rely on the following optimization passes to take advantage about the additional knowledge.

- Induction variable optimization is presented with a problem reduced in complexity due to the now partially constant strides.
- The linear loop transformation pass can decide to move the low-iteration count loop either to the outermost or the innermost nest, which allows
- either loop unrolling to unroll the innermost loop completely,
- or induction variable optimization to consider the most expensive updates for the outermost loop to improve induction variable selection for the inner nests.

2 Preparation

To be able to use the infrastructure mentioned above we need to first do some cleanup transformations on the loops, namely loop header copying, which transforms the loops into do-while loops, and loop invariant motion by using load-PRE to make loop invariant memory references regular SSA variables in the SSA web of the loop nest. To achieve this, we have moved the tree profiling pass to a later point in the optimization pipeline and inserted a set of optimization passes before it. The relevant part of the optimization pipeline now looks like (inserted passes marked with *):

Initial scalar cleanups:

```
pass-ccp
...
pass-dce
```

Kill empty loops getting in the way of loop analyzing. The WrapNoInit template leaves us with empty loops counting from 2 to -1 otherwise.

```
  pass-tree-loop-init
* pass-empty-loop
* pass-complete-unroll
* pass-tree-loop-done
```

The VRP pass above exposed new forwprop opportunities (which in turn exposes copyprop opportunities) due to folding casts again. And another may-alias pass to expose the store copyprop opportunities to DOM.

* pass-forwprop
* pass-may-alias
  pass-dominator
  ...
  pass-ch

We need a load-PRE pass to hoist loads of loop invariant strides and counts out of the loop bodies. Possible due to loop header copying.

* pass-split-crit-edges
* pass-pre
* pass-may-alias
* pass-hoist-guards

We need a copyprop pass to have the same SSA names for loop tests as the hoisted loads from PRE.

* pass-rename-ssa-copies
* pass-copy-prop
* pass-dce
* pass-tree-loop-init

Try getting rid of extra PHIs inserted by loop-init.

* pass-phi-only-cprop
  pass-tree-profile

The early loop pass is to get rid of unrelated inner loops in the TraMP3d benchmark, likewise the extra forward propagation and may-alias passes are to expose SFTs of array elements to the following DOM pass doing store copy propagation.

Entering the tree-profile pass, a properly optimized loop looks like that in Fig. 1.

From the optimization pipeline you can see that we needed to make the tree profiling code work on SSA form. This was necessary anyway because the infrastructure for loop modification and the SCEV analysis requires SSA form. In the current state profiling is now done after inlining, so profile-based inlining is disabled. After the merge of the IPA-branch it will be possible to place the early optimizations and the profiling before the final inlining pass, which is then done on SSA form.

2.1 Limitations of the current infrastructure

The current profiling and loop infrastructure presents us with several limitations that have been partially addressed for this work. First of all, the existing profile instrumentation pass does not work on SSA form, while SCEV analysis and all optimization passes require that. This has been fixed and allows moving the `tree-profile` pass to a later point in the optimization pipeline.

SCEV analysis for figuring out the number of iterations of a loop needs to be improved to deal with more cases that happen for example in TraMP3d. The situation with this has been improved by us and Sebastian Pop. For example we were not able to determine the number of iterations of the loop for (int i=i0; i<=i1+1; ++i);, which runs i1-i0+2 times if it runs at all.

Loop header copying, SCEV and load-PRE interact in interesting ways, in particular with loop nests. We and Zdenek Dvorak have developed several ideas to work around these issues for the testcases we looked at sofar. Still there are cases where invariant loads are not hoisted out of the loops even if they are known to iterate at least once. This sometimes makes analyzing of memory accesses impossible, see Fig. 2 for an illustration of the difficulty to hoist stride loads.

```
int e2 = d.sizes[2];
int e1 = d.sizes[1];
int e0 = d.sizes[0];
int s2 = a.stride[2];
int s1 = a.stride[1];
int s0 = a.stride[0];
if (e2 > 0)
  if (e1 > 0)
    if (e0 > 0)
      {
        int k=0;
        do {
```

```
int j=0;
do {
  int i=0;
  do {
    a.m[i*s0+j*s1+k*s2] = 0.0;
    ++i;
  } while (i<e0);
  ++j;
} (while j<e1);
++k;
} (while k<e2);
}
```

Figure 1: Properly optimized loop-structure for analyzing with SCEV. All invariant memory loads have been hoisted out of the loops.

```
for (int k=0; k<d.sizes[2]; ++k)

  for (int j=0; j<d.sizes[1]; ++j)

    for (int i=0; i<d.sizes[0]; ++i)
      ... *stride;
```

```
if (d.sizes[2] > 0)
  do { k=0;
    if (d.sizes[1] > 0)
      do { j=0;
        if (d.sizes[0] > 0)
          do { i=0;
            ... *stride;
          ++i; } while (i<d.sizes[0]);
        ++j; } while (j<d.sizes[1]);
  ++k; } while (k<d.sizes[2]);
```

Figure 2: Loop header copying interaction with PRE. All loop header copies need to be hoisted out of the outermost loop for PRE to hoist the load of *stride out of the outermost loop. Compare to properly optimized loop structure in Fig. 1

Further, the current CFG profile instrumentation code does not preserve loop structure information, as it inserts new basic blocks due to instrumenting edges. Also the infrastructure for loop versioning only deals with non-nested loops. We didn't fix any of those problems yet, but restricted the transformations done to produce one loop version, which then can afford to destroy loop information.

For optimization of the versioned loop the biggest problem at the moment is the missing ability of the value range propagation pass to deal with a series of predicates of the form `A && B`. This happens if we version for example the loop in Fig. 3 for the stride condition `D.3300289_111`. VRP in this case only handles nested conditional statements, not a single conditional statement with a composed condition.

In future work also need to teach the linear loop transformation pass to consider exchanging loops not only due to data locality reasons but also considering the possibility to remove a loop nest completely due to unrolling. This is an important transformation as later measurements will show.

3 Implementation

The implementation of the loop profiling pass is divided into an analysis phase and an instrumentation or transformation phase, dependent on the mode of compilation. The analysis is done before the CFG profiling analysis and instrumentation, while the instrumentation is done after CFG profiling has cleaned up after itself. Profile counters are read and stored alongside the analysis data.

3.1 Loop analysis

During analysis we walk the loop tree and search for loop nests with a depth of at least two where each of the loop satisfies the following constraints:

- The loop has a single exit edge.
- The loop has at most one direct child.
- The number of iterations can be symbolically computed at compile time and is invariant in the whole interesting nest.

The last constraint ensures we can insert instrumentation code on the exit edge of the outer loop and that we can use the symbolic number of iterations in the transformation phase for the loop versioning condition. We rely on PRE to move the necessary defining statements to before the loop header copies.

For each such nest found, we walk the innermost loop list of basic blocks to identify and analyze memory loads and stores for their access strides using the scalar evolution of the memory address. To be interesting for profiling, those strides need to be invariant with respect to the outermost loop. From this data we compute a single boolean value that is true if all innermost loop strides are one. Building this value in a different way, or computing multiple values for other conditions is easily possible, too. A useful extension would be to check if all memory access strides for the innermost loop are the same, or to handle loads and stores separately for the checks. Another useful property to profile is alignment and data dependency, information that can be used for example by the vectorizer.

3.2 Loop instrumentation

In the instrumentation phase we insert single-value profiling counters for the conditions built during the memory access stride analysis. The outcome of the profile is then if the condition was true or false most or all of the time.

For the loop iteration counts we insert interval profiling counters with an interesting range of one to two, which gives us exact counts for once and twice iterating loops as well as the overall number of times the nest was entered.

All profiling counters are inserted on the outermost loop single exit edge, so the instrumentation is cheap and all counts are relative to the number of invocations of the whole loop nest. Instrumenting on the exit edge requires all instrumented values definition site to dominate the insertion place, which restricts the set of possibly instrumented values to loop invariant ones. The instrumentation place also requires previous optimization passes to hoist all these invariants out of the loop nest, which is possible due to canonicalization to do-while loops done by loop header copying. In Fig. 3 the instrumented code after the tree-profiling pass is shown for a selected loop from the TraMP3d-4 benchmark.

```
<bb 2>:
  D.3300088_50 = lhs->domain_m[0].domain_m[1];
  D.3300097_65 = lhs->domain_m[1].domain_m[1];
  D.3300106_80 = lhs->domain_m[2].domain_m[1];
  if (D.3300106_80 > 0) goto <L34>; else goto <L12>;
<L34>:;
  if (D.3300097_65 > 0) goto <L42>; else goto <L12>;
<L42>:;
  if (D.3300088_50 > 0) goto <L47>; else goto <L12>;
<L47>:;
  D.3300117_133 = rhs->data_m;
  D.3300119_143 = rhs->strides_m[0];
  D.3300121_150 = rhs->strides_m[1];
  D.3300124_82 = rhs->strides_m[2];
  D.3300144_177 = lhs->data_m;
  D.3300146_90 = lhs->strides_m[0];
  D.3300148_2 = lhs->strides_m[1];
  D.3300151_168 = lhs->strides_m[2];
  # i2_146 = PHI <i2_94(13), 0(5)>;
<L30>:;
  D.3300125_3 = D.3300124_82 * i2_146;
  D.3300152_155 = i2_146 * D.3300151_168;
  # i1_139 = PHI <i1_99(11), 0(6)>;
<L24>:;  D.3300283_151 = (long long int)
  D.3300122_5 = i1_139 * D.3300121_150;
  D.3300149_67 = D.3300148_2 * i1_139;
  # i0_6 = PHI <i0_194(9), 0(7)>;
<L5>:;
  D.3300120_126 = i0_6 * D.3300119_143;
  D.3300123_129 = D.3300122_5 + D.3300120_126;
  D.3300127_132 = D.3300125_3 + D.3300123_129;
  D.3300128_134 = (unsigned int) D.3300127_132;
  D.3300129_135 = D.3300128_134 * 8;
  D.3300130_136 = (double *) D.3300129_135;
  D.3300131_137 = D.3300117_133 + D.3300130_136;
  D.3300133_138 = *D.3300131_137;
  D.3300147_170 = i0_6 * D.3300146_90;
  D.3300150_173 = D.3300149_67 + D.3300147_170;
  D.3300154_176 = D.3300152_155 + D.3300150_173;
  D.3300155_178 = (unsigned int) D.3300154_176;
  D.3300156_179 = D.3300155_178 * 8;
  D.3300157_180 = (double *) D.3300156_179;
  D.3300159_181 = D.3300144_177 + D.3300157_180;
  a_188 = D.3300159_181;
  *a_188 = D.3300133_138;
  i0_194 = i0_6 + 1;
  if (D.3300088_50 > i0_194) goto <L5>; else goto <L7>;
<L7>:;
  i1_99 = i1_139 + 1;
  if (D.3300097_65 > i1_99) goto <L24>; else goto <L9>;
<L9>:;
  i2_94 = i2_146 + 1;
  if (D.3300106_80 > i2_94) goto <L30>; else goto <L55>;
<L55>:;
  D.3300280_140 = (unsigned int) D.3300106_80;
  D.3300281_114 = (long long int) D.3300280_140;
  __gcov_interval_profiler (&*.LPBX2[0], D.3300281_114, 1, 2);
  D.3300282_104 = (unsigned int) D.3300097_65;
  D.3300282_104;
  __gcov_interval_profiler (&*.LPBX2[4], D.3300283_151, 1, 2);
  D.3300284_193 = (unsigned int) D.3300088_50;
  D.3300285_187 = (long long int) D.3300284_193;
  __gcov_interval_profiler (&*.LPBX2[8], D.3300285_187, 1, 2);
  D.3300287_92 = D.3300146_90 == 1;
  D.3300288_109 = D.3300119_143 == 1;
  D.3300289_111 = D.3300287_92 && D.3300288_109;
  D.3300290_161 = (long long int) D.3300289_111;
  __gcov_one_value_profiler (&*.LPBX4[0], D.3300290_161);
<L12>:;
  return;
```

Figure 3: Instrumented loop from the TraMP3d-v4 benchmark (arc profiling code removed).

3.3 Loop transformation

The transformation phase decides whether specializations for low iteration count or constant one inner strides are worthwhile. The latter is done in case the strides proved to be one all the time, while the former is decided based upon the fraction of the times the count was low compared to the overall loop nest invocations. A value of $1/7$ has been extracted from the TraMP3d profile counts, which enables the wanted transformations in the boundary update routine. Unfortunately, if loop versioning using a condition combined from sub-conditions using logical AND, later VRP passes are not able to extract value ranges from this compound conditional, so the effect of the transformation phase is limited until this is fixed.

4 Application statistics

We have run the loop instrumentation on the TraMP3d-v4 benchmark application and collected profile data to verify if we can successfully identify worthwhile transformations. The profile collecting run invoked 10 iterations of the benchmark, which ensures that all loop nests are entered at least once. In TraMP3d-v4 there are 63 interesting loops with different kernels produced by the generic expander function templates. Out of these, for 8 loops we can successfully instrument memory access strides and 6 of these have all inner strides equal to one for all invocations. For all 63 loops we can insert counters for the number of iterations of the loops, 8 of them happen to be never executed because of benchmark flag settings.

The remaining not instrumented strides are due to our inability to load-PRE the memory accesses to the strides out of the loop nest, which causes SCEV analysis to punt on the memory accesses. Improvements in this area are subject of future work.

In the profile using compilation we schedule the periodic boundary update kernel for three specializations, one for each dimension with its iteration count being two and the innermost loop strides equal to one. The profile data in this case tells us that each of the low-iteration specialization will run 315 times, while the unversioned copy will run 2151 times. A total of 945 times, 30% of the loop nest invocations, will be spent in optimized versions of the loop.

All of the 6 loops that have inner strides equal to one for all invocations get an extra optimized loop version created.

5 Performance experiments

Performance experiments have been carried out using a small benchmark application that mimics the basic structure of an abstract C++ array operations framework like it is used in TraMP3d. The loop kernel used for the experiments is a simple assign operation applied using the following loop:

```
template <class Op>
void expand (const Array& a1, const Array& a2,
             const Domain& d, const Op& op)
{
  int ie = d.sizes[0];
  int je = d.sizes[1];
  int ke = d.sizes[2];
  for (int k=0; k<ke; ++k)
    for (int j=0; j<je; ++j)
      for (int i=0; i<ie; ++i)
        op(a1(i,j,k), a2.read(i,j,k));
}
```

with `op` assigning the second argument to the first for the benchmark and the array accesses

implemented as `storage[i*strides[0] + j*strides[1] + k*strides[2]]`.

The first experiment was to compare performance of the loops with the transformations outlined above applied to the unmodified loop. Specifically, comparing the original loop (`normal`) with the loops with inner stride optimization (`stride`), unrolling and loop exchange (`iter`) and the loops with both optimizations applied (`both`). The results are as follows.

i686	normal	stride	iter	both
full	1411	1371	1462	1471
dim0	390	333	164	159
dim1	137	125	189	98
dim2	109	101	189	97
amd64				
full	1581	1493	1606	1531
dim0	237	201	158	170
dim1	128	112	159	63
dim2	114	100	148	63
ppc				
full	2394	2380	2355	2351
dim0	785	803	141	144
dim1	317	314	141	145
dim2	284	284	142	146

The numbers are milliseconds for 100000 invocations of the loop kernel operating on 64 kB of memory for full accesses and 8 kB for the partial accesses. Thus, memory bandwidth should be that of the L1/L2 cache. The first column specifies the type of data access, `full` being a complete copy of one array to the other, while `dim0` to `dim2` copy slices of width two in dimension N (0 being the innermost loop), simulating boundary updates.

One can clearly see that for the simple loop kernel reducing the number of induction variables is a win only for register-starved machines such as the i686, while on ppc the most gain in performance is with the low iteration count loop unrolled in the innermost nest position.

With profiling enabled we are able to obtain the same performance results with training runs restricted to one of the partial accesses. We can also see that it would generate all the `both` specializations for `dim0` to `dim2` if training with the full set of tests.

Due to the limitations outlined above we were unable to produce meaningful performance numbers for the TraMP3d benchmark. Work on the FreePOOMA library has shown though, that loop kernel specialization for innermost strides being one are very important for at least register starved machines. Similar experiments with optimization for low iteration count have been unsuccessful due to the explosion in number of template function instantiations caused by a non-profile feedback directed approach. Here, the profile based optimization is the solution once the remaining issues are solved.

6 Conclusions

We have shown that with some work, the loop and profiling infrastructure of GCC can be used to perform loop specializations crucial for heavy templated C++ code. In particular, relying on the compiler to identify optimization opportunities for boundary update loops makes it possible to no longer manually specialize those routines. Compared to tackling the same problem with template specialization, the profiling approach leads to much lower compile time and text size of the resulting program.

We also have identified certain weak spots in the infrastructure of GCC. There is work ongoing to improve GCC in this regards and we hope to provide the community with a cleaned

up infrastructure for profiling and loop optimization in the 4.3 time frame.

Ultimately the goal should be to generally make CFG manipulation loop aware and manage loop structure as a required or provided pass property. This should also allow to propagate information gathered from the profiles to later passes such as the loop unroller and the vectorizer.

References

[1] David Edelsohn Daniel Berlin and Sebastian Pop. High-level loop optimizations for gcc. In *The 2004 GCC Developers' Summit Proceedings*, June 2004.

[2] Richard Günther. *Three-dimensional Parallel Hydrodynamics and Astrophysical Applications*. PhD thesis, Eberhard-Karls-Universität Tübingen, 2005.

[3] Jan Hubicka. Profile driven optimizations in gcc. In *GCC Developers' Summit Proceedings*, June 2003.

[4] John V. W. Reynders, Paul J. Hinker, Julian C. Cummings, Susan R. Atlas, Subhankar Banerjee, William F. Humphrey, Steve R. Karmesin, Katarzyna Keahey, Marikani Srikant, and Mary Dell Tholburn. POOMA: A Framework for Scientific Simulations of Paralllel Architectures. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming in C++*, pages 547–588. MIT Press, 1996.

[5] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

Multi-Language Programming

The Challenge and Promise of Class-Level Interfacing

Cyrille Comar, Matthew Gingell, Olivier Hainque, and Javier Miranda
AdaCore
{comar, gingell, hainque, miranda}@adacore.com

Abstract

Many computer applications today involve modules written in different programming languages, and integrating these modules together is a delicate operation. This first requires the availability of formalisms to let programmers denote "foreign" entities like objects and subprograms as well as their associated types. Then, proper translation of what programmers express often calls for significant implementation effort, possibly down to the specification of very precise ABIs (Application Binary Interfaces). Meta-language based approaches a-la CORBA/IDL are very powerful in this respect but typically aim at addressing distributed systems issues as well, hence entail support infrastructure that not every target environment needs or can afford. When component distribution over a network is not a concern, straight interfacing at the binary object level is much more efficient. It however relies on numerous low level details and in practice is most often only possible for a limited set of constructs.

Binary level interaction between foreign modules is traditionally achieved through subprogram calls, exchanging simple data types and relying on the target environment's core ABI. Object Oriented features in modern languages motivate specific additional capabilities in this area, such as class-level interfacing to allow reuse and extension of class hierarchies across languages with minimal constraints. This paper describes work we have conducted in this context, allowing direct binding of Ada extensible tagged types with C++ classes. Motivated by extensions to the Ada typing system made as part of the very recent language standard revision, this work leverages the GCC multi-language infrastructure and implementation of the Itanium C++ ABI. We will first survey the issues and mechanisms related to basic inter-language operations, then present the interfacing challenges posed by modern object oriented features after a brief overview of the Ada, C++, and Java object models. We will continue with a description of our work on Ada/C++ class-level interfacing facilities, illustrated by an example.

1 Interfacing Across Programming Languages - Introduction

Two general aspects of Multi-Language Programming are the formalisms available to denote and use "foreign" entities exposed from a different language than the one in which they are referred to, and the support infrastructure for what programmers express. "Interfacing" can cover many different things, such as access to foreign data, foreign type representa-

tion, calls to foreign subprograms, handling of foreign events like exceptions, and reuse of object class hierarchies. In any case, an interface always implies agreement between the involved parties. For instance, a subroutine call will only operate properly if the caller and the callee agree on how arguments are passed (in what order, using what machine resources), who allocates/releases this or that part of the stack, how aligned the stack pointer is expected to be, etc. Likewise, operating on a foreign variable requires a way to describe or denote the variable's "native" type to ensure a correct interpretation of the actual value layout. Typically, more powerful formalisms make programmers lives easier at the price of more complicated underlying infrastructure.

1.1 Core Mechanisms - Basic Capabilities

A first set of basic interfacing possibilities is provided by explicit programming language features associated with well established *calling conventions* and low level rules for the target environment specified in base *Application Binary Interface* (ABI) documents.

Among other things, base ABI documents describe binary files formats, basic data type layouts, stack frame organization, and machine level conventions for passing parameters to and returning results from subprograms. See [19] and [12] for examples of such documents for the i386 and amd64 architectures. Additional calling conventions may apply in some environments, such as the `stdcall`/`fastcall` variants on x86-Windows [17], or for some specific programming languages as illustrated by the differences between Pascal and C in arguments passing order. These conventions provide a common ground for basic inter-language interfacing capabilities and binary code interoperability, ensuring for instance proper interaction between GCC compiled code and target operating system libraries.

On top of the common base conventions we have just surveyed, various standard devices are available on the programming languages side. As a first example, the Ada Reference Manual (ARM) includes a full annex dedicated to the issue [22, Annex B], covering interfacing with C, Cobol, and Fortran, and allowing implementations to support other languages. The minimum support specified in this annex consists of standard packages for each language, for instance the `Interfaces.C` hierarchy for C, and specific compiler *pragmas*:

- *Pragma Import*, to import an entity defined in a foreign language into an Ada program, thus allowing a foreign-language subprogram to be called from Ada, or a foreign-language variable to be accessed from Ada.
- *Pragma Export*, to export an Ada entity to a foreign language.
- *Pragma Convention*, to specify that an Ada entity should use the conventions of another language for passing parameters to subprograms, or else to represent a data type in memory (for example determining matrix element ordering).
- *Pragma Linker_Options*, to specify the system linker parameters needed when a given compilation unit is included in a program.

The following code example illustrates the use of some of these facilities to call a C function from Ada to print out an `int` value found at a provided address. It uses the standard `Interfaces.C` package to get access to the Ada type corresponding to `int`, declares an Ada subprogram to represent the C service interface, and imports the service by way of an

`Import` pragma. The latter tells the compiler that the subprogram is external with C convention and states what symbol (link name) should be used to refer to it.

```
with Interfaces.C; use Interfaces;
procedure Binding_Example is

   -- Map and use C function
   -- void dump_int_at (int *ptr);

   procedure Dump_Int_At (Ptr : access C.Int);

   pragma Import
    (Convention => C,
     Entity     => Dump_Int_At,
     Link_Name  => "dump_int_at");

   Myint : aliased C.Int := 12;
begin
   Dump_Int_At (Myint'Access);
end;
```

The Ada `Access` *attribute* used here in `Myint'Access` corresponds to the `&` unary addressing operator in C: It produces an address, called an *access value*, said to designate the entity. Ada access values are normally subject to *accessibility checks* mandated by the language to prevent the creation of dangling pointers [22, 3.10.2-24]. Roughly, an access value may only be assigned to an object of an access type if the value lifetime is guaranteed to be shorter than the lifetime of the target type. Performing these checks requires run-time code in some cases, raising the predefined *Program_Error* exception in case of failure. With GNAT, accessibility checks result in automatic extra argument passing in calls to subprograms with access parameters. A noticeable effect of the C convention applied to `Dump_Int_At` in our example is to disable this circuitry, as the extra parameter is not part of the base interface and only makes sense for Ada subprograms.

As other examples, C++ provides *linkage specifications* such as `extern "C"` to allow the use of C++ entities in other languages, and calls to foreign routines from Java are possible thanks to an exhaustive Java Native Interface specification [11].

1.2 Higher Level Facilities and Paper Overview

As time goes by, programming languages evolve, higher level features are introduced, implementation choices are made, and binary compatibility issues, especially with respect to other languages, are not always part of the picture upfront. This is legitimate, as a concept in one language doesn't necessarily have a counterpart in others, and because complex factors come into play views inevitably vary on what scheme is best in each specific case. Still, commonalities do occur even for sophisticated features, and the capability to interface across languages at these higher levels is often desirable and an interesting challenge. For instance, an Ada top-level subprogram might be interested in catching exceptions raised by C++ subcomponents, or vice-versa. Although the concept of "exception" is similar in both languages, there are variations in the way it is precisely mapped on each side, and determining the appropriate semantics for such a facility is difficult to start with.

This paper describes work we have conducted in this context, on GNU Ada/C++ "class-level interfacing," to allow direct binding of Ada extensible tagged types hierarchies to C++ classes in both directions. Motivated by extensions to the Ada typing system made as part of the very recent language standard revision [1], this work leverages the GCC multi-language infrastructure and implementation of the Itanium C++ ABI [5] to simplify interfacing between OO languages at the class-level.

In Section 2 we briefly describe the Ada OO model and its relationship with the C++ and Java models. In Section 3 we present in greater detail what "class-level interfacing" involves and the various possible approaches. In Section 4 we analyze the GNAT specific capabilities for interfacing Ada with C++, illustrated

with a commented example in Section 5, and then offer our conclusions.

2 Static OO Models Comparison: Ada, Java, C++

Booch [2] defines Object Orientation around seven principles: Abstraction, Encapsulation, Modularity, Hierarchy, Typing, Concurrency and Persistence. The first two principles are about separating how objects are defined and used from how they are represented and implemented. Modularity is about organizing programs as a collection of separate components with defined interactions and limited access to data. Typing and Hierarchy are about distinguishing different kinds of objects and structuring them according to their common characteristics. A complete description of these principles can also be found in [9, Section 1.3.2].

In C++ and Java, the notion of "class" is central to all these principles even though modularity is also achieved through name-spaces and separate files. In those languages, classes allow grouping of data members along with their associated function members (methods). They also specify their position in a hierarchy by specifying their immediate parents and offer visibility restriction mechanisms for their members.

In Ada, the first three concepts (Abstraction, Encapsulation, and Modularity) are associated with packages and "private" declarations while the Typing concept is clearly associated with the Ada typing model. The Hierarchy principle is found both in packages and types: the child package construct allows the programmer to define a hierarchy of packages, and the type derivation allows him to create hierarchies of types.

Ada 83, Ada's original definition, was considered an Object-Based language. It was based on the above principles without offering any mechanism for dynamic polymorphism. In fact, dynamic dispatching was deliberately banned from the language since it was, at the time, considered incompatible with its safety requirements. In this first model, a class is represented by a private type along with its primitive operations (methods) encapsulated in a package. The implementation of the private type is typically a record grouping all data members, and the implementation of methods are hidden in the package body.

The second revision of the language, known as Ada 95, enriches its typing system with a new variety of record called "tagged" records. The main characteristic of these records is that they can be extended during derivation and thus are used as the basis for dynamic polymorphism under a single inheritance model. Both C++ and Java fully support single inheritance. Contrary to those languages where polymorphism is implicit, Ada distinguishes it through an explicit notation: T'Class is the polymorphic, called *class-wide*, version of a specific tagged type T, which means that the actual run-time type of an object declared of type T'Class can be T or any of its descendant.

In Java, all methods are dispatching. In C++, methods are dispatching when they are declared "virtual." In Ada, all methods are potentially dispatching and a call dispatches or not depending on the nature of the object it applies to. Dispatching will only occur when the latter has a polymorphic type, as illustrated by the code excerpt below:

```
-- A call is dispatching if the controlling
-- argument type is classwide:

X : T'Class := ...;
Y : T       := ...;
...
X.T_Method;      -- dispatching
Y.T_Method;      -- not dispatching
```

C++ offers full-scale multiple inheritance. That is to say, a class may have several parents and inherits all their data and function members. This is a powerful capability providing a great deal of expressive power. At the design level, it is particularly convenient for composing concepts represented by independent classes. Programming with full multiple inheritance requires familiarity with the answers provided by the language to tricky questions such as: What happens when a class inherits multiple times from the same ancestor through different derivation paths? What happens when inheriting methods with the same profile from different parents? A thorough overview of how C++ answers such questions is available from [21], along with many ideas on how multiple inheritance can be implemented efficiently. Nonetheless, although multiple inheritance has proven to be a very powerful paradigm for skilled programmers, its extensive use may have negative consequences for the readability and long term maintainability of software.

In recent years, a number of language designs [6, 7] have adopted a compromise between full multiple inheritance and strict single inheritance, which is to allow multiple inheritance of *specifications*, and only single inheritance of *implementations*. Typically this is obtained by means of *"interface"* types. An interface consists solely of a set of operation specifications: it has no data components and no operation implementations. A type may implement multiple interfaces, but can inherit code from only one parent type. This model has much of the power of full-scale multiple inheritance, but without most of the implementation and semantic difficulties of the C++ multiple inheritance model [10].

Ada 2005 provides support for such abstract interface types [1, Section 3.9.4]. Its characteristics are introduced by means of an interface type declaration and a set of subprogram declarations. The interface type has no data components and its primitive operations are either abstract or null, in which case they behave as if their body was empty. A data type that implements an interface must provide non-abstract versions of all the abstract operations of its parents. Here is a code sample to illustrate the declaration of interface types and the associated multiple inheritance capability in Ada 2005:

```
package Interfaces_Example is
  type I1 is interface;
  function P (X : I1) return Integer
    is abstract;

  type I2 is interface and I1;
  procedure Q (X : I1) is null;
  procedure R (X : I2) is abstract;

  type Root is tagged record with private;
  procedure A (Obj : T);
  function B (Obj : T) return Integer;

  type DT is
    new Root and I1 and I2 with private;
  -- DT1 must implement P, and R
  ...

  type DT2 is new DT with private;
  -- Inherits all the primitives and
  -- interfaces of the ancestor

private
  type Root is tagged record with
    -- Root components
    ...
  end record;

  type DT is
    new Root and I1 and I2 with record
    -- DT components
    ...
  end record;

  type DT2 is new DT with record
    -- DT2 components
    ...
  end record;
end Interfaces_Example;
```

The interface I1 has one subprogram, *P*. The interface I2 has the same operations as I1 plus two subprograms: the null subprogram *Q* and the abstract subprogram *R*. Then, we define the root of a derivation class that has two primitive operations, *A* and *B*. *DT* extends the root type and also inherits the two interfaces I1 and I2, so it is required to implement all the associated abstract subprograms. Finally, type *DT2* extends

DT1, inheriting all the primitive operations and interfaces of its ancestor.

OO languages that provide abstract interface types [6, 7] have a run-time mechanism that determines whether a given object implements a particular interface. Accordingly Ada 2005 extends the membership operation to interfaces and allows the programmer to write the predicate *O in I'Class*. Let us consider an example that uses the types declared in the previous fragment and displays both of these features:

```
procedure Dispatch_Call
  (Obj : I1'Class) is
begin
  -- 1: dispatch call
  ... := P (Obj);

  --    2: membership test
  if Obj in I2'Class then

    -- 3: interface conversion plus
    --      dispatch call
    R (I2'Class (Obj));
  end if;

  -- 4: dispatch to predefined op.
  I1'Write (Stream, Obj)
end Dispatch_Call;
```

The type of the formal *Obj* covers all the types that implement the interface I1. At –1– we dispatch a call to the primitive *P* of I1. At –2– we use the membership test to check if the actual object also implements I2. In order to issue a dispatching call to the subprogram R of interface I2, at –3– we perform a conversion of the actual to the class-wide type of interface I2. If the object does not implement the target interface and we do not protect the interface conversion with the membership test, then the predefined exception *Constraint_Error* is raised at run-time. Finally at –4– we see that, in addition to user-defined primitives, we can also dispatch calls to predefined Ada operations: *'Size*, *'Alignment*, *'Read*, *'Write*, *'Input*, *'Output*, *Adjust*, *Finalize*, or the equality operator.

Ada 2005 also extends abstract interfaces for its use in concurrency, but this topic is not discussed in this paper. For details on the GNAT implementation of synchronized interfaces see [15].

3 Interfacing at the class level

3.1 Basic Requirements

In general, reusing an object-oriented system requires two distinct capabilities: creating instances of existing classes and defining new classes inheriting from them. Reusing an OO system written in a different language requires the additional capability: to "see" foreign classes and use them with as few restrictions as possible. In particular it implies the possibility of defining in one language an instance of a class which has been implemented in another. Another interesting capability is inheriting from foreign classes, which implies that dynamic binding can cross language boundaries transparently. Although of less general interest, Run Time Type Information (RTTI) queries such as membership tests are also worth mentioning.

For such interfacing capabilities to make sense, minimal commonalities between the OO models are required to preserve coherence between a class hierarchy defined on one side and used on the other.

3.2 Common Approaches

A well-known approach to inter-language class-level interfacing consists of resorting to a common meta language. CORBA [18] offers an interesting case of the definition of such a model. CORBA's main goal is to support the development of Object-Oriented distributed systems. Thus inter-computer communication

plays an important role. If we abstract the communication component however, CORBA offers a model for interfacing systems that may be written in different languages and thus offers a language independent object model. This model is described using an Interface Definition Language (IDL). The CORBA IDL defines the concepts needed to describe the most common abstractions: basic and composite data types, modules, exceptions, and class hierarchies, possibly with multiple inheritance. Not being an implementation language, only the definition part of a class needs to be provided in the CORBA IDL and corresponds to a Java interface. In fact they are also called interfaces in IDL jargon. Hence, CORBA IDL seems an ideal solution for interfacing at the class level since it offers the common ground on which languages with different object models can usefully communicate.

At the practical level, however, the situation is not ideal. Within the CORBA framework, each language requires a binding between its native OO model and the Definition Language, an IDL compiler is needed to transform IDL models into a set of native specifications or header files, and these then have to be connected to the existing system. So, not only does the user need to learn and use a yet another language, the final system ends up with a thick layer for the interfacing part composed of the two bindings mentioned above connected by a complete communication middleware (Object Request Broker, ORB). In situations where the various subsystems are not intended to be deployed on different machines, this can represent a very significant overhead both in development effort and in the amount of code dedicated to interfacing.

The use of an Interface Definition Language is not limited to CORBA. It is also used in other contexts where interfacing at class level is sought. [4] offers a good description of such a case for interfacing two languages with quite different OO models: OCaml and C#. The IDL used in this context is very close to Java syntax and the paper gives a good description of the notion of shadow (or Proxy) classes, another typical model for class level interfacing between two incompatible worlds.

The "shadow/proxy class" idea is to define two matching class hierarchies on each side of the language fence. For each class implemented on one side, a shadow class is defined on the other side where all its methods are wrappers that ultimately call the corresponding foreign method. On the shadow side, each class instance needs to be associated to a real instance on the other side, which can be done as part of the initialization of the shadow instance.

The SWIG system [20] is worth mentioning in this context. SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages such as Java, Python, Ruby or Scheme, most of which offer their own OO model. As with CORBA, SWIG uses an IDL. Its syntax is very close to C/C++ header files, so interface files can be written quickly by simplifying the existing header files of the system to interface. SWIG automatically creates the hierarchy of shadow classes that will allow those various OO languages to access pre-existing C++ class hierarchies.

The shadow class mechanism becomes complicated when the original language features garbage collection, since the shadow object may end up being the only valid reference to the real object and is usually hidden from the original environment. When the language does not provide garbage collection, the opposite problem can arise: how to make sure that those shadow objects or their counterpart are released properly before becoming unreachable? All these issues are described in great detail in the SWIG documentation.

Apart from the aforementioned families of approaches, direct interfacing at the binary level can sometimes be achieved, alleviating the need for intermediate software layers. This is what we have done for Ada/C++ interfacing with the GNAT compiler, as described in the following section.

4 The GNAT Approach to Ada/C++ Interfacing

The interfacing mechanisms mentioned in the previous section have been designed to be independent of compiler technologies. They generate potentially heavy glue code whose only requirements are related to the semantics of the languages to interface and not to their actual implementation.

As compiler implementors with full control over code generation on one side of the interface, our perspective is different. Our purpose is to provide a low-level mechanism that simplifies interfacing and allow production of lighter glue code when possible. For instance, when an object is part of an Ada/C++ interface, a heavy duty interfacing mechanism such as CORBA requires the following steps: 1) marshall the object to transform it from its Ada representation to a machine independent representation such as CDR in the CORBA case; 2) send this encoded data through the communication channel (ORB for CORBA); and 3) unmarshall the data into its C++ representation.

From the compiler viewpoint, a much simpler method can be used if one side can mimic the data representation expected by the other. In such a situation interfacing becomes as simple as sharing a name or a reference. In this context, our goal is to extend the base Ada interfacing pragmas introduced in Section 1.1 to encompass the class concept and its associated mechanisms, such as dynamic dispatching. This is possible thanks to the commonalities between the Ada and the C++ object models: a C++ class maps naturally to an Ada tagged type, a class data member is a tagged record component, a virtual function member maps to an Ada primitive operation, and static members functions or constructors can be mapped to Ada operations on the classwide type. The following subsections describe two different schemes we have developed to achieve this goal.

4.1 Original Scheme for Ada95

When the original Ada95/C++ interfacing mechanism was designed in the mid 90s, a study of various C++ compilers showed wide variation in the layout of C++ objects and their virtual function tables. As a consequence, we decided to provide a model of interfacing to C++ which depended as little as possible on the choices made by particular C++ implementations. In this approach, the GNAT compiler made no assumptions about how objects generated by the C++ compiler were laid out, and required that the user determine and provide a correct matching representation in Ada themselves.

For instance, the compiler made no assumptions about where a virtual function table pointer would appear in an imported object. Hence, in the declaration of the corresponding type in Ada the user had to provide a dummy pointer field and mark it explicitly with a `pragma CPP_Vtable`. Additionally the compiler had no special knowledge of how a virtual function table was actually laid out, leaving it up to the user to determine whether or not he needed to provide specific offsets in his method bindings via `pragma CPP_Virtual`.

In addition, no knowledge about what might be needed to call a C++ method was encapsulated in the compiler itself. Instead, the compiler delegated the responsibility for accessing the vtable and calling methods through it to a set of routines in the run-time with a well defined procedural interface. This abstraction meant it was possible to adapt GNAT to changes in C++ compilers or to adapt it to new compilers very easily at the run-time level without actually having to make any changes in the compiler itself.

On the one hand, this approach enabled a sufficiently motivated user to find a way of interfacing to C++ objects generated by a wide variety of compilers. For instance, users interested enough in finding the virtual function pointer in objects generated by the Sun C++ compiler and determining at what offsets it had placed what methods could, with enough effort, put together a useful Ada binding.

On the other hand, this process was labor intensive and error prone, and required a level of knowledge about the implementation of both compilers that the user may not have had and was unlikely to be interested in acquiring. While in principle the facilities the user required were provided, in practice there was a great deal left to be desired.

4.2 Redesign for Ada 2005 - Leveraging the C++ ABI

An alternate approach recently added to the GNAT compiler takes advantage of knowledge of the C++ ABI [5]. This approach takes responsibility for the details and complexities which the previous approach left to the end user. This ABI is also followed by GCJ, the GNU Java compiler [8, Section 12.1]. For each tagged type the compiler generates a primary dispatch table associated with its single-inheritance line of derivation and a secondary dispatch table for each abstract interface type inherited by the tagged type. This model incurs storage costs, in the form of additional pointers to dispatch tables in each object and *thunks* that adjust the value of the pointer to the object implementing abstract interface types.

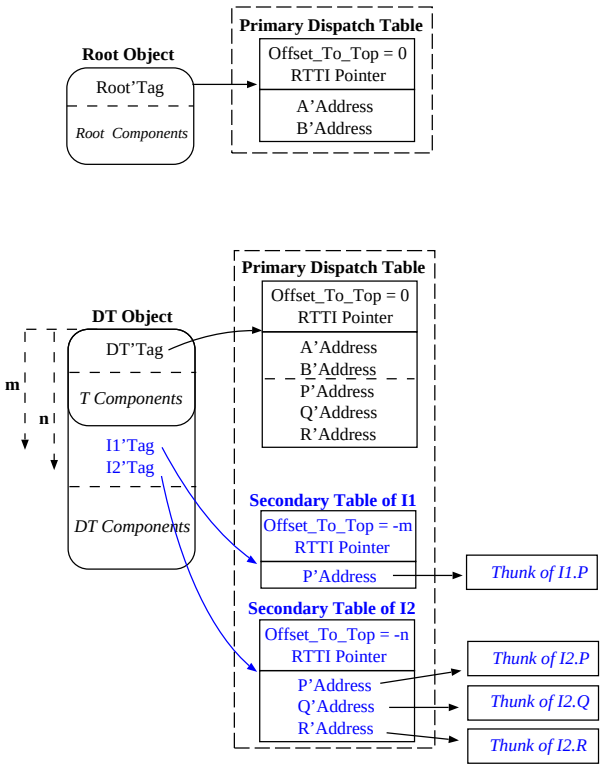


Figure 1: Layout compatibility with C++

Following with the example presented in section 2, Figure 1 represents the layout of the tagged types *Root* and *DT*. The dispatch table has a header containing the offset to the top and the Run Time Type Information Pointer (RTTI). For a primary dispatch table, the first field is always set to 0. The tag of the object points to the first element of the table of pointers to primitive operations. At the bottom of the same figure we have the layout of DT, type derived from Root that implements two interfaces (I1 and I2). The layout of the object (left side of the figure), shows that the derived object contains all the components of its parent type plus 1) the tag of all the implemented interfaces, and

2) its own user-defined components. Concerning the contents of the dispatch tables, the primary dispatch table is an extension of the primary dispatch table of its immediate ancestor, and thus contains direct pointers to all the primitive subprograms of the derived type. The *offset_to_top* component of the secondary tables holds the displacement to the top of the object from the object component containing the interface tag. The offset-to-top values of interfaces I1 and I2 are *m* and *n* respectively. This offset provides a way to find the top of the object from any derived object that contains secondary dispatch tables and is necessary in type conversions. In addition, rather than containing direct pointers to the primitive operations associated with the interfaces, the secondary dispatch tables contain pointers to small fragments of code called *thunks*. These thunks are generated by the compiler, and used to adjust the pointer to the base of the object.

The main difference between the current ABI layout provided by the Ada compiler and the official C++ ABI [5] is the contents of the RTTI pointer. On the Ada side this pointer references a record containing information required to support Ada semantics (accessibility level, expanded name of the tagged type, etc.) plus two additional tables: a table containing the tag of all the immediate ancestors of the type, and a table containing the tag of all the abstract interface types implemented by the type plus its corresponding offset-to-top values in the object layout. These tables give run-time support to the membership test and interface conversions respectively. Figure 2 completes the run-time data structure described in previous section with the GNAT *Type Specific Data* record.

It is clear that this difference introduces several incompatibilities. For example, on the Ada side we cannot make use of the membership test on a class imported from the C++ side, and similarly on the C++ side the dynamic cast operator cannot be used with tagged types imported from the Ada side. We are working on this area to reduce these layout differences.

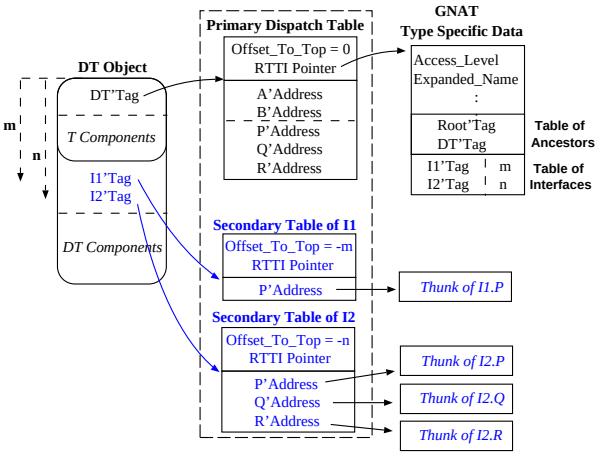


Figure 2: GNAT Layout

Regarding the C++ ABI's [5] completeness for use in the implementation of other OO languages, we have found that the case of variable sized tagged objects is not supported. Complications arise when a tagged type has a parent that includes some component whose size is determined by a discriminant and the type is also derived from abstract interface types. For example:

```
type Root (D : Positive) is tagged record
  Name : String (1 .. D);
end record;

type DT is new Root and I1 and I2 with ...
Obj : DT (N);
-- N is not necessarily static
```

In this example it is clear that the final position of the components containing the tags associated with the secondary dispatch tables of *DT* depends on the actual value of the discriminant at the point the object Obj is elaborated. Therefore the offset-to-top values can not be placed in the header of the secondary dispatch tables because these tables are shared by all the objects of the type. The C++ ABI does not address this problem for the simple reason

that C++ classes do not have non-static components.

In order to solve this problem we decided to store the offset-to-top values immediately following each of the interface tags of the object (that is, adjacent to each of the object's secondary dispatch table pointers). In this way, this offset can be retrieved when we need to adjust a pointer to the base of the object. There are two basic cases where this value needs to be obtained: 1) The thunks associated with a secondary dispatch table for such a type must fetch this offset value and adjust the pointer to the object appropriately before dispatching a call; 2) Class-wide interface type conversions need to adjust the value of the pointer to reference the secondary dispatch table associated with the target type. In this second case this field allows us to reach the object's base address, but we also need this value in the table of interfaces to be able to displace down the pointer to reference the field associated with the target interface. For this purpose the compiler generates object specific functions which read the value of the offset-to-top hidden field, and stores pointers to these functions in the table of interfaces. For further information see [16].

5 A Commented Example

In this section we present the new GNAT features for interfacing with C++ by means of an example. This example consists of a classification of animals; classes have been used to model our main classification of animals, and interfaces provide support for the management of secondary classifications. We will first present a case in which the types and constructors are defined on the C++ side and imported from the Ada side, and latter the reverse case.

5.1 Importing from C++

The root of our derivation will be the *Animal* class, with a single private attribute (the *Age* of the animal) and two public primitives to set and get the value of this attribute.

```
class Animal {
 public:
   virtual void Set_Age (int New_Age);
   virtual int Age ();
 private:
   int Age_Count;
};
```

Abstract interface types are defined in C++ by means of classes with pure virtual functions and no data members. In our example we will use two interfaces that provide support for the common management of *Carnivore* and *Domestic* animals:

```
class Carnivore {
public:
   virtual int Number_Of_Teeth () = 0;
};

class Domestic {
public:
   virtual void Set_Owner (char* Name) = 0;
};
```

Using these declarations, we can now say that a *Dog* is an animal that is both Carnivore and Domestic, that is:

```
class Dog : Animal, Carnivore, Domestic {
 public:
   virtual int  Number_Of_Teeth ();
   virtual void Set_Owner (char* Name);

   Dog(); // Constructor
 private:
   int  Tooth_Count;
   char *Owner;
};
```

In the following examples we will assume that the previous declarations are located in a file named *animals.h*. The following package demonstrates how to import these C++ declarations from the Ada side:

```
with Interfaces.C.Strings;
use  Interfaces.C.Strings;
package Animals is
```

```
   type Carnivore is interface;
   function Number_Of_Teeth (X : Carnivore)
     return Integer is abstract;
   pragma Convention (CPP, Number_Of_Teeth);

   type Domestic is interface;
   procedure Set_Owner
     (X     : in out Domestic;
      Name : Chars_Ptr) is abstract;
   pragma Convention (CPP, Set_Owner);

   type Animal is tagged private;
   pragma CPP_Class (Animal);

   procedure Set_Age
     (X : in out Animal; Age : Integer);
   pragma Import (CPP, Set_Age);

   function Age (X : Animal) return Integer;
   pragma Import (CPP, Age);

   type Dog is new Animal
     and Carnivore and Domestic with private;
   pragma CPP_Class (Dog);

   function Number_Of_Teeth (A : Dog)
     return Integer;
   pragma Import (CPP, Number_Of_Teeth);

   procedure Set_Owner
     (A : in out Dog; Name : Chars_Ptr);
   pragma Import (CPP, Set_Owner);

   function New_Dog return Dog'Class;
   pragma CPP_Constructor (New_Dog);
   pragma Import (CPP, New_Dog, "_ZN3DogC2Ev");
private
   type Animal is tagged record
     Age : Integer := 0;
   end record;

   type Dog is new Animal
     and Carnivore and Domestic with
   record
     Tooth_Count : Integer;
     Owner       : Chars_Ptr;
   end record;
end Animals;
```

Thanks to the compatibility between GNAT run-time structures and the C++ ABI, interfacing with these C++ classes is easy. The only requirement is that all the primitives and components must be declared exactly in the same order in the two languages. The code makes no use of the GNAT-specific pragmas `CPP_Vtable` and `CPP_Virtual` described in Section 4.1.

Regarding the abstract interfaces, we must indicate to the GNAT compiler by means of a `pragma Convention (CPP)`, the convention used to pass the arguments to the called primitives will be the same as for C++. For the imported classes we use `pragma CPP_Class` to indicate that they have been defined on the C++ side; this is required because the dispatch table associated with these tagged types will be built on the C++ side and therefore will not contain the predefined Ada primitives which Ada would otherwise expect.

Finally, for each user-defined primitive operation we must indicate by means of a `pragma Import (CPP)` that they are imported from the C++ side.

As the reader can see there is no need to indicate the C++ mangled names associated with each subprogram because it is assumed that all the calls to these primitives will be dispatching calls. The only exception is the constructor, which must be registered in the compiler by means of `pragma CPP_Constructor` and needs to provide its associated C++ mangled name because the Ada compiler generates direct calls to it. In order to further simplify interfacing with C++, we are currently working on a utility for GNAT that automatically generates the proper mangled names for C++ imported subprograms, as generated by the G++ compiler.

With the above packages we can now declare objects of type Dog on the Ada side and dispatch calls to the corresponding subprograms on the C++ side. We can also extend the tagged type Dog with further fields and primitives, and override some of its C++ primitives on the Ada side. For example, here we have a type derivation defined on the Ada side that inherits all the dispatching primitives of the ancestor from the C++ side.

```
with Animals; use Animals;
package Vaccinated_Animals is
   type Vaccinated_Dog is
      new Dog with null record;
   function Vaccination_Expired
```

```
      (A : Vaccinated_Dog) return Boolean;
   pragma Convention
      (CPP, Vaccination_Expired);
end Vaccinated_Animals;
```

It is important to note that, because of the ABI compatibility, the programmer does not need to add any further information to indicate either the object layout or the dispatch table entry associated with each dispatching operation.

5.2 Exporting to C++

Now let us define all the types and constructors on the Ada side and export them to C++, using the same hierarchy of our previous example:

```
with Interfaces.C.Strings;
use Interfaces.C.Strings;
package Animals is
   type Carnivore is interface;
   function Number_Of_Teeth (X : Carnivore)
     return Integer is abstract;
   pragma Convention (CPP, Number_Of_Teeth);

   type Domestic is interface;
   procedure Set_Owner
     (X    : in out Domestic;
      Name : Chars_Ptr) is abstract;
   pragma Convention (CPP, Set_Owner);

   type Animal is tagged private;
   pragma Convention (CPP, Animal);

   procedure Set_Age
     (X    : in out Animal;
      Age : Integer);
   pragma Export (CPP, Set_Age);

   function Age (X : Animal) return Integer;
   pragma Export (CPP, Age);

   type Dog is new Animal
     and Carnivore
     and Domestic with private;
   pragma Convention (CPP, Dog);

   function Number_Of_Teeth (A : Dog)
     return Integer;
   pragma Export (CPP, Number_Of_Teeth);

   procedure Set_Owner
     (A    : in out Dog;
      Name : Chars_Ptr);
   pragma Export (CPP, Set_Owner);

   function New_Dog return access Dog'Class;
   pragma Export (CPP, New_Dog);

private
   type Animal is tagged record
     Age : Integer := 0;
   end record;

   type Dog is new Animal
     and Carnivore and Domestic with
   record
     Tooth_Count : Integer;
     Owner       : Chars_Ptr;
   end record;
end Animals;
```

Compared with our previous example the only difference is the use of pragma *Export* to indicate to the GNAT compiler that the primitives will be available to C++. Thanks to the ABI compatibility, on the C++ side there is nothing else to be done; as explained above, the only requirement is that all the primitives and components are declared in exactly the same order. For completeness, let us see a brief C++ main program that uses the declarations available in *animals.h* (presented in our first example) to import and use the declarations from the Ada side, properly initializing and finalizing the Ada run-time system along the way:

```
#include "animals.h"
#include <iostream>
using namespace std;

void Check_Carnivore (Carnivore *obj) { ... }
void Check_Domestic (Domestic *obj)   { ... }
void Check_Animal (Animal *obj)       { ... }
void Check_Dog (Dog *obj)             { ... }

extern "C" {
 void adainit (void);
 void adafinal (void);
 Dog* new_dog ();
}

void test () {
 Dog *obj = new_dog();   // Ada constructor
 Check_Carnivore (obj); // Check secondary DT
 Check_Domestic (obj);  // Check secondary DT
 Check_Animal (obj);    // Check primary DT
 Check_Dog (obj);       // Check primary DT
}

int main () {
 adainit ();  test();  adafinal ();
 return 0;
}
```

6 Conclusion

The C++ ABI [5] was first defined as part of a new processor ABI, but it has evolved into a processor independent ABI for C++ which can be used as a de-facto standard for other languages (ie. currently the GNU C++, Ada and Java compilers support this ABI). This evolution not only allows mixing C++ objects compiled with different compilers in the same executable, but also allows multi-language object-oriented programs compiled into a single executable. The common ABI allows the programmer to mix objects from different languages and also permits him the use of features such as dynamic dispatching, which are not limited by language boundaries.

It is well known that several modern static Object-Oriented languages offer similar support for single inheritance and multiple inheritance of abstract interface types. However, the current C++ ABI does not completely fulfill all the requirements of these languages. For example, in this paper we have shown that this ABI should be extended for languages with variable sized objects like Ada. We think that it would be desirable to extend this ABI with new sections covering the basic data structures supporting Object Oriented features, such as dynamic dispatching, in a language independent way to give GCC full support to multi-language programming at the class level. This would improve interfacing capabilities between the OO languages supported by GCC and would open new opportunities for software reuse in a world where programming language trends evolve rapidly.

For this work to be of direct use to the GCC users interested in reusing libraries written in several languages (ie. Ada, C++, Java), a tool for automating the generation of the interface files would be hightly desirable. SWIG [20] seems to offer a very promising framework for developing such a tool since it provides all the technology for generating shadow class hierarchies. In this context, languages with ABI compatibility have an important benefit: shadow methods would not be wrappers anymore but "direct" views of the real methods because the need for shadow objects can be replaced by direct views of the real object, thus improving the efficiency of the code and eliminating all the complexity related to memory management.

References

[1] Ada Rapporteur Group. *Annotated Ada Reference Manual with Technical Corrigendum 1 and Amendment 1 (Draft 16): Language Standard and Libraries*. (Working Document on Ada 2005). Ada-Europe, 2006.

[2] G. Booch *Object-Oriented Analysis and Design* Addison-Wesley, 2nd edition,1993. ISBN: 0805353402

[3] J. Byous. *Java Technology: The Early Years*, 2006. `http://java.sun.com/features/1998/05/birthday.html`.

[4] E. Chailloux, G. Grégoire, R. Montelatici *Mixing the Objective Caml and C+ Programming Models in the .NET Framework* The 3rd International Conference on .NET Technologies, Plenz, Czech Republic May 30-June 1, 2005

[5] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. *Itanium C++ Application Binary Interface (ABI)*, Revision 1.86, 2005. `http://www.codesourcery.com/cxx-abi`

[6] E. International. *C# Language Specification (2nd edition)*. Standard

ECMA-334. Standardizing Information and Communication Systems, December, 2002.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (3rd edition)*. Addison-Wesley, 2005. ISBN: 0-321-24678-0.

[8] *Guide to GNU GCJ*, 2005. `http://gcc.gnu.org/onlinedocs/gcj/`

[9] *Handbook for Object-Oriented Technology in Aviation* `http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot/`

[10] ISO/IEC. *Programming Languages: C++ (1st edition)*. ISO/IEC 14882: 1998(E). 1998.

[11] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. ISBN: 0-201-32577-2, Addison-Wesley Professional; 1st edition (June 10, 1999).

[12] M. Matz, J. Hubicka, A. Jaeger, M. Mitchell. *System V Application Binary Interface - AMD64 Architecture Processor Supplement*. June 2005, Available from `http://www.x86-64.org/documentation/`

[13] J. Miranda, E. Schonberg. *GNAT: On the Road to Ada 2005*. SigAda'2004, November 14-18, Pages 51-60. Atlanta, Georgia, U.S.A.

[14] J. Miranda, E. Schonberg, G. Dismukes. *The Implementation of Ada 2005 Interface Types in the GNAT Compiler*. 10th International Conference on Reliable Software Technologies, Ada-Europe'2005, 20-24 June, York, UK.

[15] J. Miranda, E. Schonberg, K. Kirtchov. *The Implementation of Ada 2005 Synchronized Interfaces in the GNAT Compiler*. SigAda'2005, November 13-17. Atlanta, Georgia, U.S.A.

[16] J. Miranda, E. Schonberg. *Abstract Interface Types in GNAT: Conversions, Discriminants, and C++*. 11th International Conference on Reliable Software Technologies, Ada-Europe'2006, June, Porto, Portugal.

[17] N. Trifunovic. *Calling Conventions Demystified*. `http://www.codeproject.com/cpp/calling_conventions_demystified.asp`

[18] Object Management Group. *Common Object Request Broker Architecture: Core Specification* Version 3.0.3, March 2004.

[19] *System V Application Binary Interface - Intel 386 Architecture Processor Supplement*. Prentice Hall Trade, Third Edition 1994, ISBN: 0-131-04670-5. Fourth Edition available from `http://www.caldera.com/developers/devspecs/abi386-4.pdf`

[20] *Welcome to SWIG*. `http://www.swig.org/`

[21] B. Stroustrup *Multiple Inheritance for C++* The C/C++ Users Journal, May 1999 issue `http://www-plan.cs.colorado.edu/diwan/class-papers/mi.pdf`

[22] S. Taft, R. A. Duff, and R. L. Brukardt and E. Ploedereder (Eds). *Consolidated Ada Reference Manual with Technical Corrigendum 1. Language Standard and Libraries*. ISO/IEC 8652:1995(E).

Springer Verlag, 2000. ISBN:
3-540-43038-5.

Interprocedural optimization on function local SSA form in GCC

Jan Hubička
SuSE ČR s. r. o
jh@suse.cz

Abstract

GCC is slowly shifting towards interprocedural and intermodule optimization. The paper describe experimental implementation of interprocedural optimization on single static assignment form (SSA) and give an guide to writing SSA aware interprocedural optimization pass using the new framework. Some simple improvements allowed by SSA optimizations are implemented and benchmarked to discuss pros and cons of interprocedural optimization on SSA compared to current GCC non-SSA implementation.

1 Introduction

The GCC compiler has a pretty mature intraprocedural optimizer framework that, despite the high number of different architectures GCC can target, is able to compete well with proprietary solutions that are specialized for single architectures.

Instead, the GCC interprocedural optimization framework is in its early stages. Several optimizations passes were implemented with varying success, but the limitations of the framework are clear, and more work is necessary in order to make the design of passes easier (partly addressed by this paper) and to make whole framework more scalable.

In this paper we consider reorganizing the interprocedural optimizations to work on the functions in single static assignment form (SSA form from now, see [Cytron91]) in order to strengthen the analysis and improve the flexibility of optimization passes.

One of major drawbacks of GCC interprocedural framework is the memory consumption caused by the GIMPLE intermediate language. Memory issues are so serious and are considered a major showstopper towards implementing link-time whole program optimization. While in our work we are not attempting to solve the memory issues, we are trying to make it no worse. There are SSA based intermodule optimizers in existence, such as LLVM [Lattner03] with significantly lower memory footprint, proving that this approach makes sense for a production compiler.

2 Design overview

During the tree-SSA project [Novillo03], the GCC intraprocedural optimization framework was reengineered to allow multiple intermediate languages and progressive lowering, instead

of optimizing directly on a very low level intermediate language (RTL).

Front-ends now are supposed to produce GENERIC, a high level intermediate language similar to C parse trees. GENERIC is then lowered to GIMPLE, a restricted subset of GENERIC that is essentially a three-address intermediate language, represented as trees and still with source level control flow representation. Further transformation include lowering of control flow statements into conditional jumps (with a control flow graph on top), and later construction of the SSA form. Later on, GIMPLE is brought back out of SSA form and function bodies are transformed into the RTL intermediate language used by code generation.

Despite a theoretical possibility to have optimizations working on different levels of GIMPLE, all tree-level intraprocedural optimizations done now (except for constant folding and basic control flow cleanups) are performed on GIMPLE in SSA form with control flow graph built. In contrast, the interprocedural optimizations (profiling, constant propagation, cloning, inlining and alias analysis) were performed initially on GENERIC, and now on low level GIMPLE before conversion to SSA form.

This design is believed to have smaller overall memory footprint, since the function bodies are expanded one at a time into the more memory intensive SSA form. On the other hand this imposes restriction on the pass ordering, since it is impossible to run any intraprocedural optimization passes before interprocedural optimization is finished.

3 Implementation details

The SSA data structures were modified to allow multiple functions in SSA form at once. To accomplish this, most of global variables holding SSA form of function have been localized to the `struct function` structure used to hold other function specific data. Some of datastructures don't localize easily however.

3.1 SSA version numbers

The SSA versions (names) of each variable are represented by unique tree nodes that are assigned unique integers (version numbers). These numbers are used not only for debug output, but also by optimization passes to store pass specific data in on-side arrays and bitmaps. This means that the numbers must be reasonably dense.

In the current implementation the SSA version numbers are kept local to each function. This means that interprocedural passes dealing with SSA versions from different functions at a time needs to keep hashtables based on the addresses of the SSA names instead of arrays. If this become an issue in future, either the local passes would need to be moved from arrays to hashtables or two SSA version numbers (global and local) would need to be assigned to each variable.

3.2 Referenced variable numbers

Similarly to version numbers, information on the referenced variables used to be kept in an array, so that local information could be easily attached. We changed this array to a hash table in the mainline compiler, at the beginning of our project.

3.3 Variable annotations

A lot of information about variables is stored in so-called variable annotations—structures

pointed to by direct pointer from variable declarations. The annotations currently hold both pass local data and data passed across optimization passes. Unfortunately the variable annotations can not be easily privatized, since annotations of global variables needs to be shared across multiple functions.

Actually, annotations mostly hold information that is local to optimization passes (or passed from a pass to subsequent one), so that it can easily be shared by intraprocedural passes on different function bodies. In fact the only field that was made private is `default_def`.

Aliasing information is also stored in the variable annotations, and is local to functions. Since the optimizer currently is not building any aliasing information in the early optimization passes, this does not cause any issues. (Aliasing information would also consume too much memory because of the virtual operands. It is probably impractical to build it for whole compilation unit). In future however we probably want to make the early optimization passes to take aliasing into account, at least in a limited fashion, so these issues should be resolved in longer term.

The aliasing information should probably be moved to separate datastructure allocated only for pointers, where it is actually used. This should conserve memory as well as make the representation more flexible. However, the only mean to assign the datastructure to variable currently is an hashtable. This may prove too slow because aliasing info is accessed frequently during the operand scan.

From the memory consumption standpoint, the variable annotations are poorly designed, since the lifetime of temporary objects is extended across whole SSA optimization queue. Since the annotations accounts roughly of 7% of overall memory consumption, we also suggest trying to move as much as possible out of the annotations, either eliminating them completely, or keeping only the global information that need to be computed for each variable in the function. Ironically no such information is stored currently in the annotations at all and it seems also natural to store such information directly in the variable declarations.

4 Pass manager

The GCC pass manager was extended to allow interprocedural passes. The toplevel pass queue now consist of interprocedural passes, while the subpasses are considered to be intraprocedural (the passmanager automatically takes care to execute the subpasses for each function in compilation unit). In future this might be relaxed to allow interprocedural subpasses but there is no reason for doing so at the moment.

Earlier implementation of interprocedural optimization in GCC were performing analysis of all functions for all interprocedural passes first and later applying the results of interprocedural analysis on each function locally. Each pass had defined `analyze` method called for each function, `execute` method called once all functions was analyzed and finally `modify` method called again for each function separately. This was believed to allow more scalable intermodule optimization.[1]

This scheme, however, turned out to be difficult to manage because of the interactions between passes. It seems unnecessary to introduce such a restriction on interprocedural pass

[1] If all analysis are performed before any modification, the analysis phase can be done locally at compilation time and written into fake object files in function summaries. The link-time optimizer then can read the summaries first and perform interprocedural propagation of collected data. Compilation then can be done at function basis with loading only the necessary function bodies into compiler memory reducing peak memory usage. See [Hubička04] for more discussion on the topic.

designers in such an early stages of development on the framework. Once the implementation is sufficiently mature and we have better understanding of the interactions of individual passes, we might revisit this idea. It might be however more profitable to simply forget about it and concentrate more on reducing memory usage of our intermediate language.

5 Order of optimization passes

At present the interprocedural pass queue includes the following passes:

1. Removal of unreachable functions and variables.
2. Early inlining.
3. Early intraprocedural passes (profiling, control flow graph cleanup, conversion to SSA, constant propagation, value range propagation and dead code removal) and rebuilding the callgraph edges.
4. Interprocedural constant propagation and function cloning.
5. Inlining
6. Removal of unreachable functions.
7. Alias analysis.
8. Type escape analysis.
9. Points-to analysis.
10. Intraprocedural passes and final output of function.
11. Output of static variables still referenced by optimized code.
12. Local optimization and output of individual functions.
13. Output of static variables still referenced by the optimized function bodies.

The order is generally natural, perhaps with the exception of early inlining pass. Early inlining was introduced to help C++ testcases with high abstraction penalty and employs simplified heuristics, whereby we only inline functions that are smaller than the expected call overhead. In particular, wrapper functions are eliminated reducing significantly (by more than factor of 10) the profile instrumentation overhead on the TraMP3d testcase [TraMP3d]. The pass should also improve the effectiveness of early local optimization passes on testcases with a lot of small functions, where these passes are otherwise close to useless.

Interprocedural constant propagation is performed before inlining so that the inliner can deal more aggressively with the specialized bodies of functions. Unfortunately at the time of writing the paper, the interprocedural constant propagation is limited to create at most one clone of each function body (and only in the case the operand is the same constant in all calls to the function but would not be propagated without cloning because function might be called externally), which makes the interaction with the inliner suboptimal. The really interesting case of function called with different constant operands form different places specialized into multiple forms is not considered for the moment.

To enhance the effectiveness of interprocedural alias analysis, it might be also be profitable to redo early local optimization passes before it. However, the lack of local alias analysis information in early optimization passes causes any statement accessing memory to be considered volatile and thus left unoptimized. As a result, early optimizations won't improve code enough to make any difference for aliasing analysis.

6 Writing an interprocedural GCC pass

Thanks to the new pass manager framework, the interface to interprocedural passes is very similar to interface to local optimization passes. The `tree_opt_pass` structure needs to be filled in:

```
struct tree_opt_pass my_pass =
{
  "pass_name"
  gate_function,
  execute_function,
  NULL, NULL,
  /* Local subpasses */
  0, /* Static pass number. */
  TV_PASS, /* tv_id */
  PROP_cfg | PROP_ssa,
  /* properties_required */
  0, /* properties_provided */
  0, /* properties_destroyed */
  0, /* todo_flags_start */
  TODO_dump_cgraph
  | TODO_dump_func,
  /* todo_flags_finish */
  0 /* Used by RTL passes */
};
```

The function `gate_function` is used to control execution of the pass and `execute_function` is called when the pass should be performed. The pass then needs to be registered into optimization queue in `passes.c`.

6.1 Walking functions in program

To analyze functions, one should look at linked list of callgraph nodes. The list contains all functions, external or internal. To explore only the functions whose body is known, the flag `analyzed` needs to be tested:

```
struct cgraph_node *node;
for (node = cgraph_nodes; node;
     node = node->next)
  if (node->analyzed)
    ....
```

6.2 Walking callgraph

The callgraph edges (call sites) are represented as linked lists of `struct cgraph_edge` objects. Each callgraph node points to list of callers by `node->callers` and list of callees by `node->callees`. These lists are built during analysis pass and are maintained up to date until final local optimization passes that are destructive to callgraph.

6.3 Walking function bodies

Most functions that manipulate control flow graph or SSA form are not yet aware of multiple functions. When the control flow graph needs to be manipulated, it is easiest to change the `current_function_decl` and `cfun` pointers. It is also necessary to setup the control flow graph hooks:

```
push_cfun
 (DECL_STRUCT_FUNCTION
    (node->decl));
tree_register_cfg_hooks ();
current_function_decl
 = node->decl;
```

After the analysis is completed, it may be necessary to destroy dominance info (if computed) since this datastructure is global.

```
free_dominance_info
 (CDI_DOMINATORS);
free_dominance_info
 (CDI_POST_DOMINATORS);
pop_cfun ();
```

If the transformations affected the callgraph, it is necessary to update the callgraph datastructure, either by hand or via `rebuild_cgraph_edges`. If substantial changes were made to the function body, it might be profitable to re-do early optimizations by `tree_early_optimization_passes` that subsume `rebuild_cgraph_edges`.

6.4 Walking variables

Variables are, similarly to functions, grouped in a linked list:

```
struct cgraph_varpool_node *vnode;
for (vnode
       = cgraph_varpool_nodes_queue;
     vnode;
     vnode = vnode->next_needed)
  analyze_variable (vnode);
```

7 Experimental results

In this section we compare the memory consumption, compilation time and quality of produced code of our experimental branch compared to mainline compiler from date of last merge to the branch (2006-04-06).

7.1 Cost of SSA form

The SSA form and is more memory intensive because of the `SSA_NAME` wrappers and other datastructures. We made no effort to reduce memory usage of the SSA form and merely localized all datastructures needed. Still, the negative effect is slightly outweighed by the scalar cleanups (constant propagation, value range propagation, copy propagation, forward propagation and dead code removal) performed just after converting to SSA form.

On TraMP3d testcase (consisting of large number of tiny function), the memory consumption increase after the lowering passes is 211 MB compared to 209 MB on mainline compiler. This testcase should expose near to worst-case behavior since it consists of number of very tiny functions. On `combine.c` (GCC module) testcase, the memory is 8.4 MB compared to 8.5 MB for mainline compiler. It looks like the simple cleanups enabled by SSA conversions are effective enough to pay back the extra memory cost.

7.2 Cost of performing inlining on the whole compilation unit

The extra memory cost of moving the inline transformation early as discussed in Section 4 is measured by introducing garbage collector pass just after the inliner pass. This way, we can compute how much memory is still referenced.

On the IPA branch, the memory consumption for TraMP3d testcase just after inlining peaks at 390 MB, while on mainline it is only 184 MB. Again, TraMP3d should expose near to worst case behavior. There are no noticeable differences for `combine.c` testcase since very little inlining is performed there.

While this memory increase seems serious, it can be argued that it is bound by overall unit growth parameter of the inliner and thus should not lead to uncontrolled memory consumption problems. However, we felt it was too early to change inlining order in GCC 4.2, and decided to delay its submission for later version when we can use the extra flexibility introduced by the change.

It is trivial to modify our experimental branch back to mainline behavior in this case and it can be easily verified that the overall memory peaks of both compilers are same then.

7.3 Overall compilation time

Compilation time is affected by multiple factors. Obviously the optimization queue was extended by another 4 scalar optimization passes, that should account together by about 0.5–2% of compilation time. Additionally, the inliner now has to maintain the SSA form, with some cost in compilation time.

However, the early optimizations save work after inlining and can reduce the memory footprint of the function body just after the first alias analysis pass. Thus, overall compiler performance is sped up for some testcase, as can be seen by decrease in overall allocation of GGC memory in both `combine.c`. As a result, the compile time performance tests show a pretty mixed picture. `combine.c` compiles about 2% slower, but the whole bootstrap is a few percent faster, because early optimization is effective on `insn-attrtab.c` and saves a lot of memory for the aliasing information.

For the TraMP3d testcase, compilation time benchmarks are not directly comparable because the inliner decisions are significantly different. The experimental compiler seems to be about 4% slower because of more aggressive inlining; on the other hand, the resulting binary is both smaller and slightly faster.

The overall compilation time of SPECint benchmarks remains about the same for single file optimization and improve from 510s to 490s for whole program optimization.

8 Runtime performance

The SPECint 2000 benchmark results (except for twolf benchmark being misscompiled) of mainline compiler are compared to the experimental branch in Table 1. The tests was compiled with `-O3 -ffast-math` optimization setting. Three levels of interprocedural optimizations are considered: with single file optimization, pseudo-link-time optimization (where all the source files are parsed first, via `--combine`, and optimized as single compilation unit) without and with whole program assumptions (via `-fwhole-program` that make all functions and variables local with the exception of `main()`. GCC is not able to combine C++ modules into single compilation unit and thus eon benchmark (only C++ benchmark in SPECint) is always compiled with the same settings.

For broader picture, the comparison to Intel C++ compiler (ICC 9.0) is included too. The single file optimization benchmark was performed with `-axW -ip -O3`, while whole program with `-axW -ipo -O3`. It shall be noted that ICC is not able to tune for AMD chips and thus suboptimal pentium4 tuning was used instead. ICC is able to do full linktime optimization for eon benchmark too.

The results are not surprising. The branch brings up to 2% improvements, almost entirely because of improved inliner decisions in the crafty benchmark, and improved aliasing in the gcc benchmark. Both are caused by early optimization. The improvements are mostly visible in link time optimization, without whole program assumptions that (even on mainline) simplify the inlining of functions that are called just once in whole program.

Noticeable improvements are also in overall code size savings of roughly 117 Kb, 467 Kb, 382 Kb[2] from overall size of binaries (compiled with single file, link time and whole program setting accordingly), suggesting that inliner is able to make better decisions in both performance and code size metrics.

[2]The SPEC binaries compiled with dynamic linking and without debug info are about 5MB.

Author also measured 7% runtime speedup of TraMP3d benchmark but due to its overall instability relative to inliner decisions, this might be just "luck" (earlier version of compiler had opposite results). It can be expected however, that for more complex (and/or less hand optimized) testcases than SPECint benchmarks, the benefits will be more noticeable.

9 Conclusion

As expected, the new SSA based intermodule framework improve the runtime performance of some benchmarks that are sensitive to inlining. The benefits come mostly from pre-inline optimization passes and are supposed to increase as the interprocedural optimization is made more effective.

Surprisingly, the memory consumption problems caused by transition to SSA seems to be no worse than with the current (non-SSA) framework. Performing inlining before other interprocedural passes, however, causes significant memory consumption growth—which is fortunately limited by overall unit growth limit of inliner. So far, this change is however independent on the rest of work on IPA branch, so both issues can be considered separately.

Sadly, memory consumption in mainline GCC is considered to be one of the major showstoppers on the way towards intermodule optimization and radical reductions will be necessary in future. Perhaps, completely switching to a new intermediate language is necessary, as discussed in [Lattner03] or developing different memory representation of GIMPLE not based on nested tree structure.

10 Acknowledgments

The plan for moving interprocedural optimization into SSA form and was discussed at the 2005 GCC Summit with Daniel Berlin, Diego Novillo, and Kenneth Zadeck. Daniel Berlin contributed patch to avoid `referenced_vars` array. Razya Ladelsky adapted interprocedural constant propagation to work on SSA form. Olga Golovanevsky contributed a devirtualization pass. Paolo Bonzini proofread the paper, helped to clarify it, and significantly reduced the amount of Czenglish.

References

[Cytron91] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck, Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, *ACM Transactions on Programming Languages and Systems 13,4* (1991), 451–490.

[Novillo03] D. Novillo, Tree SSA — A New Optimization Infrastructure for GCC *Proceedings of the 2004 GCC Developers Summit* (2003), `http://www.gccsummit.org/2003`

[Hubička04] J. Hubička, Call graph module in GCC, framework for inter-procedural optimization, *Proceedings of the 2004 GCC Developers Summit* (2004), `http://www.gccsummit.org/2004`

[Lattner03] C.A. Lattner, Architecture for a Next Generation GCC, *Proceedings of the 2004 GCC Developers Summit* (2003), `http://www.gccsummit.org/2003`, `http://www.llvm.org`

[TraMP3d] Template heavy C++ testcase
available at:
http://www.suse.de/
~gcctest/c++bench/

Benchmark	single file			linktime		whole program		
	non-SSA	SSA	ICC 9.0	non-SSA	SSA	non-SSA	SSA	ICC 9.0
gzip	1206	1214	1225	1188	1210	1158	1175	1323
vpr	858	855	857	848	848	864	864	859
gcc	1072	1050	960	1074	1077	1101	1100	980
mcf	539	543	543	543	541	543	544	543
crafty	1944	2066	2100	1864	2116	2086	2097	2211
parser	828	829	801	826	832	836	830	811
eon	2414	2473	1803	2422	2477	2422	2478	2033
perlbmk	1479	1460	1419	1407	1472	1464	1470	1513
gap	1110	1147	1085	1174	1162	1194	1210	1094
vortex	1708	1737	1626	1824	1859	1857	1905	1917
bzip2	1020	1020	1005	1021	1019	1058	1072	1011
Geomavg	1188	1200	1138	1189	1214	1214	1227	1194

Table 1: Runtime performace (in SPECint ratios, bigger is better)

Improved Superblock Optimization in GCC

Robert Kidd and Wen-mei Hwu
Center for High Performance and Reliable Computing
University of Illinois at Urbana-Champaign
{rkidd,hwu}@crhc.uiuc.edu

Abstract

Superblock scheduling is a common technique to increase the level of instruction level parallelism (ILP) in generated code. Compared to a basic block, the Superblock gives an optimizer or scheduler a longer range over which instructions can be moved. The bookkeeping necessary to execute that move is less than would be necessary inside an arbitrary trace region. Additionally, the process of forming Superblocks generates more instructions that are eligible for movement. These factors combine to produce a significant increase in the ILP in a section of code.

By identifying the key feature of Superblock formation that allows this increase in ILP, we can generalize the concept to describe a class of similar optimizations. We refer to techniques in this class as *structural* techniques. Combining several optimizations in this class with aggressive classical optimization has been shown in the OpenIMPACT compiler to be particularly useful in developing ILP when compiling for the Itanium processor.

As a motivation for our work, we present an investigation into the value of structural compilation in the OpenIMPACT compiler. In this domain, structural techniques have been credited with a 10% to 13% increase in code performance over a compiler that implements only classical optimizations.

As a first step toward developing structural compilation techniques in GCC, we implemented Superblock formation at the Tree-SSA level. By performing structural transformations early, we give the compiler's high level optimizers an opportunity to specialize the transformed program, thereby cultivating higher levels of ILP. The early results of this modification are mixed, with some benchmarks improving and others slowing. In this paper, we present details on our implementation and study the effects of this structural transformation on later optimizations. Through this, we hope to motivate future work to implement and improve optimizations that can take advantage of the transformed control flow.

1 Introduction

As an EPIC (Explicitly Parallel Instruction Computing) processor, the Intel Itanium Processor Family relies heavily on the compiler to extract performance from a program. The architecture provides a large number of functional units, but the hardware does not dynamically schedule instructions to discover instruction level parallelism. Instead, the compiler

must find instructions that can execute in parallel during its code generation and scheduling stage. Within the context of a traditional optimizing compiler, control flow presents barriers that make it difficult to statically generate a parallel schedule. Superblock formation [3] is one technique to overcome the restrictions of control flow.

Originally, Superblock formation was developed solely to support instruction scheduling. In this traditional method, a Superblock is constructed using trace formation and tail duplication. The result of Superblock formation is a long single-entry, multiple-exit chain of basic blocks. This Superblock is then passed to a variant of a list scheduler that is capable of moving instructions across basic block boundaries. Compared to trace scheduling [1], the lack of side entrances into a Superblock simplifies the task of moving instructions between basic blocks.

Superblock formation is useful for more than instruction scheduling. Tail duplication eliminates all but one of the control flow paths into a basic block. As a result, much tighter bounds can be drawn on the state of variables at the block entrance. Optimizations such as constant propagation can be profitably applied to the duplicated tail to specialize the block, counteracting the code expansion inherent in Superblock formation.

In the OpenIMPACT compiler [6], Superblock formation is one of a class of structural compilation techniques. Other members of this class include function inlining and hyperblock formation. These optimizations use profile feedback to radically alter the control flow graph (CFG) to produce straight line sections for the typical course of execution. Traditional optimizations are then applied to this CFG to specialize blocks and discover ILP. While this technique does entail a large amount of code duplication, it packs the useful instructions into a tight schedule, resulting in little increase in instruction cache pressure.

Within GCC, we modified the pre-existing RTL Superblock formation pass to work at the Tree-SSA level. The overall performance change due to this modification is minimal. Certain benchmarks run faster, while others run slower. However, this patch gives us a starting point to investigate the possibility of applying structural optimization techniques within GCC. In the following sections, we will discuss the effects, good and bad, of the patch and suggest a future direction based on our experience with OpenIMPACT.

The primary goal of our work is to improve the performance of code compiled by GCC for the Itanium processor. However, we believe that the structural compilation model will also work well on traditional superscalar processors. The code specialization derived from traditional optimizers should apply to any architecture.

2 The Superblock and structural compilation

Although it has been throughly covered in literature, it is useful to review here the process through which a Superblock is constructed. This process is illustrated in Figure 1. Starting with a control flow graph annotated with profile weights (part (a)), the *trace formation* pass determines the typical path of execution. This pass constructs the hot trace by following the typical path of execution until execution frequency falls below some threshold or a loop back edge is encountered. The resulting trace is highlighted with a dotted outline in Figure 1(b). After constructing the trace, the *tail duplication* pass eliminates side entrances. If a basic block on the trace has more than one predecessor, as is the case for block z, that block is

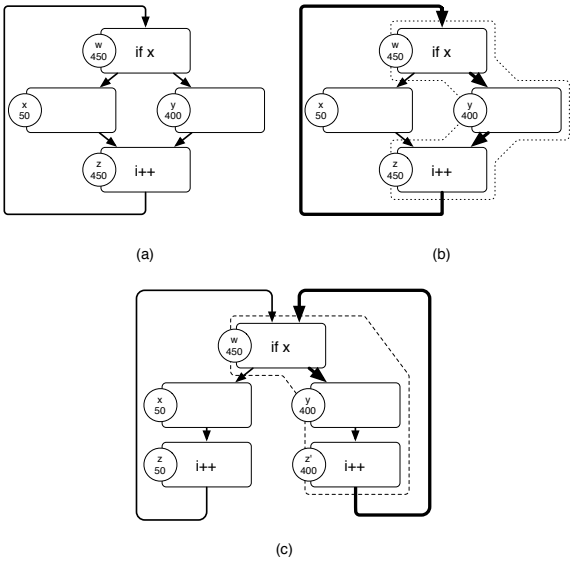


Figure 1: Superblock formation process

copied and the duplicate inserted into the trace. The sole predecessor of the duplicate will be on the trace, as seen in part (c). The result of this process is a Superblock, a series of basic blocks that has a single entrance at the top and one or more side exits. This is indicated by the dashed line in Figure 1(c).

Structural compilation [5] is a generalization of Superblock-based optimization. Structural optimizations share a common attribute in that they target side entrances (join points) of a frequently executed trace. Inside the join block and below, program context is blurred as the compiler must assume that any predecessor may have executed before the join block. Duplicating the join block and its children onto the hot trace has the effect of extending the context of the trace into the duplicated block. Specialization of the duplicated code through classical optimizations reduces the number of instructions on the trace, and use of features such as speculation extend the range over which instructions can move. The end result for the typical path of execution is a shorter schedule with higher levels of ILP.

In the structural model, code duplication is performed early in the compilation path. Code expansion limits are raised to allow more code duplication. The expansion must be done somewhat speculatively, as it is difficult to predict the precise effect later optimizations will have, given a certain level of duplication. Therefore, a large amount of code duplication is inherent to the structural model. The code generated using this model is bigger than that generated with a traditional model, so one might expect a degradation in instruction cache efficiency. However, the expansion is mitigated by two factors. First, the Itanium processor provides large caches, which help offset the increase in code size. The second factor, which applies to all architectures, is that the code duplicated by these techniques should appear outside the path of typical execution. By specializing for the typical path, we reduce the number of redundant instructions and pack the useful ones more closely. This reduces the number of cache lines that need to be fetched and increases the number of useful instructions per line.

The compilation method used in GCC to this point more closely resembles what we term the *incremental* approach. In this method, traditional global scheduling techniques are evolved and refined to deliver higher levels of ILP. Control flow may be altered, but it is done in response to the needs of the optimizer or scheduler. The overall CFG tends to remain the same, and can restrict optimization opportunities. An analogy with simulated annealing is apt. The incremental model is a well tested and reliable method to arrive at an optimal point in the range of possible schedules. However, in many cases, this optimal schedule is a local minimum. By applying structural techniques, we hope to move the optimizer's starting point to a place where a more optimal schedule can be found.

Benchmark	g-no-spec	I-CL	I-NS	I-CS
164.gzip	506	602	677	752
175.vpr	498	607	644	719
181.mcf	257	332	330	341
186.crafty	591	646	677	704
197.parser	494	520	523	541
252.eon	517	364	428	429
254.gap	421	558	573	599
256.bzip2	426	652	658	698
300.twolf	553	724	830	921
geomean	462	540	575	609
geomean2	456	567	596	637

Heading	Compiler version	options
g-no-spec	GCC as of 3/6/2006	-O3, profile feedback enabled, no speculation
I-CL	OpenIMPACT	classical optimizations only
I-NS	OpenIMPACT	structural opti, no speculation, pointer analysis and modulo scheduling disabled for 252.eon
I-CS	OpenIMPACT	control speculation, otherwise like I-NS
Scores generated on an Itanium 2 1.0 GHz, 1.5M L3 cache		

Table 1: Comparison of classical and structural optimizing compilers

Table 1 presents estimated[1] SPECint2000 scores to illustrate the benefit that a structural optimization pass can have on code performance. In this table, all benchmarks have been compiled using profile feedback.

The g-no-spec column is our baseline GCC configuration. This is a recent version of GCC mainline that lacks speculation support. This configuration includes the RTL level Superblock formation pass[2]. I-CL is our baseline OpenIMPACT configuration. Classical optimizations are run, but structural techniques are disabled. This configuration is roughly equivalent to g-no-spec. I-NS turns on structural transformation, and I-CS turns on control speculation.

The geomean row shows the geometric mean for all nine benchmarks. Geomean2 excludes 252.eon from the comparison, as GCC's score is inflated relative to OpenIMPACT for the purposes of this paper. OpenIMPACT has a long history as a C compiler, but C++ support has only been added recently. Even now, C++ support is implemented by lowering the incoming code to C and compiling. This is functional, but inefficient. As a proper C++ compiler, GCC has a distinct advantage over OpenIMPACT on 252.eon that is outside the scope of this paper.

A comparison of the I-CL and g-no-spec columns shows that OpenIMPACT's classical optimization path improves performance approximately 24% over GCC. This can be attributed to better alias analysis in OpenIMPACT and more aggressive settings on the clas-

[1]These numbers are the result of runs on real hardware, but do not reflect a rigorous SPEC run. We have followed SPEC's run rules with respect to training/reference inputs and consistency of optimization settings, but we have been unable to complete a full run of the suite. We therefore label these results "estimated" as per SPEC's rules for research use.

[2]Results from the Tree-SSA level Superblock formation pass are presented in Section 3.

sical optimizers. Comparing I-CL and I-NS, we see that turning on structural transforms in OpenIMPACT gives a 5% improvement in performance. We believe 5% to be a reasonable estimate for the performance benefit possible in GCC with the addition of structural transformation.

The I-CS column shows the effect of combining control speculation and structural transforms in OpenIMPACT. We see a 12% to 13% increase in performance over the classical configuration when these two features are combined. Our Superblock work should therefore fit in nicely with other work in progress to add speculation support to the compiler.

Although it is not useful as a comparison between GCC and OpenIMPACT, it is interesting to note that within OpenIMPACT, the structural technique is particularly useful for 252.eon. In this case, indirect call profiling and function inlining combine to reduce the cost of virtual method calls. Although OpenIMPACT does not support static C++ optimizations such as class hierarchy analysis, it is able to approximate them in some cases through structural transformation. Even with proper C++ support, structural transformation will likely still be beneficial for heavily object-oriented code.

3 Technical details and performance results

At the RTL level, GCC has included a Superblock formation pass for some time [2]. This pass prepares Superblocks for the Haifa interblock scheduler, but runs after most of GCC's optimization passes. Structural compilation requires this transformation to happen early in the compiler; to achieve this, we implemented a Superblock formation pass at the Tree-SSA level.

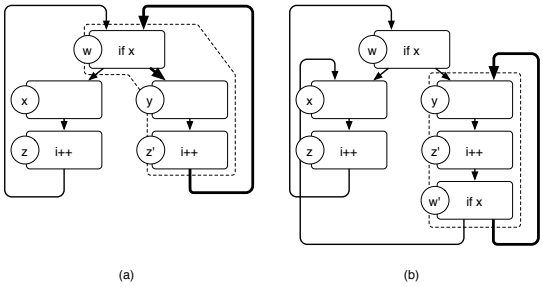


Figure 2: Loop header duplication to form a simple loop from a Superblock

Because the CFG manipulation API is shared between RTL and Tree-SSA, the modifications necessary to run Superblock formation at the Tree level were minor. The primary change necessary was to add SSA variables generated in the duplicated tail to the ϕ-nodes for its successor blocks. We also adjusted the trace formation routine so that it generates Superblock loops that are of a simple form that can be processed by the loop optimizers. This is illustrated in Figure 2.

Figure 2(a) shows the problem that can occur when forming a Superblock inside a loop. By tail duplicating block z to form z', we created a Superblock for the common case inside the loop. However, because tail duplication stopped when it found the loop backedge, we were forced to add a second control flow arc back to the loop header. The loop is no longer in a simple form that can be processed by GCC's loop optimizers. If trace formation instead follows the backedge one time and stops after duplicating the loop header, our Superblock forms a simple loop, as in Figure 2(b). The form of this simple loop increases the effectiveness of later optimizations, as will be seen in Section 4.2.

The final change we made was to remove the Superblock layout pass. We leave code layout to the basic block reordering stage later in RTL.

The Tree-SSA Superblock formation executes immediately after SSA form is constructed and before any optimizers. Because tail duplication has the potential to create new pointer loads and stores, it is possible that formation will interfere with alias analysis. We have tested building Superblocks before and after alias analysis, but neither configuration is noticeably better than the other.

Our comparison baseline is GCC mainline revision 112576, which supports Superblock formation at the RTL level. Against this baseline, we compare a build of the ia64-improvements branch. This branch includes the changes to mainline through revision 112576, but uses the Tree-SSA level Superblock formation pass. We run both versions of GCC with the -O3 flag and with profile feedback enabled. Both versions support speculation on Itanium. For the ia64-improvements branch, we set an aggressive Superblock expansion limit of 300% as opposed to the 100% limit used with the RTL level pass. We ran performance numbers on three architectures: x86, x86_64, and Itanium. Machine specifications are listed in Table 2.

Tables 3 and 4 show the estimated change in performance for selected SPEC2000 benchmarks. For each architecture, we present the score for GCC mainline (stock), the score for the ia64-improvements branch (SB), and the percent difference between the two. We have not yet fully analysed the cause of the slight performance degradation on x86_64, but we believe this processor to be more sensitive to code scheduling than x86. Forming Superblocks at the Tree-SSA level is beneficial for x86 and on Itanium for the floating point benchmarks. Superblock formation tends to produce simpler, straighter loops that are more palatable to loop optimizers. This helps explain the performance improvement in loop-intensive floating point code.

For 191.fma3d on x86, Superblock formation gives a significant performance improvement. This improvement is due to a drastic reduction in the time spent executing one function, `platq_stress_integration`. The prologue of this function sets up a number of variables using `MIN` and `MAX` statements. These statements are biased, so they become simple assignments inside the Superblock. Constant propagation can then simplify the body of the function.

For integer codes, the results are more mixed. These control-intensive benchmarks are where we expect to see a significant performance improvement on Itanium, and yet we see a negligible change. This is not entirely unexpected. As we have observed in OpenIMPACT, the combined efforts of several structural techniques are often necessary to get a significant performance improvement. These passes must push the program's CFG to a critical point where optimizations can radically specialize the typical path of execution. It is encouraging to note that 181.mcf, 186.crafty, and 256.bzip2 do improve. 186.crafty is an extremely control-intensive chess simulator, and the improvement here replicates a result from OpenIMPACT detailed in [5]. The improvement in 181.mcf, a memory-intensive benchmark, can be attributed to the combined effect of Superblock formation and data speculation. This benefit of combining speculation and structural transformation has also been observed in OpenIMPACT. These results strongly suggest that results from OpenIMPACT will apply to GCC as development progresses. We will expand on effect of Superblock formation on 256.bzip2 in Section 4.2.

Finally, Table 5 compares the size of the benchmark executable across the two compiler configurations. Even with an aggressive Superblock expansion limit, executable size does not significantly increase. By duplicating blocks, Superblock formation gives the op-

Configuration	Processor	Operating System
ia64	Itanium 2, 1.6 GHz, 6 MB L3	Linux 2.6.8, LP64
x86_64	Athlon 2800+, 1.8 GHz, 512 KB L2	Linux 2.6.12, LP64
x86	Xeon, 2.4 GHz, 512 KB L2	Linux 2.4.18, LP32

Table 2: Machine configurations

	ia64			x86_64			x86		
Benchmark	stock	SB	%	stock	SB	%	stock	SB	%
164.gzip	810	810	0.00%	914	850	-7.00%	676	696	2.96%
175.vpr	929	923	-0.65%	771	760	-1.43%	484	472	-2.48%
175.gcc				1032	1033	0.10%	922	947	2.71%
181.mcf	682	706	3.52%	587	587	0.00%	535	544	1.68%
186.crafty	942	983	4.35%	1531	1586	3.59%	460	477	3.70%
197.parser	901	901	0.00%	859	864	0.58%	731	763	4.38%
252.eon	792	785	-0.88%				831	790	-4.93%
254.gap	651	651	0.00%	999	1004	0.50%			
255.vortex				1454	1532	5.36%			
256.bzip2	798	825	3.38%	897	889	-0.89%	580	585	0.8%
300.twolf	1039	944	-9.14%	792	793	0.13%	606	611	0.83%
geomean	830	830	-0.01%	947	947	0.05%	631	638	1.04%

Table 3: Estimated change in SPECint2000 performance

	ia64			x86_64			x86		
Benchmark	stock	SB	%	stock	SB	%	stock	SB	%
168.wupwise	605	578	-4.46%	1186	1140	-3.88%	903	918	1.66%
171.swim	773	804	4.01%	1142	1138	-0.35%	634	633	-0.16%
172.mgrid	346	347	0.29%	755	751	-0.53%	479	478	-0.21%
173.applu	519	538	3.66%	900	895	-0.56%	666	665	-0.15%
177.mesa	976	986	1.02%	1394	1383	-0.79%	483	487	0.83%
179.art	2716	2730	0.52%	777	785	-1.03%	272	268	-1.47%
183.equake	511	500	-2.15%	1085	1097	1.11%	960	967	0.73%
187.facerec	583	602	3.26%	791	772	-2.40%	477	479	0.42%
188.ammp	847	868	2.48%	882	884	0.23%	400	391	-2.25%
189.lucas	815	864	6.01%	1220	1200	-1.64%	546	538	-1.47%
191.fma3d	294	294	0.00%	870	874	0.46%	464	525	13.15%
200.sixtrack	371	368	-0.81%	451	449	-0.44%	429	431	0.47%
301.apsi	580	579	-0.17%	839	834	-0.60%	502	497	-1.00%
geomean	638	644	1.01%	912	906	-0.65%	527	531	0.75%

Table 4: Estimated change in SPECfp2000 performance

	int				fp		
Benchmark	stock	SB	%	Benchmark	stock	SB	%
164.gzip	1069787	1071587	0.17%	168.wupwise	1629360	1633480	0.25%
175.vpr	1310609	1307185	-0.26%	171.swim	1590538	1591482	0.06%
176.gcc	5836321	5749873	-1.48%	172.mgrid	1588244	1488644	0.03%
181.mcf	942326	942566	0.03%	173.applu	1691129	1691929	0.05%
186.crafty	1413018	1505658	-0.52%	177.mesa	2444067	2435987	-0.33%
252.eon	6984346	6986010	0.02%	179.arg	1016630	1020022	0.33%
254.gap	2253387	2247859	-0.25%	183.equake	1020942	1015102	-0.57%
255.vortex	2398115	2366931	-1.30%	187.facerec	1804178	1806962	0.15%
256.bzip2	2398115	2366931	-0.12%	198.ammp	1387201	1388161	0.07%
300.twolf	1519425	1516401	-0.20%	189.lucas	1677890	1680458	0.15%
				191.fma3d	4125512	4124912	-0.01%
				200.sixtrack	3890478	3887366	-0.08%
				301.apsi	1897353	1900977	0.19%

Table 5: Effect of Superblock formation on executable size (in bytes) for ia64

timizers more opportunity to eliminate instructions, controlling code expansion and increasing performance.

4 Analysis of performance change

On Itanium, 256.bzip2 and 300.twolf showed a significant change in performance. This section delves deeper into these two benchmarks to determine why performance improved or diminished.

We analysed the benchmarks using the i2prof.pl and q-tools packages. These packages use HP's libpfm library and pfmon utility to retrieve values from the Itanium performance counters to analyse.

i2prof.pl is a Perl script that wraps the pfmon utility to record raw counter values for the entire run of a program. Various performance metrics are determined from these counters, but they apply to the entire program and cannot be attributed to individual functions or lines.

```
for ( node = list; node;
      node = node->next ) {
   a = ...
   if (condition)
      b = ...
   else
      b = a
   *arg += b ...
}
```

Figure 3: Kernel from `new_dbox_a`

Q-tools provides the q-syscollect utility, which performs statistical sampling using the Itanium performance monitoring unit. At a specified interval, the program is interrupted and the program counter (PC) of the instruction triggering the event being monitored is recorded. Q-tools also includes the qprof program, which generates a gprof-style per-function report from the per-PC data.

4.1 300.twolf

300.twolf slowed by 9% when Superblock formation was performed at the Tree-SSA level. A

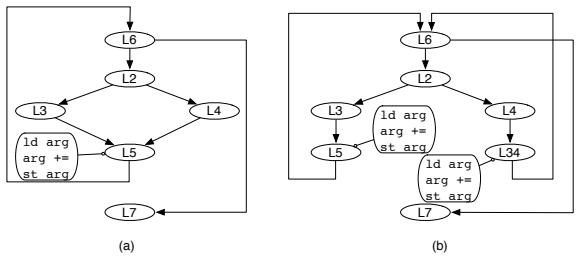


Figure 4: Structure of `new_dbox_a` before and after Superblock formation

comparison of execution profiles showed that much of the extra execution time could be attributed to the function `new_dbox_a`. This function is called approximately 120 million times during execution. Figure 3 illustrates the kernel of this function.

In this loop, `arg` is an integer pointer argument used to return a value from the function. The integer `arg` points to is updated, but the value of `arg` is not changed inside this function. Because of this, GCC's partial redundancy elimination (PRE) stage is able to move the load and store of `arg` out of the for loop.

Figure 4 shows the control flow graph of this kernel before and after Superblock formation. Tail duplication moves a copy of the update of `arg` onto both sides of the biased `if` statement, shown in Figure 4(b). In this case, the Superblock formation pass did not duplicate the loop header as we would expect. The cause of this is being investigated. If Superblock formation had duplicated the loop header, we would expect the PRE pass to move the load out of the more frequently executed side of this loop. It is not yet clear whether GCC's alias analysis framework is strong enough to completely remove the load from the Superblock loop. We plan to investigate further the interaction between Superblock formation and alias analysis.

```
L6:  j = 0;
     tmp = yy[j];
L8:  while (ll_i != tmp) {
L7:   j++;
      tmp2 = tmp;
      tmp = yy[j];
      yy[j] = tmp2;
     }
L9:  yy[0] = tmp;

     if (j == 0)
L10:  /* do something */
      /* yy is not touched */
     else
L11:  /* do something else */
      /* yy is not touched */
      /* L12 and L20 appear here */
L21: ...
```

Figure 5: The core of `generateMTFValues` from 256.bzip2

4.2 256.bzip2

Performance of 256.bzip2 increased by approximately 3% with the addition of the Tree-SSA Superblock pass. The execution profile did not show a significant difference in the run time of any one function, so we used i2prof.pl to collect overall performance statistics for the program. These statistics showed that the Superblock version experienced fewer L1D cache misses than the standard version. Sampling the `L1D_READ_MISSES_ALL` counter with q-syscollect, we were able to locate a loop in `generateMTFValues` that experienced a decreased number of cache stalls.

The loop in question is shown in Figure 5. This loop searches for a specific character in the array `y`, determines the index `j` of that character, rotates elements 0 through `j - 1` to positions 1 through `j`, and writes the desired character to element 0. The important feature to note is that this loop does a number of single

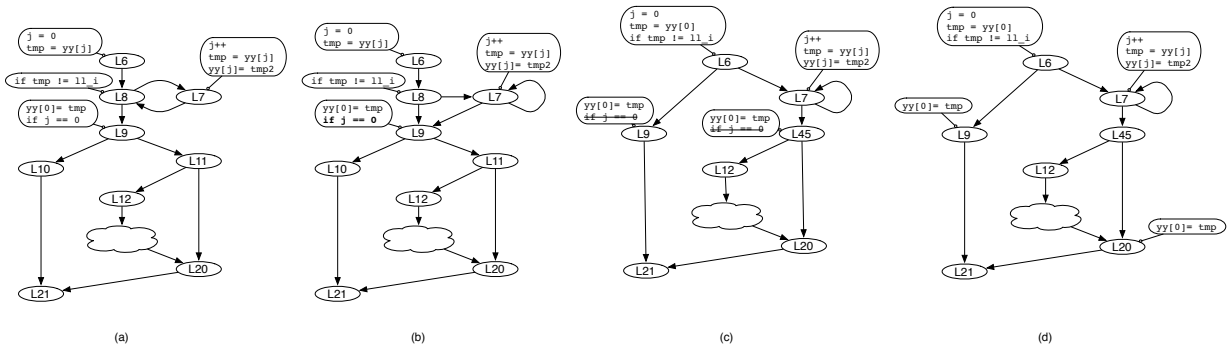


Figure 6: Interaction of structural techniques and traditional optimizations

byte loads and stores to the character array. In certain circumstances, scheduling two stores to nearby addresses within a couple cycles of one another can trigger an L1D stall on the Itanium 2.

This stall is due to a quirk in the design of the L1D cache on the Itanium 2. Although the processor advertises two store ports, in reality, the L1D cache is only pseudo-dual ported for stores [4]. Cache lines are split into eight single ported banks. Store coalescing hardware helps mitigate the penalty that would otherwise be associated with scheduling multiple stores to sequential addresses. However, if two stores that cannot be coalesced attempt to access the same bank, the younger one will be forced to stall. This appears in the benchmark compiled by GCC mainline. The store to `yy[0]` in L9 is scheduled one cycle after the store to `yy[j]` inside the loop. When the loop exits, the store in L9 may stall if `yy[0]` uses the same bank as `yy[j]`. Structural compilation transforms the CFG enough that the store can be moved many cycles later in the schedule, eliminating this stall.

Figure 6 shows the evolution of `generateMTFValues` as it is processed by the optimizers. We have annotated significant lines of code onto their corresponding CFG nodes. Figure 6(a) shows the core of `generateMTFValues` immediately after SSA form is constructed. Blocks L7 and L8 form the rotate loop, which cycles zero or more times. As the loop exits, L7 writes the `j`th element of the array. One cycle later, L9 writes into element 0 of the array and branches based on the value of `j`. If `j` is 0, L10 falls through to the rest of the function. Otherwise, L11 leads into a complex block of code.

Figure 6(b) shows the function after Superblock formation. Tail duplication copied the header of the L7-L8 loop so that L7 now forms a single block loop. We now have two flows into L9. Along the L8-L9 arc, `j` will always be 0. Along L7-L9, `j` will be non-zero. The original purpose of L9 was to determine whether the loop iterates, and by duplicating L8, we have made L9 redundant.

Figure 6(c) shows the kernel after constant- and value range propagation (VRP). VRP duplicates block L9 to make L45. VRP then propagates the value of `j` from L6 to L9 and from L7 to L45. This, in turn, allows for the elimination of the `if` statements in L9 and L45. None of the code in the subgraph headed by L12 uses the array `yy`, so in part (d), store sinking is able to move the write to `yy[0]` to L20. When this code is finally scheduled, the write to `yy[0]` doesn't occur until at least 12 cycles after the loop body, eliminating the stall.

5 Future work

Fully implementing the structural compilation model in GCC will be more a matter of tuning pieces already written than writing new code. At this point, Tree-SSA optimizers should generally be capable of moving instructions across basic blocks. There are already several structural-style passes written at the Tree-SSA level, such as the loop unroller. Study into moving these types of passes forward in the compilation order would be worthwhile.

Parameter and heuristic tuning is a matter that still needs to be addressed. Many of GCC's parameters governing code expansion are set conservatively relative to OpenIMPACT. If code expansion is done early enough, these can be set quite aggressively without adversely affecting the instruction cache performance of the generated code. Heuristics in the loop unroller refuse to unroll loops containing control flow for fear of increasing the number of branch mispredicts. Early unrolling combined with predication support may make unrolling such loops profitable, and so heuristics like these should be revisited.

Other code expanding transforms, such as branch target expansion, could be easily implemented within GCC. It would also be useful to investigate running multiple rounds of expansion. A Superblock-unroll-Superblock sequence could potentially give a very nice straight code sequence with high levels of ILP.

6 Conclusion

Although it has not yet demonstrated an overall performance improvement for Itanium, the Tree-SSA Superblock formation pass holds promise. We can already see an improvement in certain benchmarks, such as 186.crafty and 256.bzip2 due to the structural compilation model. Implementing additional early structural transformation passes will give optimizers more freedom to move and simplify program code. At the same time, we must ensure that the optimization and analysis passes can accept and use the modified control flow. When the transformation and optimization stages are fully compatible, we expect to replicate the consistent, positive performance results from the OpenIMPACT compiler.

7 Acknowledgments

We would like to thank the members of the OpenIMPACT research group and the GCC community for their help in this work. We would also like to thank the Gelato federation for its support of this project.

References

[1] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[2] Jan Hubička. Profile driven optimizations in GCC. *GCC Summit*, 2005.

[3] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

[4] Intel Corporation. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization, Document Number 251110-003*, May 2004.

[5] J. W. Sias, S.-Z. Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. W. Hwu. Field testing IMPACT EPIC research results in Itanium 2. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 26–39, June 2004.

[6] UIUC OpenIMPACT Effort. The OpenIMPACT IA-64 Compiler. `http://gelato.uiuc.edu/`.

Matrix flattening and transposing in GCC

Razya Ladelsky
IBM Haifa Labs
razya@il.ibm.com

Abstract

The layout of data in memory can have a significant effect on the performance of applications. Several compilation techniques can be used to optimize this layout. This paper describes two such optimizations: the first is matrix flattening, whose purpose is replacing a m-dimensional matrix with its equivalent n-dimensional matrix, where n<m. The frequent case is when a multidimensional matrix is flattened to its equivalent one dimensional matrix. This reduces the level of indirection needed for accessing the elements of the matrix. The second optimization is matrix transposing, which swaps rows and columns, and by doing so improves cache locality.

Both optimizations are interprocedural, and use the TREE-SSA based interprocedural framework, currently on ipa branch. In this paper we describe the algorithms used, as well as implementation issues. Preliminary results show substantial improvements for some floating point benchmarks, and no degradation for others. Finally we discuss the current status, future work and potential extensions.

1 Matrix reordering—Motivation

In any processor that contains some sort of memory hierarchy, access to a close location is faster than to farther ones. Achieving high performance requires effective use of cached data, meaning cache locality should be exploited as much as possible. One way to achieve locality is to transform loop nests. There has been a lot of work in the area of loop transformations. Among the techniques used are unimodular and nonunimodular iteration space transformations, tiling, loop fusion, and affinity scheduling. These techniques focus on changing the iteration space traversal order, and by doing so, they indirectly improve cache locality. Data transformation, however, focus directly on the data space, by changing the data layouts for better locality to be achieved. Unlike loop transformations, data transformations are not constrained by data dependencies, and can be applied to imperfectly nested loops. Also, while loop transformations affect all the matrices referenced in the loop nest, data transformations do not.

Not much work has been done in the area of data transformations in GCC. In this paper we present two matrix layout transformations. First we present matrix flattening, which flattens multi-dimensional dynamic matrices into contiguous memory space to achieve better reference locality. The second optimization also flattens multi dimensional dynamic matrices into one dimensional matrices, but reorders the elements of the flattened matrix differently, i.e. not corresponding to the original dimensional organization.

2 Flattening a matrix

Throughout this paper, we'll refer to the dimensions of matrices as inner/outer, lower/higher. For a matrix m[i1][i2]...[ik], we refer to i1 as the outer most dimension, and ik as the inner most. It is also convenient to number them. We'll number i1 as dimension 0, i2 as dimension 1, and so on. A lower dimension means an outer one.

2.1 Flattening—the idea

In order to explain the matrix flattening idea, we'll look at a two dimensional dynamically allocated matrix, `a` (as defined in C language). Let's denote the outer dimension as dimension 0, with size N, and the inner one as dimension 1, of size M. A typical creation of `a` will be:

```
a = (int**) malloc (N)
for (i=0; i<N; i++)
  a[i] = (int *) malloc (M);
```

Let's assume M=3 for simplicity of the example. The layout for this matrix organization is described in Figure 1:

```
a[0] = {x0, y0, z0}
a[1] = {x1, y1, z1}
...
a[N-1] = {xN-1, yN-1, zN-1}
```

Figure 1: two dimensional matrix example

Each access to an element of the matrix involves two levels of indirections. An access to a[i][j] requires accessing to a[i] in order to load the address of the inner dimension, and then accessing the j-th element of that dimension. Flattening the matrix `a` means that we allocate only one memory space (of size N*M), and place the elements contiguously in it. The matrix becomes one dimensional. An access to a[i][j] will be translated to accessing `base_of_new_allocated_matrix + new_offset`. The two references to memory that were required before, are replaced with only one memory access. If this access is in a loop, many memory references will be saved.

The resulting flattened matrix can be visualized as:

```
{x0 y0 z0 x1 y1 z1 ... xN-1 yN-1 zN-1}
{ dim 1 } { dim 1}     {    dim 1    }
{             dim 0                   }
```

An access to a[i][j] will now be an access to a[i*M+j].

We see that the elements are organized according to their original allocation, with the elements of the rows placed serially, and the rows placed one after the other in the order of iterating the outer dimension from 0 to N-1 (i.e. a[0], then a[1], and so on).

Flattening can be done for multiple dimension matrices whose dimension is greater than 2 as well. According to the same concept demonstrated for a 2 dimensional array, the elements of the inner most dimension are laid out serially. Iterating the outer dimension serially determines the order of the inner dimension, and so on, up to the outer most dimension. For example, given the the matrix in Figure 2:

The flattening can be visualized as:

{x0 x1 y0 y1 z0 z1 v0 v1 w0 w1 s0 s1}

We'll look at this example again in Section 6.4 that deals with transposing, and we'll see that there are alternative flattened layouts.

2.2 Partial flattening

In some cases the whole matrix cannot be flattened. This happens if some of its dimensions escape the application. Since flattening

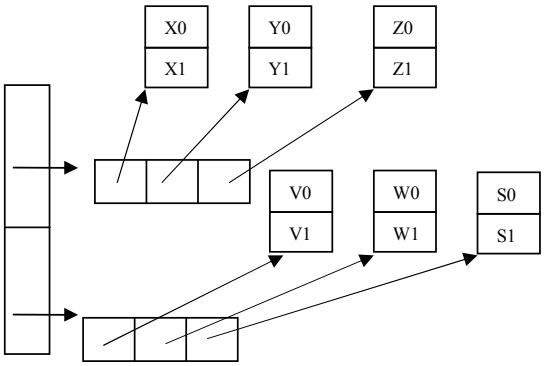


Figure 2: three dimensional matrix example

changes the definition of the matrix, we need to see all uses of the dimensions we flatten. Therefore, the escaped dimensions can't be flattened.

We'll present escape cases using an example of a three dimensional matrix, a[M][N][L]. The escape cases are:

1. The matrix, or part of it, is passed as an argument, e.g. call func(a[i][j]) or call func (&a[i][j][k]) causes the innermost dimension to escape.
2. part of the matrix is modified: a[i][j] = x; Again, the last (inner most) dimension escapes.
3. Multiple allocations for the same dimension. This is actually a private case of assigning a value to part of the matrix (case 2).

We call the outer most dimension that escapes, the `minimum indirection level`, which we'll also mark as `min_escape_level`, as any lower dimension (lower == outer) doesn't escape, and is safe for transformation, while any higher dimension is considered escaping. The dimensions from 0 (outer most) to the minimum indirection level are considered safe for transformation, because we know their behavior, while the rest should be left unchanged.

For the above escape examples of matrix `a`, `a`'s minimum indirection level is 2, meaning that dimensions 0,1 are safe for transformation and will be flattened, while the inner most dimension remains unchanged. Therefore, an access of a[i][j][k] to the old matrix, will be replaced by a[i*M+j][k] of the new partially flattened matrix.

3 Implementation Overview

Matrix flattening and transposing code has been developed in the ipa branch. Matrix flattening and transposing is one of the IPA passes, and is enabled by the `-fmatrix-flatten` flag. The option `-fipa-dump-mreorg` dumps information related to the optimization. The matrix flattening and transposing code can be found in the file `matrix-reorg.c`. The implementation is divided into three parts:

Analysis stage – performs a local analysis of the method to collect information about the allocation sites and the access sites for each matrix. the information is recorded in the mi structure (described below).

Decision making stage – decides whether to flatten the matrix, or the transposed matrix.

Transformation stage – changes the allocation and access sites in the various functions that access the matrix.

3.1 Data Structures

We need to collect a lot of data regarding the allocation and accesses of the original matrix. Everything is stored in three main structures:

`matrix_info` stores matrix information. It contains the following fields:

actual_dims – Maximum number of indirections used to access the matrix.

min_indirect_level_escape – Minimum indirection level that escapes. 0 means that the whole matrix escapes, k means that dimensions k to actual_dims escape.

malloc_for_level – Holds the allocation site for each level (dimension).

allocation_function_decl – The location of the allocation sites (all allocations must be in one function only)

free_info – The calls to free at each level of indirection.

dimension_size – An array holding the size for each dimension.

dim_hot_level – Array representing the hotness of each level, for transposing decision.

access_l – An array of the access sites of the matrix.

dim_map – A mapping from the old dimensions to their new order in the flattened matrix (also used for transposing)

`access_site_info` stores information about matrix access sites. It contains the following fields:

stmt – The access statement

offset – In case it is a `PLUS_EXPR`, this is the offset.

level – The level of indirection of this access statement.

iterated_by_inner_most_loop – Used for deciding whether to flatten the matrix or the transposed matrix.

`allocation_info` structure mainly contains *stmt*, which is the allocation statement.

3.2 Matrix reorg functions

The driver of matrix reorg is `matrix_reorg()`. The analysis part of matrix reorg is implemented by the following functions:

`analyze_matrix_allocation_site()` – Performs analysis of the allocation sites: recognizes the various dimensions' allocations and records them in the allocation site structures of the matrix.

`analyze_matrix_accesses()` – Recognizes and records the access sites of the matrix. By analyzing the accesses, it determines and records the minimum escape level and the actual dims of the matrix (the maximum level the matrix is accessed at).

`analyze_transpose()` – Decides whether to flatten the matrix as it is or flatten a transposed matrix. Determines the permutation of dimensions in which the matrix will be laid out.

The transformation part of the optimization is implemented by the functions:

`transform_allocation_sites()` – Replaces multiple mallocs with the equivalent malloc for the flattened matrix.

`transform_access_sites()` – Changes the access sites of the matrix to access the new flattened (possibly transposed) matrix.

4 Analysis phase

The analysis part of the optimization collects information about the allocation and access sites of the matrix. In the process, it determines K, the escape level of a N-dimensional matrix (K <= N), that allows flattening of the external dimensions 0, 1,..., K-1. An escape level of 0 means that the whole matrix escapes and no flattening is possible. The analysis phase is divided into analyzing the allocation sites and analyzing the access sites. We'll demonstrate the analysis methods using ssa representation. Both analyses are recursive. The two triggering calls from the analysis driver are as described below:

```
for (i=0; i < num_ssa_names; i++)
{
 ssa_var = ssa_name (i)
 stmt = DEF_STMT (ssa_var)
 if rhs is a matrix decl
   analyze_matrix_accesses
     (ssa_var, (level=)0)
 if lhs is a matrix decl
   analyze_allocation_site
     (DEF_STMT(ssa_var), (level=)0)
}
```

`analyze_allocation_site (stmt, level)` – Given a statement whose left hand side is a matrix variable, we traverse backwards to find the definitions that reach this variable, until we get to the allocation site (malloc, calloc, etc.) If we get some other kind of definition, we mark the matrix escaping. The patterns we are looking for are demonstrated in Figure 3.

```
analyze_allocation_site(stmt, level) {
 if (code (stmt) != MODIFY_EXPR)
   mark_matrix_escaping (level)

 rhs = get right hand side of stmt
 if (code (rhs) == SSA_NAME)
   analyze_matrix_allocation_site(DEF_STMT(rhs))
 else if (code (rhs) == CALL_EXPR) {
   if (call ! memory_allocation
       && call ! memory_free)
     mark_matrix_escaping (level)
   else
     add allocation site if there isn't
     a prior allocation statement at
     this level
 }
}

/* If needed, update the minimum escape
   level of the matrix.  */
mark_matrix_escaping (level) {
 if (matrix->min_escape_level > level)
   matrix->min_escape_level = level
}
```

Figure 3: Analyze Allocation Sites Algorithm

`analyze_accesses (ssa_var, level)` – Given a ssa var that is related to the matrix, and level of indirection corresponding to the current access level, we determine whether the matrix escapes at that level. If not, we record this access with the appropriate level. We follow the uses of the `ssa_var`, and analyze what happens to them. The handling of various use cases is described in Figure 4.

5 Transformation phase

In this phase we define the new flattened matrices that replace the original matrices in the code. We need to transform the allocation and the access sites of the matrix.

```
analyze_accesses(ssa_name, level) {
 use_stmt = use_stmt (ssa_var)
 switch (code(use_stmt)) {
  case PHI:
    /* The statement is of the form:
       res = PHI <ssa_name, ...>  */
    check escaping levels of arguments of PHI:
    if not all have the same level of escaping
      mark_matrix_escaping
           (minimum of escaping levels)
      return
    else
      analyze_matrix_accesses (res, level)

  case CALL_EXPR:
    /* The statement is of the form:
       ... = call (ssa_name, ...)  */
    if (call != alloc or free)
      mark_matrix_escaping (level)
    else
      record alloc or free, dim = level

  case MODIFY_EXPR:
    if ssa_name in lhs {
      /* The statement is of the form:
         *ssa_name = rhs */
      if rhs != ssa var
        mark_matrix_escaping (level)
      else
        /* Analyze the ssa var definition. */
        def_stmt = DEF_STMT (rhs)
        analyze_allocation_site
                  (def_stmt, level +1)
    } else ssa_name in rhs {
      /* Several optional patterns:  */

      /* Form is  ... = *ssa_name  */
      record access site with dim = level
      level ++
      goto acc

      /* Form is ... = call (ssa_name, ...) */
      if (call != alloc or free)
        mark_matrix_escaping (level)
      else
        record alloc or free site, dim = level

      /* Form is ... = ssa_name + ...  */
      record access site with dim = level

      /* Form is ... = ssa_name  */
acc:  if (lhs is ssa var)
        analyze_matrix_accesses(lhs, level)
      else
        mark_matrix_escaping (level)
    }
 }
}
```

Figure 4: Analyze accesses

5.1 Transforming allocation sites

Given an allocation function, which is the function that contains all of the matrix's memory allocations (an allocation for each dimension), we need to modify the code such that all memory allocations of dimensions that should be flattened, will be replaced with a single memory allocation of the equivalent size.

We calculate and produce code that reflects a new symbolic size for each dimension of the flattened (part of the) matrix. The new size symbolizes the over-all size of all elements contained within this dimension. Each new dimension size is placed in a new global variable, which we'll annotate as Ti, so it could be read from all functions that use the matrix (these values are used when changing the access sites of the matrix). The new size for dimension 0 holds the over-all memory size that should be allocated for the flattened part of the matrix. In the same manner of creating global variables for the new dimension sizes, we need to keep a global variable for each original size of the original dimension. We'll denote it T_ORIGi for dimension i, and it is used by the access sites to calculate the new offset.

When we start the transformation, we already have information about the matrix we want to transform. We collect this information in the analysis phase. This is the input for our algorithm. The minimum escape level has already been determined, and so has the original size for each dimension.

We'll define the following symbolic structures for each dimension i, where `i=0`, ..., `min_escape_level-1`: (`min_escape_level-1` is the inner most dimension of the matrix that should be flattened):

`dim_size_orig[i]` holds the original size of the dimension i.

```
transform_allocation_sites {
 for all i = (min_escape_level-1 to 0) {
  add_new_global_variable T_ORIGi
  emit code : T_ORIGi =  dim_size_orig[i]

  dim_size_orig [i] = Ti
 }
 prev = type_size[min_escape_level-1]
 for all i = (min_escape_level-1 to 0) {
  dim_num =
    dim_size_orig[i]
                / type_size[i]
  dim_size =
    prev * dim_num

  add_new_global_variable Ti
  emit code : Ti =  dim_size

  prev =  dim_size [i] = Ti
 }

 Remove all allocation statements for
 dimensions (1,...,min_escape_level-1)
 from the allocation function

 Change the allocation statement of level 0
 to allocate size according to T0.
}
```

Figure 5: Transform Allocation Sites Algorithm

`dim_size[i]` is initialized similarly to `dim_size_orig[i]`. This value changes throughout the algorithm to reflect the new symbolic dimension size.

`type_size[i]` holds the original size of the type of elements in dim i.

The algorithm is introduced in Figure 5.

The algorithm's output: for each dimension `i`, where `i=0,...,min_escape_level-1`, `dim_size[i]` and `Ti` contain the new symbolic dimension size for dimension `i`. `Ti` was inserted to the allocation function and can be read by other functions that access the matrix. All allocations of the dimensions that need to be flattened were replaced by a single equivalent memory allocation.

In other words, for a matrix of N dimensions, where we flatten dimensions 0 to D, the matrix's allocation sites:

```
malloc (dim (0))
malloc (dim (1))
 ...
malloc (dim (D))
malloc (dim (D+1))
...
malloc (N-1)
```

are transformed to

```
malloc (dim(0) * dim(1) * ...
        * dim (D) * size_type (D))
malloc(dim(D+1))
....
malloc (N-1)
```

The handling of free statements is simple: we remove all statements that free allocated memory for dimensions that we flattened, except for the free of the outer most flattened dimension.

5.2 Transform Access sites

Since the definition of the matrix has changed, it is obvious that the accesses to the matrix should be modified. We need to calculate the new offset from the base address of the new flattened matrix. For matrix `a` of k dimensions of sizes D1...Dk, and whose escape level is m<=k, we'll denote `a'` as the new allocated matrix. The original access a[I1][I2]...[Ik] will be translated to:

b[I(m+1)]...[Ik]

where

b = a' + I1*D2...*Dm + I2*D3...*Dm + ... + Im

In other words, access a[I1][I2]...[Ik] is translated to:

a'[I1*D2...*Dm + I2*D3*...*Dm + ...
+ Im][I(m+1)]...[Ik]

In the analysis phase, we determined the level of each access site, which is the dimension of the matrix accessed by this access site. We need to change only accesses whose level is lower (outer) than the matrix's escape level, because these are the accesses to the dimensions that were flattened.

The algorithm is introduced in Figure 6. We'll refer to the same structures described in the previous subsection.

```
transform_matrix_accesses {
 level = access -> level
 while (access in access list) {
   if (level <= min_scape_level)
     continue;
   if (access->stmt includes offset) {
     num = access->offset / type_size [level]
     dim_num =
       dim_size_orig[level] / type_size [level]

     new_offset =
       num *
         (dim_size[level] / dim_num)

     replace offset with new_offset
   }
   if (access->stmt includes dereferencing
       && acc_info->level < min_escape_level-1)
          remove the stmt from code
 }
}
```

Figure 6: Transform Accesses Algorithm

Notice that `dim_size[level]` was assigned with the global variable Tlevel and `dim_size_orig[level]` was assigned with the global variable `T_ORIGlevel`. (These global variables were defined in the allocation function, see `transform_allocation_sites` algorithm in the previous subsection).

The value `dim_size[level] / dim_num` represents the size of the dimension at `level + 1`.

The output of the `transform_accesses` algorithm is the code created for the new offsets and the removal of accesses to flattened dimensions.

6 Matrix Transpose

6.1 Transposing—the idea

If we look at the order in which we organized the flattened matrix, we notice that the elements of the inner most dimension were placed consecutively. For example, for a two dimensional matrix `a`, the elements of the array a[0] were placed one after the other, then the elements of a[1], and so on. This could produce good cache behavior if the elements of the inner dimension are accessed sequentially by the program. For example:

```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    access to a[i][j]
```

However, if the accesses are of the following form:

```
for (i=0; i<N; i++)
 for (j=0; j<M; j++)
   access to a[j][i]
```

The outer dimension is iterated sequentially. We basically iterate the columns and not the rows. Therefore, if we placed the elements of the columns serially, and the columns one after the other, we would have had an organization with better (cache) locality. Another way of looking at it is that if the matrix a was organized in a column-major fashion, we would have had better data locality. Flattening this column-major matrix could produce even better performance.

In order to achieve this effect, we conceptually transpose the matrix `a` (i.e. swap the columns and rows), and then flatten it. For example, the row-major matrix `a` in Figure 1 was flattened in Section 2.1. Now we show the flattened `transposed` matrix:

```
{x0 x1...xN-1 y0 y1...yN-1 z0 z1...zN-1}
{dimension 0} {dimension 0}{dimension 0}
{           dimension   1              }
```

dimension 0 was originally the outer dimension, and dimension 1 was the inner one. When organizing the transposed layout, dimension 0 is treated as the inner dimension, and dimension 1 as the outer one.

This can be enhanced for multiple dimensioned matrices as well (the transformation for a three dimensional matrix is exemplified in the further subsections).

6.2 Transposing - transformation phase

The decision making phase explained in the next subsection, determines whether to flatten the `transposed` matrix. It then supplies a mapping of the dimensions, which we'll denote dim_map. It is a function that maps the dimensions according to their new order in the flattened matrix. It is basically a permutation of the dimensions. The mapping for the example we saw in the above subsection, will be:

```
dim_map[0] = 1
dim_map[1] = 0
```

For dimensions i=0,1,...,k, dim_map[k] represents the dimension that will be treated as the innermost dimension, dim_map[0] represents the outermost.

The transformation part of matrix flattening which was shown in Figure 5 was enhanced to work with this dim_map, in order to enable flattening of transposed matrices as well.

```
transform_allocation_sites {
 for all i = (min_escape_level-1 to 0) {
   add_new_global_variable T_ORIGi
   emit code : T_ORIGi =  dim_size_orig[i]

   dim_size_orig [i] = Ti
 }
 prev = type_size[min_escape_level-1]

 for all i = (min_escape_level-1 to 0) {
   dim_num =
     dim_size_orig[dim_map[i]]
                 / type_size[dim_map[i]]
   dim_size =
     prev * dim_num

   add_new_global_variable Ti
   emit code : Ti =  dim_size

   prev =  dim_size [dim_map[i]] = Ti
 }

 Remove all allocation statements for
 dimensions (1,...,min_escape_level-1)
 from the allocation function

 Change the allocation statement of level 0
 to allocate size according to T0.
}
```

Figure 7: Transform Allocation Sites - complete algorithm

For flattening the matrix without transposing, the mapping is simply the identity function. Note that only the allocation sites transformation needs to interact with the dim_map. Once it assigns the structure dim_size (which holds the size of each dimension in the new flattened organization), the access sites transformation already looks at updated dimension sizes structure, which actually identifies how the allocation is done.

The algorithm flattens the matrix while organizing the elements in the order determined by the dim_map. The algorithm is demonstrated in Figure 7.

6.3 Transposing—decision-making phase

As we've seen, the profitability of flattening the transposed matrix depends on the accesses to

the matrix. If all, or most accesses to the matrix are of the form

```
for (i=0; i<N; i++)
 for (j=0; j<M; j++)
   access to a[j][i]
```

then it would be profitable to flatten the transposed matrix. However, if all or most accesses are of the from

```
for (i=0; i<N; i++)
 for (j=0; j<M; j++)
   access to a[i][j]
```

then it would be more valuable to flatten the matrix as it is. In order to calculate the profitability, we collect two types of information regarding the accesses:
1. profiling information used to express the hotness of an access, that is how often the matrix is accessed by this access site (count of the access site).
2. which dimension in the access site is iterated by the inner most loop containing this access.

The matrix will have a calculated value of weighted hotness for each dimension. Intuitively the hotness level of a dimension is a function of how many times it was the most frequently accessed dimension in the highly executed access sites of this matrix.

As computed by following equation:
m n
$\sum\sum dim\_hot\_level[i] +=$
j i
acc[j]->dim[i]->iter_by_inner_loop * count(j)

Where n is the number of dims and m is the number of the matrix access sites.

The organization of the new matrix should be according to the hotness of each dimension. The hotness of the dimension implies the locality of the elements.

6.4 Flattening transposed matrix - example

Let's look at the three dimensional matrix in Figure 2:

Let's assume that there are two access sites to this matrix.

```
access site 1:
 for (iterate i)
  for (iterate j)
    for (iterate k)
      mat [i][j][k]

access site 2:
 for (iterate i)
  for (iterate j)
    for (iterate k)
      mat [k][i][j]
```

and `count(access site 2) > count(access site 1).` According to the decision making function, we have:

access site 1: the inner most dimension (dimension 2) is iterated in the inner most loop,

dim2->iter_by_inner_loop == 1 .
access_site 1 -> dim_hot_level [0] = 0
access_site 1 -> dim_hot_level [1] = 0
access_site 1 -> dim_hot_level [2] = count (access_site 1)

access site 2: dimension 1 is iterated in the inner most loop
access_site 2 -> dim_hot_level [0] = 0
access_site 2 -> dim_hot_level [1] = count (access_site 2)
access_site 2 -> dim_hot_level [2] = 0

Summing up the two access sites produces:
dim_hot_level [0] = 0
dim_hot_level [1] = count (access_site 2)
dim_hot_level [2] = count (access_site 1)

Since count(access site 2) > count(access site 1), the hottest dimension is dimension 1, then dimension 2 and lastly, dimension 0.

The resulting dim_map will be:
dim_map [0] = dimension_0
dim_map [1] = dimension_2
dim_map [2] = dimension_1

and the resulting flattened matrix:

{x0 y0 z0 x1 y1 z1 v0 w0 s0 v1 w1 s1 }

7 Issues

We'll discuss a few algorithmic and design issues related to matrix flattening and transposing optimizations.

7.1 Alignment issue

Transforming the allocation of the matrix might cause a misalignment issue that did not exist before the flattening. For example, flattening a 2 dimensional matrix according to the algorithm previously introduced, results in placing row after row sequentially. Originally these rows were created by separate memory allocations, which guaranteed their alignment. For optimizations that follow matrix flattening, there is no correctness problem because they already "see" the new flattened matrix. However, problems may occur for optimizations that precede matrix flattening. One such example is manual vectorization. Vector operations may take into account that the rows are aligned. The new matrix layout does not maintain the alignment for the rows anymore. This may be problematic and cause crashes. There are several possible solutions. One is padding between the rows. Another is to disable the optimizations if such vector operations exist.

We chose not to insert padding for two reasons: first is that it could effect cache behavior, and secondly is that it gets very complicated when we flatten a transposed matrix.

Therefore, the matrix flattening optimization should be disabled when vector operations exist. We disable the optimization if any of the following exist in the code:

1. convert expression of vector type.
2. call to builtin function which gets the matrix (or part of it) as an argument.
3. ASM operations.

7.2 Fortran matrices

The matrices in fortran are already flattened by the front end. Therefore matrix flattening is useless for Fortran matrices. However, transposing can be useful for these matrices. The flattened matrices need to be recognized and then reordered in a more optimal organization. This is currently not handled.

7.3 Whole program

Matrix flattening requires whole program view, as we change the definition of the matrices, and therefore all uses must be seen by the analysis and transformation. Therefore, we apply the optimization only if whole program flag is enabled.

8 Status

Matrix flattening code was submitted to ipa branch in March 2006. It includes the capability to flatten dynamically allocated non-escaping C matrices. The code produced improvements of 35% on art and 9% on equake (tested on linux powerpc).

The code for transposing is currently being developed, and preliminary results show a 200% improvement for transposing the matrices in art. The decision making phase has not been completed. It includes using the profiling information and also gathering the information about which dimension was iterated by the inner most loop at each access. This work will be submitted shortly.

Currently the code handles only dynamically allocated C matrices. Future enhancements include enhancing this work for other types of matrices: statically defined matrices, Fortran matrices, and so on. Future work also includes extending the patterns recognized by the analysis phase, and also enhancing the reordering algorithms for various matrix layouts.

9 Acknowledgements

I would like to thank Revital Eres and Mustafa Hagog, who wrote the preliminary versions of matrix flattening code, which I continued developing.

I would like to thank Jan Hubicka, Daniel Berlin and Sebastisn Pop who advised with design and implementation issues.

I would like to thank Peter Bergner for reviewing this paper, the IBM Haifa team for helpful discussions, and all GCC contributors who offered help and comments.

References

[1] M. Cierniak, W. Li. *Unifying data and control transformations for distributed shared memory machines*. ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.

[2] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. *A Matrix-Based Approach to Global Locality Optimization* . Parallel Architectures & Compilation Techniques (PACT'98).

[3] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, J. Ramanujam. *A Data Layout Optimization Technique Based on Hyperplanes* . Proceedings of the Eleventh International Parallel Processing Symposium, 1997.

[4] M. Kandemir, A. Choudhary, J. Ramanujam, N. Shenoy, P. Banerjee *Enhancing Spatial Locality Via Data Layout Optimizations* . In Workshop on Automatic Parallelisation, Southhampton, UK, Sept. 1998.

[5] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[6] Ken Kennedy, Randy Allen *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann; 1st edition (October 22, 2001)

A report on the progress of GNU Modula-2 and its potential integration into GCC

Gaius Mulley
University of Glamorgan
gaius@glam.ac.uk

Abstract

This paper reports on the status of the GNU Modula-2 front end to GCC and the extensions made to Modula-2 and gdb to ease its potential integration into the main GCC source tree. GNU Modula-2 (`gm2`) is maturing into a reliable tool and it now builds and passes its regression tests on the following platforms: x86, Opteron, Athlon 64, Alpha, Itanium processors running GNU/Linux, Sparc based Solaris, PowerPC MacOS, x86 Open Darwin and the x86 processor running FreeBSD. GNU Modula-2 currently conforms to all three Programming in Modula-2 dialects as defined by Wirth.

The paper also describes the two categories of language extensions made. The first category follows the tradition of other GCC front ends by allowing the in-lining of assembly language, conditional compilation, procedure inlining and allowing users to cleanly exploit the GCC library of built-in functions and constants. The second category provide easy access to C libraries. The work presented here discusses the portable implementation of open arrays, module priorities, coroutine primitives and multi-word sets. It also reports on many of the key design decisions taken during the construction of GNU Modula-2 and their various implications.

1 Introduction

Source code which cannot port to another architecture will die [Gancarz]. The motivation for producing a Modula-2 front end for GCC includes providing a robust compiler for production systems, providing a migration path for legacy source code and producing a compiler which can generate enhanced semantic error messages for student programmers.

Modula-2 is a relatively small language in contrast to C++ and Ada yet its flexibility with low level processes, bit manipulation, interrupt handling make it an ideal language with which to implement small footprint embedded systems. Its other strengths are an enforced modularity, the abstract data type and its distinctive definition and implementation module files. It also found favour amongst many academic undergraduate teaching programs during the 1980s and 1990s. Some eminent academics argue that a language, of a similar simplicity to Modula-2, is ideal for teaching new students to program. [1]

Currently there are only a few commercial Modula-2 compilers being actively maintained. Code which was written ten or fifteen years ago may still be compiled by older commercial (possibly unmaintained) Modula-2 compilers,

[1] `http://www.iticse2002.dk/conference/Talk/iticse2002.html`

however a number of these compilers generate 16 bit code. While the 32 bit x86 processors remain, compilers targeting these processors may be run in compatibility mode. Time is running out as the computing industry is switching to 64 bit microprocessors [AMD64] [Opteron] [Intel1] [IBM970]. While x86 emulation, 16 bit backwards compatibility or running 32 bit code on a 64 bit platform are all possible they all have serious drawbacks. In order for the older source to be compiled into a native executable it will either have to be translated into another high level language or alternatively a Modula-2 compiler which can target these new generation of microprocessors will have to be acquired. GNU Modula-2 suits this purpose as it has the advantage of being closely tied to GCC. Not only does this produce excellent code and good architectural and operating system coverage but it also utilises many of the GCC features. For example GNU Modula-2 can: invoke the C preprocessor to manage conditional compilation; in-line `SYSTEM` procedures, intrinsic functions and memory copying routines; provide access to assembly language using the GCC syntax.

GNU Modula-2 currently supports all dialects of *Programming in Modula-2* as defined by Wirth [Wirth1] [Wirth2] [Wirth3] and work is underway to support the ISO dialect [ISO]. It enhances a number of language features, for example: it allows sets to be declared of any ordinal type; abstract data types are not restricted to a pointer type in the implementation module; constants, types, variables may be declared in any order. The compiler provides numerous command line options which: enable runtime and compile checking, specific optimisations, runtime behaviour of `DIV` and `MOD` operators as well as library dialect and various linking options.

Given that the original C compiler has become the GNU compiler collection it is fitting that a Modula-2 front end should exist.

2 Previous work

There was a previous GNU Modula-2 effort undertaken by the computer science department at the State University of New York at Buffalo [Bowen]. A substantial amount of work was achieved but funding terminated before the compiler was complete. In particular Modula-2 support was added to the GNU debugger `gdb`. The compiler had an elegant method of interfacing to C through the use of the keywords `DEFINITION MODULE FOR "C"` in the definition module. This front end matched GCC release 2.3.3 in 1994. Since then the GCC internals have changed substantially to incorporate a different paradigm of garbage collection.

Previously at the University of Glamorgan a Modula-2 compiler (`m2f`) was produced which performed detailed semantic checks and informative error messages [Mulley] [Lewis]. A number of these checks were performed post intermediate code optimisation in an attempt to maximise the knowledge the compiler had about the program source. This resulted in the compiler having the ability to detect elementary infinite loops.

3 Goals of GNU Modula-2

GNU Modula-2 will support the language as defined by Wirth [Wirth1] [Wirth2] [Wirth3] in “Programming in Modula-2” (PIM) and also the ISO Modula-2 standard [ISO]. The GNU Modula-2 project has opted for a release early strategy [Raymond] and it will initially support the PIM dialect of the language before implementing the ISO standard. The differences between the last three PIM editions are so small that they can all be incorporated into one compiler and they may be individually selected by command line options.

The GNU Modula-2 implementation must respect the fact that Modula-2 allows programmers to declare types, variables, constants and procedures in any order. Also the compiler must not include any artificial programming limits [Pronk] and therefore the implementation must avoid fixed array sizes and use dynamic data structures throughout [Stallman2].

Given that modern software projects are unlikely to be completely written in Modula-2 and a large target audience of `gm2` will be maintainers of legacy software it is vital to include good access to other languages. This is crucial since it is a core aim that `gm2` will be one component of the GNU compiler collection. A clean interface between Modula-2 and C also aids the development of `gm2`. There will be two types of extensions to Modula-2 the first facilitates access to other languages and the second provides features found in the other GNU compilers.

4 Extensions to Modula-2

There is no mechanism to manage conditional compilation in any of the four language specifications of Modula-2 mentioned earlier. The GNU implementation of Modula-2 allows the C preprocessor to be invoked when the option `-Wcpp` is present on the command line. This option tells the C preprocessor to operate in traditional mode, using assembler as a base language and it preserves comments and pays no attention to single quote or double quote characters. Thus macro argument symbols are replaced by the argument values even when they appear within string or character constants. It also ensures that the symbols `#` and `##` have no special meaning. This strategy also matches the behaviour of another GNU compiler front end (namely `f77`).

GNU Modula-2 allows local procedures to be compiled in-line. Currently the programmer has to declare a procedure using the keyword sequence `PROCEDURE __INLINE__`. Again this matches the GNU C compiler. If a procedure is exported then the procedure is still inlined locally and a copy of the code is placed in the object file to resolve external references.

The `__BUILTIN__` keyword occasionally appears in a definition module. In this case the procedure is implemented internally within the compiler back end. This allows the compiler to utilise its library of optimal routines without changing an applications import list. For example `memcpy`, `alloca` can be exported from the library `libc` and `sin`, `cos`, `log2` are exported from `MathLib0`.

Figure 1 contains the complete PIM compliant `MathLib0` definition module which defines many standard mathematical functions and two constants.

By examining the definition module the user can immediately determine which of the functions will be in-lined if the appropriate optimisation flags are present on the `gm2` command line. In the implementation module the mapping between PIM function names and the GCC back end functions are stated. Figure 2 contains the first 30 lines of the `MathLib0` implementation module.

In this implementation module the keywords `__ATTRIBUTE__` `__BUILTIN__` are used to denote the mapping between the internal GCC function name and the equivalent Modula-2 function. It is also worth noting that this module imports `cbuiltin` and `libm`. The module `cbuiltin` declares all GCC built-in functions whereas the module `libm` provides access to the C library `libm.a`. The above implementation module is constructed by calling upon `cbuiltin` functions wherever possible and only falling back upon the services of `libm` when no built-in is available.

```
DEFINITION MODULE MathLib0 ;

CONST

pi  =3.14159265358979323846264338327950288419 72;
exp1=2.71828182845904523536028747135266249775 72;

PROCEDURE __BUILTIN__ sqrt (x: REAL) : REAL ;

PROCEDURE __BUILTIN__ sqrtl (x: LONGREAL) :
                      LONGREAL ;

PROCEDURE __BUILTIN__ sqrts (x: SHORTREAL) :
                      SHORTREAL ;

PROCEDURE exp (x: REAL) : REAL ;
PROCEDURE exps (x: SHORTREAL) : SHORTREAL ;

PROCEDURE ln (x: REAL) : REAL ;
PROCEDURE lns (x: SHORTREAL) : SHORTREAL ;

PROCEDURE __BUILTIN__ sin (x: REAL) : REAL ;

PROCEDURE __BUILTIN__ sinl (x: LONGREAL) :
                      LONGREAL ;

PROCEDURE __BUILTIN__ sins (x: SHORTREAL) :
                      SHORTREAL ;

PROCEDURE __BUILTIN__ cos (x: REAL) : REAL ;
PROCEDURE __BUILTIN__ cosl (x: LONGREAL) :
                      LONGREAL ;
PROCEDURE __BUILTIN__ coss (x: SHORTREAL) :
                      SHORTREAL ;

PROCEDURE tan (x: REAL) : REAL ;
PROCEDURE tans (x: SHORTREAL) : SHORTREAL ;

PROCEDURE arctan (x: REAL) : REAL ;
PROCEDURE arctans (x: SHORTREAL) : SHORTREAL ;

PROCEDURE entier (x: REAL) : INTEGER ;
PROCEDURE entiers (x: SHORTREAL) : INTEGER ;

END MathLib0.
```

Figure 1: PIM compliant Mathlib0 definition module

In keeping with other GNU compilers `gm2` allows in-line assembly language statements through the `ASM VOLATILE` keywords. The `ASM` statement in `gm2` is an extension to the statement sequence EBNF rule found in the PIM appendices [Wirth1] [Wirth2] [Wirth3]. These follow the method outlined in the GCC manual [Stallman1]. For example on the Pentium[Intel2] the following function adds the two `CARDINAL`s `i` and `j` together and places the result in `k`.

```
ASM VOLATILE ("movl %1,%%eax; \
               addl %2,%%eax; movl %%eax,%0"
      : "=g" (k)           (* outputs *)
      : "g" (i), "g" (j)   (* inputs  *)
      : "eax") ;           (* we trash *)
```

The `VOLATILE` keyword indicates that the instruction has important side effects and the back end is told not to reschedule other instructions across it. The `"g"` informs the back end that the following expression requires an integer register (`"f"` indicates that a floating point register is required). The output integer variable `k` must have an operand string `"=g"`. Finally the back end is told that the `eax` register is destroyed.

Interfacing to other languages is performed by using a language specific definition. The keywords `DEFINITION MODULE FOR "C"` indicate

```
IMPLEMENTATION MODULE MathLib0 ;

IMPORT cbuiltin, libm ;

PROCEDURE __ATTRIBUTE__ __BUILTIN__
          ((__builtin_sqrt))
          sqrt (x: REAL): REAL;
BEGIN
   RETURN cbuiltin.sqrt (x)
END sqrt ;

PROCEDURE __ATTRIBUTE__ __BUILTIN__
          ((__builtin_sqrtl))
          sqrtl (x: LONGREAL): LONGREAL;
BEGIN
   RETURN cbuiltin.sqrtl (x)
END sqrtl ;

PROCEDURE __ATTRIBUTE__ __BUILTIN__
          ((__builtin_sqrts))
          sqrts (x: SHORTREAL) : SHORTREAL ;
BEGIN
   RETURN cbuiltin.sqrtf (x)
END sqrts ;

PROCEDURE exp (x: REAL) : REAL ;
BEGIN
   RETURN libm.exp (x)
END exp ;
```

Figure 2: Section of Mathlib0 implementation module

the implementation module is written in C. It also causes parameters in all exported procedures to be adjusted to match the C calling convention. The example below shows how access to the `libc` function `printf` is achieved. The first parameter `a: ARRAY OF CHAR` will be mapped onto `char *` but will be type compatible with `ARRAY OF CHAR`, all subsequent arguments will be promoted to the Modula-2 type `SYSTEM.WORD`.

```
DEFINITION MODULE FOR "C" libc ;

EXPORT UNQUALIFIED printf ;

PROCEDURE printf (a: ARRAY OF CHAR; ...) ;

END libc.
```

5 Structure of the GNU C compiler

The internal details of the latest release of the GNU C compiler are well documented in [Stallman1] and older versions in [Pizka] and [Granlund].

Until the introduction of GIMPLE and GENERIC into GCC 4.0 the C compiler could be considered as a one pass compiler with four phases. The first phase parses input source and builds a tree structure describing the program's behaviour. This is then manipulated by the second phase to produce a register transfer language (RTL) description. The RTL is a lisp like generic assembly language and this is heavily optimised by the third phase before being transformed into target assembly language by the fourth phase.

It is the duty of the first phase, the compiler front end, to resolve all types, check the correctness of the declarations and enforce the language rules.

5.1 GNU Modula-2 configuration compliance with the GNU C compiler

The file structure of a GCC front end is expected to contain certain key configuration files together with the source code. GNU Modula-2 adheres to this structure and it can be grafted onto GCC 3.3.6 in the subdirectory `gcc-3.3.6/gcc`. It includes the following configuration files:

- `Make-lang.in` defines the high level rules for building the front end. Typically these include rules to build the compiler driver, in this case the command line tool `gm2`, and the rules to build the `info` files, support tools and different compiler generations.

- `Makefile.in` is only used by the maintainers to create GM2 release snapshots.

- `lang-options.h` defines the GNU Modula-2 language specific options. These include: range checks, semantic checking options, dialect options, optimisation, library and linking options.

- `config-lang.in` describes the executables which will be built and a list of `Makefile`s which will be automatically created [MacKenzie] by `./configure` in the top level directory.

- `lang-specs.h` defines all the command line options which are legal in the front end. It also determines how and which support tools will be invoked. In GNU Modula-2 the C preprocessor can be invoked by `-Wcpp`. The specialist `gm2` specific linking options are also defined in this file. The `-Wmakeall` command line option will compile the current module and all dependents and perform the final link.

The main data type used in the interface between front end and the GCC back end is the `tree`. Trees are used to represent constants, types, variables, procedures and all statements. They maybe chained together to represent parameter lists, sequences of record fields or an enumerated data type. The `tree` is implemented in C as a pseudo abstract data type. In the implementation of GNU Modula-2 front end (itself written in Modula-2 and C) this type is presented as an abstract data type. There exists a definition module `gccgm2.def` which provides a functional interface to a wide range of `tree` operators. A corresponding `gccgm2.c` implements this specification. This works extremely well in practice as the separation and purpose of the Modula-2 and C components are clear.

6 GNU Modula-2 compiler options

The GNU Modula-2 compiler options are fully documented in the `texinfo` based manual [GM2]. This section describes some of the more interesting compiler options. The compiler provides extensive runtime checking through the `-Wbounds`, `-Wreturn`, `-Wnil`, `-Wcase` options, which: check array bounds, functions execute a `RETURN` statement, pointers do not dereference through `NIL` and all case expression values are tested.

The compiler can be told to compile PIM2, PIM3, PIM4 dialect Modula-2 via the `-Wpim2`, `-Wpim3` and `-Wpim4` switches respectively. The `-Wiso` is only partially complete, and it currently gives access to the ISO SYSTEM module and modifies the library path to include the ISO libraries.

The options `-Wextended-opaque` and `-Wcpp` enable abstract data types to be implemented by any type and preprocess all source code with the C preprocessor respectively.

One expected category of users are students learning to program. The option `-Wstudents` checks for bad programming style and it checks whether variables of the same name are declared in different scopes and whether variables look like keywords [Lewis]. The `-Wpedantic` option forces the compiler to reject nested `WITH` statements referencing the same record type and does not allow multiple imports of the same item from a module. It also checks that: procedure variables are written to before being read; variables are not only written to but read from; variables are declared and used. It also checks to see that `FOR` loop indices are not used outside the end of a loop without being reset.

The `-Wpedantic-param-names` ensures that procedure parameter names are the same in the definition module and in the implementation module counterpart. This is not necessary in ISO or PIM versions of Modula-2, but it can be extremely useful, as long as code is intentionally written in this way.

Lastly the `-funbounded-by-reference` option enables optimisation of unbounded parameters by attempting to pass non `VAR` unbounded parameters by reference. This optimisation avoids the implicit copy inside the callee procedure. GNU Modula-2 will only allow unbounded parameters to be passed by reference if, inside the callee procedure, they are not written to, no address is calculated on the array and it is not passed as a `VAR` parameter. Note that it is possible to write code to break this optimisation, therefore this option should be used carefully. For example it would be possible to take the address of an array, pass the address and the array to a procedure, read from the array in the procedure and write to the location using the address parameter. Due to the dangerous nature of this option it is not enabled when the `-O` option is specified.

7 Structure of the GNU Modula-2 front end

The language Modula-2 allows declarations to occur in any order and GNU Modula-2 allows an abstract data type to be implemented as any type (not restricted to a pointer type). The Modula-2 import and export rules together with the out of order declaration naturally lends itself to using a multi-pass approach to compilation.

The GNU Modula-2 compiler uses a `flex` built lexical analysis phase to build a dynamic buffer for all source tokens. This is then parsed twice to resolve enumerated types, exports, abstract data types and all the forward declarations. It is parsed for a third time to produce quadruple intermediate code. At this point the quadruples are optionally optimised and semantically checked before being converted into `tree`s and passed to the GCC backend.

The front end symbol table contains a `tree` field for each table entry. This is necessary to implement the optimisation phases in the front end and which provide extra knowledge for semantic analysis. At this point constants and literals will have their `tree` fields initialised in the front end symbol table. Once all the checking has been performed the remaining front end symbol are converted into `tree`s. This technique works well as the front end handles all the backward declarations and it is only when the front end symbol table is entirely complete that many of the type `tree`s are created. This makes the interface to the back end simpler and much easier to debug. The interface routines can ignore many of the error nodes which are only created by the back end when the input source is illegal.

8 Bootstrapping GNU Modula-2

The GNU Modula-2 front end source tree includes the Modula-2 source code for the compiler and libraries as well as a modified version of `p2c`. The modifications to `p2c` allow it to translate the Modula-2 front end component into C. The main changes were: to implement the PIM2 dialect of Modula-2, implement `BITSET` set types and abstract data types.

When building GNU Modula-2 natively the `make gm2.paranoid` test may be performed, this proceeds to compile the Modula-2 sources into object form using the previous version of the compiler. These objects and the GCC back end are linked to form a second generation compiler. The second generation of the compiler is used to create a third generation of the compiler and finally all three compilers are requested to produce assembly language files for all Modula-2 sources. These assembly language files are then `diff`ed to check that the compiler is completely stable. This paranoid test has proven very worthwhile during the development cycle. It also gives a high degree of confidence that the second generation of the compiler is behaving in exactly the same way as the first generation of the compiler and therefore it can be debugged using `gdb` against the original source code (rather than the translated intermediate C code of the first generation compiler).

9 Implementing open arrays using trees

The `tree`s that the back end provide are used right at the start of the compilation process. Initially the back end creates key base types such as `integer_type_node`, `char_type_node` and constants of zero and one. The Modula-2

front end continues to create language specific types such as `BOOLEAN` and initialises trees for any built-in functions that the back end offers. GNU Modula-2 obtains a reference to `memcpy` and `alloca` in `gccgm2.c`. [2]

```
tree gm2_memcpy_node
       = builtin_function
         ("__builtin_memcpy",
          memcpy_ftype, BUILT_IN_MEMCPY,
          BUILT_IN_NORMAL, "memcpy");

tree gm2_alloca_node
       = builtin_function
         ("__builtin_alloca",
          alloca_ftype, BUILT_IN_ALLOCA,
          BUILT_IN_NORMAL, "alloca");
```

In many instances the Modula-2 types can be mapped onto the equivalent C data types. However, as with many other languages, there will be specialist data types required which have no direct C equivalent. The most prominent examples in GNU Modula-2 are that of the open array or unbounded array and large sets. The open array mechanism allows programmers to specify an array parameter to a procedure has no fixed limit. The example below is a declaration for a procedure to concatenate two strings (performing $a := a + b$).

```
PROCEDURE concat (VAR a: ARRAY OF CHAR;
                  b: ARRAY OF CHAR) ;
```

GNU Modula-2 creates an internal `unbounded` type which is declared as a `RECORD`

```
unbounded = RECORD
               _arrayAddress: ADDRESS ;
               _arrayHigh   : CARDINAL ;
            END ;
```

A call to `concat` will involve the caller creating two unbounded temporary structures for parameters `a` and `b`. It fills in the fields to an `unbounded` structure with the address of the array and the last legal index. These two structures become the parameters into the procedure `concat`. GNU Modula-2 adopts the policy of callee save and therefore in the example `concat` must make a copy of the non `VAR` parameter (`b`). This is achieved using the following `tree` and it is called from within the front end Modula-2 source.

```
nBytes :=
  mult(add(indirect(add(addr(param),
                        offset(arrayHighField)),
                    getIntegerType()),
           getIntegerOne())
       findSize(arrayType)) ;

addr :=
  indirect(add(addr(param),
               offset(arrayAddressField)),
           getIntegerType()) ;

newArray := gccMemCopy(gccAlloca(nBytes),
                       addr,
                       nBytes) ;
```

This mechanism works well and utilises the functional interface to the `tree` data structure provided by the GCC back end. Essentially the single front end primitives `VAR a: ARRAY OF` *data type* and `a: ARRAY OF` *data type* are implemented by considering their C equivalent using non simple data types and in-lining calls to libc and in-lining C statements. Of course GNU Modula-2 does not generate C but rather it generates the same internal `tree`s that the GNU C compiler generates. In turn these `tree`s define the construction and manipulation of open array data structures.

10 Implementing sets using trees

In ISO Modula-2 set types may have more members than bits in a machine word. For example a user may define large set types in the following way.

[2] `alloca_ftype` and `memcpy_ftype` are the prototypes for the respective functions

```
TYPE
   foo = SET OF CHAR ;
   bar = SET OF [-1..01000H] ;
VAR
   a: foo ;
   b: bar ;
BEGIN
   a := {'a', 'c', d', 'z'} ;
   b := {1, 2, 3, 5, 7, 11, 01000H} ;
```

As the GCC back end does not contain a large set basic type GNU Modula-2 was forced with two choices. Either a new basic type would be added to the GCC back end or GNU Modula-2 could manufacture this type in a similar way to that of an open array. It was decided to manufacture multi word set types rather than introduce a new data type for the GCC back end. The advantages of this technique include simplicity and a clearer separation between the GCC releases and the GNU Modula-2 front end releases. The front end implements set comparison and set element testing routines based on single a multi word set types. It attempts to propagate constants by providing `tree` functions which exploit constant operands whenever possible.

In fact this practice has proved sensible as from GCC release 4.0 onwards the basic word sized set type has been removed. In its place GCC now provides GIMPLE and GENERIC which allow front ends to construct language specific types. Thus the changes to the GNU Modula-2 front end when migrating to GCC 4.1 will include recreating a set type via GIMPLE and GENERIC. GNU Modula-2 was in effect using the GIMPLE technique for both open arrays and large set types, thus the changes should be reasonably well isolated.

The GNU Modula-2 front end manufactures the `SET OF CHAR` construct for a 32 bit data word length processor by generating a GCC `tree` representing the record shown in figure 3. To hide this transformation from the user there exist a collection of patches to be applied to the

```
RECORD
  : SET OF [CHR(0)..CHR(WordLength-1)];
  : SET OF [CHR(WordLength)
            ..CHR(2*WordLength-1)];
  : SET OF [CHR(2*WordLength)
            ..CHR(3*WordLength-1)];
  : SET OF [CHR(3*WordLength)
            ..CHR(4*WordLength-1)];
  : SET OF [CHR(4*WordLength)
            ..CHR(5*WordLength-1)];
  : SET OF [CHR(5*WordLength)
            ..CHR(6*WordLength-1)];
  : SET OF [CHR(6*WordLength)
            ..CHR(7*WordLength-1)];
  : SET OF [CHR(7*WordLength)
            ..CHR(8*WordLength-1)];
END ;
```

Figure 3: Record representing SET OF CHAR

Modula-2 components of `gdb`. These patches allow users to print types and display data in a Modula-2 source code representation. The modified `gdb` understands that a structure containing word sized sets with contiguous ranges and `NULL` field names are to be displayed as a single large set. This mechanism is a pragmatic and release early [Raymond] solution and works well for small to medium sized sets, clearly a more compile time scalable solution is required for really large sets.

11 Modula-2 and interrupt priorities

Another important aspect of Modula-2 is that it provides all the necessary primitives to implement a microkernel either through language constructs or system procedures [Wirth4]. The language allows for modules to be specified to run with a specific interrupt mask. This has the effect that any procedure declared within a module will also inherit this interrupt mask. Thus when an exported procedure is invoked it will automatically set the processor interrupt mask to that of its parent module and restore the previous processor interrupt mask before

returning. For example in figure 4, it can be seen that procedure `foo` is declared in the inner module which was specified to operate with an interrupt mask of 7 whereas the outer module was specified to operate with an interrupt mask of 0. When the procedure `foo` is called from the outer module the current interrupt mask is saved and set to 7. After `foo` returns the interrupt mask is restored back to 0 again.

```
MODULE outer[0] ;

   MODULE inner[7] ;
   EXPORT foo ;

   PROCEDURE foo ;
   BEGIN
   END foo ;

   END inner ;

BEGIN
   foo
END outer.
```

Figure 4: Example of a procedure associated with an interrupt mask

11.1 Modula-2 and processes

The `SYSTEM` module as defined by Wirth [Wirth1][Wirth2][Wirth3] provides four procedures which can be used to create a process, context switch between two processes and switch to another process should an interrupt occur. The prototypes for these `SYSTEM` module procedures are given in figure 5. The procedure `NEWPROCESS` instantiates the procedure represented by parameter `p` into a process `new`. Whereas the procedure `TRANSFER` context switches from process `p1` to process `p2`. The procedure `IOTRANSFER` initially context switches from process `first` to `second` however when an interrupt occurs it saves the current processor volatile environment in `second` and then context switches back to the process `first`. The `LISTEN` procedure briefly removes the processor interrupt mask.

```
PROCEDURE NEWPROCESS (p: PROC;
                      a: ADDRESS;
                      n: CARDINAL;
                      VAR new: PROCESS)

PROCEDURE TRANSFER (VAR p1: PROCESS;
                    p2: PROCESS)

PROCEDURE IOTRANSFER (VAR First,
                          Second: PROCESS;
                      InterruptNo: CARDINAL)

PROCEDURE LISTEN
```

Figure 5: Prototypes of the SYSTEM procedures which coordinate process activity

Fortunately the GNU Pthread library contains low level context switching primitives and these allow for a straightforward implementation of `NEWPROCESS` and `TRANSFER`. The procedure `NEWPROCESS` is implemented by calling `pth_uctx_create` and `pth_uctx_make`. These two Pthread primitives create a process context. Each Modula-2 process is represented by a single Pthread context with an associated interrupt priority mask. In GNU Modula-2 the definition for the `PROCESS` type and the interrupt range and the implementation of `NEWPROCESS` is shown in figure 6.

The implementation of `TRANSFER` and `IOTRANSFER` are shown in figures 7 and 8 respectively. There are three categories of interrupts currently implemented in this runtime system: input, output and clock interrupts. The input and output interrupts are generated by providing a mapping to a file descriptor and the clock interrupts are simulated through the use of a relative ordered time ascending list. A module `SysVec` provides procedures to map file descriptors onto simulated interrupt vectors and it also implements an interrupt dispatcher. This dispatcher is called whenever the interrupt mask is altered and the duty of the dispatcher is to build the set parameters and time parameters for `pth_select`. The values to the set parameters are derived from the active interrupt list

which were populated by successive calls to `IncludeVector` via the `IOTRANSFER` procedure. A function `TurnInterrupts` was added as a GNU extension to the module `SYSTEM`. This function modifies the current interrupt mask and returns the previous mask value but it will also call the interrupt dispatcher if the new mask allows more interrupts to become visible. The compiler generates calls to `TurnInterrupts` whenever one procedure is about to call another procedure associated with a different interrupt mask (as in figure 4).

```
PROCESS  = RECORD
              context: ADDRESS ;
              ints   : PRIORITY ;
           END ;
PRIORITY = [0..7] ;

PROCEDURE NEWPROCESS (p: PROC; a: ADDRESS;
                      n: CARDINAL;
                      VAR new: PROCESS) ;
TYPE
   ThreadProcess = PROCEDURE (ADDRESS) ;
VAR
   ctx: ADDRESS ;
   tp : ThreadProcess ;
BEGIN
   localInit ;
   tp := ThreadProcess(p) ;
   IF pth_uctx_create(ADR(ctx))=0
   THEN
      Halt(__FILE__, __LINE__, __FUNCTION__,
           'unable to create user context')
   END ;
   IF pth_uctx_make(ctx, a, n, NIL, tp, NIL,
                    illegalFinish)=0
   THEN
      Halt(__FILE__, __LINE__, __FUNCTION__,
           'unable to make user context')
   END ;
   WITH new DO
      context := ctx ;
      ints    := currentIntValue ;
   END
END NEWPROCESS ;
```

Figure 6: Implementation of NEWPROCESS and definition of the type PROCESS

Every call to `IOTRANSFER` results in a new `IOTransferState` being constructed and this is kept on the callers stack so that the context of `first` process can be restored when the interrupt is serviced. The procedure `IOTRANSFER` initially saves the current processes context into

```
PROCEDURE TRANSFER (VAR p1: PROCESS;
                    p2: PROCESS) ;
VAR
   r: INTEGER ;
BEGIN
   localMain(p1) ;
   p1.ints := currentIntValue ;
   currentIntValue := p2.ints ;
   IF p1.context=p2.context
   THEN
      Halt(__FILE__, __LINE__, __FUNCTION__,
           'switching to the same process')
   END ;
   currentContext := p2.context ;
   IF pth_uctx_switch(p1.context,
                      p2.context)=0
   THEN
      Halt(__FILE__, __LINE__, __FUNCTION__,
           'unable to context switch')
   END
END TRANSFER ;
```

Figure 7: Implementation of TRANSFER

```
IOTransferState =
   RECORD
      ptrToFirst,
      ptrToSecond: POINTER TO PROCESS ;
      next: POINTER TO IOTransferState
   END ;

PROCEDURE IOTRANSFER (VAR First,
                      Second: PROCESS;
                      InterruptNo: CARDINAL)
VAR
   p: IOTransferState ;
BEGIN
   localMain(First) ;
   WITH p DO
      ptrToFirst  := ADR(First) ;
      ptrToSecond := ADR(Second) ;
      next := AttachVector(InterruptNo,
                           ADR(p))
   END ;
   IncludeVector(InterruptNo) ;
   TRANSFER(First, Second)
END IOTRANSFER ;
```

Figure 8: Implementation of IOTRANSFER

`first` and then restores the context belonging to process `second`. The `IOTransferState` is constructed and initialised appropriately. A pointer to this record, (`q` in figure 9) is created when calling `AttachVector`. The procedure `AttachVector` associates `q` with the particular interrupt number. When the interrupt is serviced the dispatcher context switches back

to process `second` by invoking `TRANSFER` and passing the relevant fields of `q`.

```
TRANSFER(q^.ptrToSecond^, q^.ptrToFirst^)
```

The last `SYSTEM` procedure `LISTEN` simply listens to all pending interrupts briefly before returning. `LISTEN` is easily implemented by a call to the interrupt dispatcher after un-masking all interrupts. The `SYSTEM` module also provides a non standard procedure `ListenLoop` which exhibits the same behaviour as:

```
LOOP
   LISTEN
END
```

except that it allows the interrupt dispatcher to block waiting for an interrupt to occur thus respecting the underlying operating system.

The implementation of Modula-2 processes works well, it only amounts to 1600 lines of code and it allows input, output and time based interrupts to be managed through `IOTRANSFER`. These modules form part of the GNU Modula-2 runtime system and they provide a method whereby microkernel executives originally designed for stand alone systems can be executed under UNIX like operating systems.

12 GNU Modula-2 source code

At the time of writing GNU Modula-2 0.5 has been released. This front end can be grafted onto GCC-3.3.6 source tree and the front end contains patches for GCC and GDB. To make the source code grafting and build process as simple as possible there is a build package which downloads the appropriate GCC, GDB and GM2 releases, applies the various patches

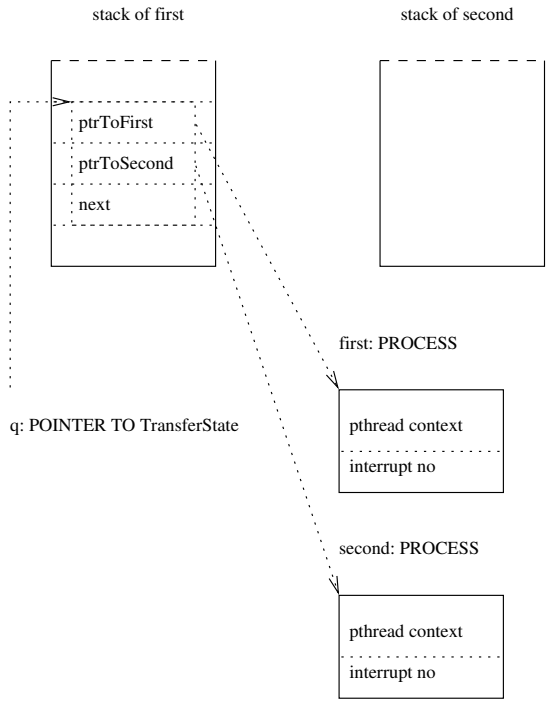


Figure 9: Interaction between IOTransferState and IOTRANSFER

and proceeds to build and install GNU Modula-2 and a modified version of GDB. This utility, `gm2-harness-0.7`, can be found on the GNU Modula-2 web site at `http://www.nongnu.org/gm2`.

13 Conclusions and further work

In conclusion `gm2` has been produced and it builds successfully with the GCC-3.3.6 release on UNIX, GNU/Linux, FreeBSD and Solaris on seven different processors.

The compiler is fully PIM Modula-2 compliant and a complete set of PIM libraries exist. The PIM libraries include: Logitech compatible libraries, University of Ulm libraries and m2f libraries. The user can specify which set of libraries an application should link against.

The compiler will bootstrap reliably and provides accurate debugging information for `gdb`. GNU Modula-2 has also been configured as a cross compiler for the StrongARM [Intel3] and MinGW platforms. It is also the only known free 64 bit implementation of Modula-2 and it will build successfully on the AMD Opteron [Opteron] and Intel Itanium [Intel1] processors running Debian Pure64.

The technique of double book keeping in the symbol table handling has been successful and simplifies the interface between the front and back end. The front end only translates error free and resolved symbols into the GCC `tree` equivalent.

Open array and multi word set implementation in GNU Modula-2 show that GCC front ends can successfully manufacture data types and manipulate data types by providing a mapping onto C derived `tree`s. This will smooth the transition to GCC release 4.1 which uses GENERIC and GIMPLE. It is expected that both: open arrays, set types could be expressed as front end `tree`s which are converted onto back end types appropriately. Ironically the choice of quadruples as front end intermediate code can also be exploited as the quadruples have a direct correspondence with GIMPLE code. This work will be undertaken in the near future.

GNU Modula-2 has implemented the ISO SYSTEM module and some of the ISO language features, clearly however the ISO Modula-2 dialect needs to be completed together with a set of ISO compatible libraries.

Lastly it is a core aim that GNU Modula-2 be integrated within the GCC source tree at a convenient time in the future.

14 Acknowledgements

I would like to thank my employer for funding this research, my colleagues for their support, and my family for being so patient. Thank you to all of the readers and contributors of the GNU Modula-2 mailing list for their many valuable bug reports, repeated test builds and patches over the last six years. Many people and organisations have been very generous in providing access to a wide variety of computing equipment which has enabled extensive testing and porting to occur.

Finally a great debt of thanks is owed to the Free Software Foundation and all its contributors without which the source code to GCC would not be free to read, modify, and redistribute.

References

[AMD64] AMD, *x86-64 Architecture Programmer's Manual*, AMD USA (2003).

[Bowen] Devon Bowen, *A Highly Portable Modula-2 Compiler*, The State University of New York at Buffalo, Computer Science Department, 226 Bell Hall, Buffalo, New York, 14260, USA (1994).

[Gancarz] Mike Gancarz, *Linux and the Unix Philosophy*, Digital Press (2002).

[GM2] Gaius Mulley, *The GNU Modula-2 front end to GCC*, Edition 0.5, Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA (2006).

[Granlund] Torbjörn Granlund and Richard Kenner, *Eliminating branches using a superoptimizer and the GNU C compiler*,

ACM SIG-PLAN Notices, Volume 27(7), p341-352 (1992).

[IBM970] IBM, *IBM PowerPC 970FX RISC Microprocessor User's Manual*, IBM, (2006).

[Intel1] Intel, *Intel Itanium Architecture Software Developer's Manual*, Volume 3: Instruction Set Reference, Revision 2.2, Intel (2006).

[Intel2] Intel, *Pentium Processor Family Developer's Manual*, Intel Literature, P.O. Box 7641, Mt. Prospect, IL 60056-7641, USA (1995).

[Intel3] Intel, *Intel StrongARM SA-1110 Microprocessor Developers Manual*, Intel, Intel, USA (2001).

[ISO] ISO/IEC, *Information technology - programming languages - part 1: Modula-2 Language*, ISO/IEC 10514-1 (1996).

[Lewis] Stuart Lewis and Gaius Mulley, *A comparison between novice and experienced compiler users in a learning environment*, 6th Annual Conference on the Teaching of Computing, 3rd Annual Conference on Integrating Technology into Computer Science Education, ITiCSE '98 18th - 31th August, Dublin Ireland, ACM 0-89791-xxx/98/03 (1998).

[MacKenzie] David MacKenzie and Ben Elliston, *Creating Automatic Configuration Scripts*, Edition 2.13, Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA (1998).

[Mulley] Gaius Mulley and Keith Verheyden, *Enhancing a Modula-2 compiler to help students learn interactively within the Ceilidh system*, Knowledge Transfer 97 (1997).

[Opteron] AMD, *Software Optimization Guide for the AMD Opteron Processor*, AMD, (2003).

[Pizka] Markus Pizka, *Design And Implementation of the GNU INSEL-Compiler gic*, Technische Universitaet Muenchen, Institut fuer Informatik (1997).

[Pronk] Cornelis Pronk, *Stress Testing of Compilers for Modula-2*, Software Practice and Experience, Volume 22(10), p885-897 (1992).

[Raymond] Eric Raymond, *The Cathedral and the Bazaar*, O'Reilly Publishers, USA (1999).

[Stallman1] Richard Stallman, *Using and Porting the GNU Compiler Collection*, Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110–1301 USA (2001).

[Stallman2] Richard Stallman, *GNU Coding Standards*, Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110–1301 USA (2006).

[Wirth1] Niklaus Wirth, *Programming in Modula-2*, Springer-Verlag, Berlin Heidelbery New York, 2nd Edition (1983).

[Wirth2] Niklaus Wirth, *Programming in Modula-2*, Springer-Verlag, Berlin Heidelbery New York, 3rd Edition (1985).

[Wirth3] Niklaus Wirth, *Programming in Modula-2*, Springer-Verlag, Berlin Heidelbery New York, 4th Edition (1988).

[Wirth4] Niklaus Wirth, *Design and Implementation of Modula*, Software

Practice and Experience, Vol 6(7),
p. 67–84 (1976).

Devirtualization in GCC

Mircea Namolaru
IBM Haifa Research Lab - HiPEAC Member
namolaru@il.ibm.com

Abstract

A major optimization for object oriented languages is converting dynamically bound function calls into (statically bound) direct calls, a process called devirtualization. This saves the dynamic dispatch overhead, and more importantly, enables further inlining of these function calls. For devirtualization we designed an extension of the Rapid Type Analysis (RTA) algorithm, a fast and effective algorithm [1]. The resulting algorithm combines RTA with a simple data-flow analysis.

We have an initial version of devirtualization for C++, implemented in GCC. We describe how the implementation makes use of the existent GCC code, and how it deals with the complexities of the C++ language. Finally we discuss the major implementation issues for completing the implementation.

1 Introduction

In order to support abstraction, object-orientated languages support dynamic binding of methods based on the run-time of the object. This presents a compiler with a problem, as it has to generate code to activate the method at run-time. In addition as the method invoked is unknown at compile-time, valuable optimizations opportunities may be lost. As dynamic binding is extensively used by object-orientated programmers, this may cause a significant overhand in performance.

In order to statically bind a method, we must be able to statically determine the type of the object upon which the method is invoked. A number of different analyses have been developed for C++ in an attempt to solve this problem.

One of them is Rapid Type Analysis (RTA), introduced in [1]. An in depth description of the algorithm, its performance and its implementation can be found in [2]. The RTA algorithm is a simple algorithm, and yet its performance compares very well versus other much more complex type inference algorithms.

Some simple data-flow analysis may succeed in devirtualizing calls for which RTA analysis have failed. For instance, a simple (intra-procedural and/or inter-procedural) type propagation algorithm may find that the object upon which a virtual method is invoked, is an object returned by a new operator (therefore its type is know statically). In spite of its simplicity, such an analysis may be quite effective, and complements RTA, creating a more powerful devirtualization algorithm.

We have designed an algorithm that combines RTA with a simple type propagation algorithm. The resulting call graph computed by our algorithm is more accurate then the one computed

by RTA and/or class propagation applied separately and this may cause better devirtualization. The differences between the original RTA algorithm and this algorithm are discussed further.

We have a partial implementation of this algorithm in GCC for C++ and we are working to complete it. RTA is conceptually simple, but due to the complexity of C++ and to the current GCC infrastructure, its implementation raises a number of issues that are discussed further.

2 The algorithm

As the RTA algorithm is the basis of our work it is presented first in a form adapted to our needs, which is slightly different from the one appearing in [2](e.g. we build the call graph on the fly)

RTA assumes that a Class Hierarchy Graph (CHG) that describes the inheritance relationship between classes is available. Another prerequisite for RTA is a call graph with only the virtual calls not resolved. The RTA analysis will compute the possible targets for a virtual call and the ones that have a single possible target can be devirtualized.

The RTA algorithm is outlined in Figure 1. It maintains a list of methods. This list is initialized with the root node of the call graph. Initially, no class in the program is considered as being live. The algorithm also maintains a list of the already visited call sites called the live call sites.

Each method in the list is analyzed in turn. For each virtual call site in the method, the static class of the object upon which the method is invoked and its live subclasses are used to find the possible targets of the virtual call and to build the corresponding edges in the call graph. As other subclass may be marked live at a later time, we maintain information about live call sites.

We mark as live all the classes instantiated in this method. When a new class is found to be live, new methods may be reached via live virtual calls invoked on a base of the new instantiated class. The information on live virtual call site is used to find these methods and to build the corresponding edges in the call graph.

At the end of the algorithm we have found all the live classes and resolved the virtual call sites in the call graph. Virtual call site with a single target can be devirtualized.

In our algorithm we integrate RTA with a simple data-flow analysis that propagates information about the types of objects passed as arguments. The algorithm is outlined in Figure 2.

If an argument of a call is an object returned by a new operator or it is a formal parameter of the method containing the call, the argument is marked as `df-dep` (data flow dependent). A virtual call invoked on a `df-dep` object (considered to be the first argument of the call) is marked as `df-dep` virtual call.

The lattice of values used for type propagation is `unknown`, `df-dep` and `rta`, where `unknown` is the minimal element and `rta` is the maximal one. For `df-dep` values there is a secondary value with the classes reaching the `df-dep` argument. The rules for propagation are described in Figure 3.

One alternative propagation rule would be that if two different classes are propagated to a `df-dep` argument, then the value propagated further is `rta`. But as the propagated classes are from the same hierarchy, maintaining all the classes reaching a `df-dep` argument could be implemented rather efficiently.

```
build_virtual_calls_targets:

 for each method m in the call graph
  visited (m) = false;
 for each class cls in the program
  live (cls) = false;
 list_methods = root_method;
  live_call_Sites = empty

 while list_methods is not empty

  remove m head of list_methods
  if (visited (m))
   continue

  for all call sites cs of m
   if is_virtual (cs)
    /* Based on the static type at cs and
       its live subclasses, find the possible
       targets at cs. */
    build_edges_cv (static_type (cs), m)
    add cs to live_call_sites
   else if is_not_virtual (cs)
    for each target tm in targets (cs)
     add tm to list_methods

  for each class cls instantiated in m
   if (live (cls)
    continue
   live (cls) = true
   for each call site cs in live_call_sites
    /* Find the method invoked at cs if
       dynamic type is cls.  */
    tm = target_cs_cls (cs, cls);
    if (tm == NULL) continue
    build_edge (m, tm)
    add tm to list_methods

  visited (m) = true;
```

Figure 1: The RTA algorithm

Initially, each class parameter of a method has the value `unknown`. A `df-dep` argument of a call that is a formal parameter of its enclosing method is initialized to the value `unknown`. A `df-dep` argument that is an object propagated from a new operator is initialized to the value `df-dep` (and the secondary value to the class instantiated by new). A non `df-dep` argument receives the value `rta`.

In the original RTA algorithm, every instantiated class assumes that all the live virtual calls may be reached by the new created object. Our algorithm optimistically assumes for a new instantiated object that `df-dep` virtual calls are not reached, letting data-flow analysis to find out if the call is reached or not.

For a `df-dep` virtual call, the computation of possible targets of the call is based on the type information propagated to the call, and not on the live classes information as in RTA.

Our algorithm requires that type information reaching the formal arguments of a method be propagated further in the call graph. Each time a method is reached, the type information of its arguments needs to be propagated to its call sites, and from there to the current targets of the call. A method is added to the method list each time new type information reaches its parameters.

Basically, this algorithm tries to infer the type of `df-dep` arguments via a simple data-flow analysis (propagation of type of objects returned by a new operator). For the rest of the arguments, the RTA analysis is used.

As in RTA, at the end of the algorithm we have found all the live classes and resolved the virtual call sites in the call graph.

3 An example for modified RTA

To illustrate the algorithms from the previous section and the differences between them we will use the example from Figure 4.

In the example, we have a simple class hierarchy, where B is a subclass of A and C a subclass of B. The class B overrides the methods foo and foo1 (A::foo and A::foo1 not shown), and the class C overrides the method foo1.

There are two virtual call sites, one in foo2 and the other in B::foo which is a `def-dep` call site.

```
build_virtual_calls_targets:

 for each method m in the call graph
  visited (m) = false
  df_init (m)
 for each class cls in the program
  live (cls) = false;
 list_methods = root_method
 live_call_Sites = empty

 while list_methods is not empty

  remove m head of list_methods
  if (visited (m))
   for all call sites cs of m
    for each target tm in targets (cs))
     /* Propagate type information from the
     formals of the method to the df-dep
     arguments in the call site.  */
     df_prop (m, cs)
     /* Check if new type information is
       propagated to the target.  */
     if (df_merge (cs, tm))
      add tm to list_methods
   continue

  for all call sites cs of m
   if is_virtual (cs) && not_df_dep (cs)
    /* Based on the static type at cs and
       its live subclasses, find the
       possible targets at cs. */
    build_edges_cv (static_type (cs), m)
    add cs to live_call_sites
   else if is_virtual (cs) && df_dep (cs)
    /* Based on the type information
       propagated at cs, find the possible
       targets at cs.  */
    build_edges_cv (propagated_type (cs), m)
   else if is_not_virtual (cs)
    /* Check if new type information is
       propagated to the target.  */
    for each target tm in targets (cs)
     if (df_merge (cs, tmp) or not visited (tm)
       add tm to list_methods

  for each class cls instantiated in m
   if (live (cls)
    continue
   live (cls) = true
   for each call site cs in live_call_sites
    /* Find the method invoked at cs if
       dynamic type is cls.  */
    tm = target_cls_cs (cs, cls);
    if (tm == NULL) continue
    /* Check if new type information is
       propagated to the target.  */
    if (df_merge (cs, tm) or not visited (tm))
     add tm to list_methods
    build_edge (cs, tm)

  visited (m) = true;
```

Figure 2: The modified RTA algorithm

```
prop (unknown, v) = v, v any value

prop (v, rta) = rta  , v any value

prop ((df-dep,A),(df-dep,B)) = (df-dep,(A, B))
```

Figure 3: Type propagation rules

The method list is initialized with foo2. First we find that the class C is instantiated and that B::foo is reachable. Then we discover that C::foo1 is reachable and that the class B is instantiated.

From this point onward, the two algorithms differ. RTA will consider that B::foo1 is reachable (via the call site from B::foo), and that the class A is instantiated. This will find two new methods reachable via the two call sites A::foo and A::foo1. So in this case, RTA will find that the classes A, B and C are live and no call site can be devirtualized.

If type propagation is done afterward, it will succeed to devirtualize the call site from B::foo. However, since the class live information remains the same, the other virtual call site could not be devirtualized.

With the modified RTA algorithm, after class B has been instantiated, we will not consider the method B::foo1 reachable. As again B::foo is reached (via the call site from foo2), we will check if new type information is propagated to it. This doesn't happen in this example, and the algorithm ends. The algorithm will find that the classes B and C are live. Since both call sites have a single target in the call graph, both can be devirtualized.

4 RTA issues

Since our algorithm is based on RTA, it shares its requirements and limitations which are de-

```
// B subclass of A
// C subclass of B
static A *pn;

B:: foo1 (A *p) {
  new A;
}

C:: foo1 (A *) {
  new B;
}

B:: foo (A *p) {
  p->foo1 ( );
}

foo2 () {
  p = new C;

  pn->foo (p);
}
```

Figure 4: An example for modified RTA

```
// B subclass of A
void *obj = (void *)new A
B *obj1 = (B *) obj
obj1->foo

    (a)


void *obj = (void *)new B
B *obj1 = (B *)
obj obj1->foo

     (b)
```

Figure 5: Downcast examples in C++

scribed further.

4.1 Type safety

RTA assumes that the dynamic type of the object upon which a virtual call may be invoked is a subclass of its static type. In C++, this assumption may be invalidated via a downcast as it is possible to see in Figure 5(a). This code is not type-safe, and it may cause a run-time exception if foo is not defined in A. If foo is defined in A, many C++ implementation (including GCC) may invoke it, which may or may not be what the programmer expected. But in this case RTA will decide that the target of this virtual call is B::foo and change the behavior of the program. As we see in Figure 5(b), in the presence of a downcast RTA may work perfectly well. To statically differentiate between such cases is not always possible.

Two possible solutions are discussed in [1]. The first is to not apply RTA if a downcast is detected in the program. As downcasting is a common C++ practice, this may restrict too much this optimization. A better solution is to print a message if a downcast is encountered. The message will indicate that RTA may change the behavior of the program for truly unsafe downcasting.

4.2 Whole program analysis

In order to ensure the correctness of RTA, the entire program code must be analyzed (otherwise some instantiation of a class may be missed). This is not always possible. In many cases, part of the code is supplied in libraries whose code is not available. Following [2], we show the modifications required to RTA to handle incomplete programs.

We differentiate between classes internal to the program and classes exported by libraries. The classes internal to the program are not know to the libraries, hence they cannot be instantiated there, but their methods can be called from libraries. This may happen if their address have been taken in the program. Therefore, we need to consider all the methods whose address have been taken in the program as roots in the call graph (in the algorithms described previously, the initial list of methods should include them). Another change required is that classes exported by the libraries are initially considered live.

We must also consider the case when the program subclasses a library class and overrides one of its methods. When such a subclass in instantiated, all its methods that override methods in the library code must be assumed to be reachable (as they may be invoked by a virtual call in the library) and inserted in the list of methods.

The whole class hierarchy of the program is known. The compiler may analyze the class hierarchy and mark the standard C++ libraries classes as exported. The rest of the classes are considered internal to the program. If the program exports also other libraries beside the standard C++ libraries, information about their exported classes should be provided by the user.

5 Implementation

In this section, we describe several implementation problems and discuss the current stage of the implementation, as well as future work items.

5.1 Class instantiation

The algorithm needs to detect all the instantiated classes in the program. In order to do this, we find all invocations of constructors in the program. Constructors for sub-objects part of a base in a derived type are not considered to instantiate a class.

The constructors are identified at the gimple (or in the development branch used at the SSA) level, with the help of a C++ hook. The advantage of this approach is that an easy adaptation of this analysis to other languages is possible. The problem is that some constructors may be inlined by the front end and are not appearing at gimple (or SSA) level. However, we thought that would be easier for a given language to modify the front end, and let the regular inlining at gimple (or SAA) level to inline the constructors.

5.2 Class Hierarchy Graph

The implementation of the RTA algorithm needs a data structure to represent the class hierarchy graph (CHG). In this graph all nodes represent classes, and the edges describe the hierarchy relationship. We need to be able to reach from a base all its immediate subclasses, and from a subclass all its direct bases. Also, we want to annotate the nodes with the specific information needed by our analysis. We have implemented the CHG as a separate data structure internal to our analysis. There is a hash table that provides a mapping from a type to its corresponding node in the CHG.

The information about the bases of a given class is already available in GCC in the binfo (base info) trees build by the front end. As a class is instantiated, we complete the CHG with edges from a base to its subclasses. This is done by an upward traversal of the CHG starting with the class instantiated. For each edge (subclass, base) traversed, an edge (base, subclass) is constructed. If a node in CHG has no downward edges to its subclasses, none of its subclasses have been instantiated (or it is a leaf node). The information about instantiated classes is also kept in the CHG nodes.

5.3 Resolving a virtual call site

In this section, we show how the information about instantiated (or live) classes is used to resolve a virtual call site in the call graph. For such a call, the static class of the object upon which the method is invoked is available. The instantiated classes in the subgraph rooted at

this class provide the possible dynamic types for this object. For every couple (static type, dynamic type) we find the method that will be invoked at run-time. These methods are the targets of the virtual call analyzed.

We have implemented a method that given the static and dynamic type upon which a virtual call is invoked returns the method invoked (a simple variant of this method was already existent in GCC for the case when the dynamic type and the static type are the same). Such a method is dependent on how the object model (the layout of objects, the virtual method tables etc) is implemented. This is the reason for which this method is provided as a C++ hook.

5.4 A special issue for C++

There is a special issue in C++, for virtual methods invoked on the `this` object in a constructor. For the example in Figure 6, if an object of type B is created, the constructor of A is invoked (via the constructor of B), and then the virtual method foo is invoked on `this`. The C++ reference manual specifies that in this case, the type of `this` is A and not B (as we would expect, since the A constructor was invoked on a B object). In order to preserve the correctness of RTA we must consider that class A is live (even if no explicit instantiation of this class have been found).

A possible solution to this problem is to do in constructors an escape analysis of `this`. If it is passed as an argument to another method (other than the implicit `this` pointer) or it is copied, it is considered as escaped. In this case we will consider the class defining the constructor as live.

```
class A {
  public A::A() { foo ();}
  virtual void foo ();
}

// B subclass of A
class B: public A {
  public void foo ();
}
```

Figure 6: A C++ problem

5.5 A simple variant of RTA

We have already implemented a simple variant of RTA to help us to build the infrastructure needed by the more complex algorithms described in this paper. It is a RTA algorithm that assumes that all the methods are reached.

In the first stage, all the methods are scanned and the classes instantiated are marked as live. In this stage we construct the CHG as shown in a previous subsection. In the second stage for all virtual calls we find their targets in the call graph, and build the corresponding edges.

In the case when a virtual call has a single target, it is replaced by a direct call. This made possible the further inlining of this method.

5.6 Call graph

In GCC, a call graph has been already implemented. At this point no analysis is done to try to infer the possible targets for a call done via a pointer. In the absence of this information we need to make very conservative assumptions regarding the methods reachable via such calls. This was another reason for which we implemented the simple variant of RTA from the previous section that assumes that all methods are reachable. For a complete RTA we will need to address the issue of calls via pointers first.

5.7 Inter-module analysis

RTA requires inter-module analysis. GCC already has an inter-module analysis capability for C (enabled by the option -combine), that at this stage is not functional for C++. The problems that prevent this have been detailed in [5]. The lack of this capability for C++ is a problem that needs to be solved, in order to make possible the implementation of powerful inter-procedural algorithms in GCC C++.

5.8 RTA implementation

For an efficient implementation of the RTA, we need to be able to reach from a newly instantiated class all the live virtual calls that may be affected. This can be done by maintaining a mapping from a class to the live virtual calls that are statically invoked on it. For an instantiated class, we will start an upward traversing in the CHG starting at the class instantiated. During this traversal, the mapping will provide all the virtual call sites that may be affected by this class instantiation. This is a work item.

5.9 Type propagation

The inter-procedural constant propagation optimization already implemented in GCC [4] computes information about formal parameters of a method used as arguments in a call in the method. It provides an inter-procedural propagation engine, that could be extended to also propagate type information.

For determining that an object returned by a new operator reaches an argument in a call site (found in the same method as the new) the intra-procedural constant propagation implemented in SSA may be used.

An implementation, based on the existent GCC infrastructure, of the type propagation needed by the modified RTA algorithm is another work item.

5.10 Other languages

We have seen that the implementation of RTA (or one of its variants) make use of only two language hooks, one for detecting constructors in the language and the other for finding the method invoked by a virtual call given the static and the dynamic type upon which the call is invoked.

6 Future work

We intend to complete the implementation of the modified RTA algorithm. As we have seen, there are parts in the infrastructure that are missing (an accurate call graph and the inter-module capability for C++). We will see how we can help (together with GCC community) to make them functional.

We intend to asses and to tune the performance of this algorithm on the new SPEC2006 that has much more C++ benchmarks then SPEC2000 (where eon was the single C++ benchmark).

7 Conclusions

We have presented an algorithm that integrates two simple, but effective analysis, RTA and a simple data-flow analysis (type propagation). The resulting analysis is more accurate then if these analysis are applied separately.

We have described a partial implementation of this algorithm and outlined the issues for completing its implementation.

8 Acknowledgements

I would like to thank the IBM Haifa team for advising with implementation and design issues, to Peter Bergner for reviewing this paper, as well as to Daniel Berlin and all other GCC developers that provided helpful comments.

References

[1] David F. Bacon and Peter F. Sweeney. *Fast static analysis of C++ virtual function calls.* OOPSLA'96.

[2] David F. Bacon. *Fast and Efective Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, University of California at Berkeley, 1998.

[3] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. *Interprocedural Constant Propagation.* Symp. on Comp. Construct, 1986.

[4] Razya Ladelsky and Mircea Namolaru. *Interprocedural Constant Propagation in GCC.* GCC Developer's Summit, 2005.

[5] Geoff Keating. *Inter-module analysis in GCC.* GCC Developer's Summit, 2005.

[6] CodeSourcery and others. *Itanium C++ ABI (Revision: 1.86).* `http://www.codesourcery.com/cxx-abi/`.

[7] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. *A study of devirtualization techniques for a Java Just-In-Time compiler.* OOPSLA, 2000.

[8] David Detlefs and Ole Agesen. *Inlining of Virtual Methods.* European Conference on Object-Oriented Programming, 1999.

[9] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[10] Ken Kennedy and Randy Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann, 2001.

OpenMP and automatic parallelization in GCC

Diego Novillo
Red Hat Canada
dnovillo@redhat.com

Abstract

This paper describes the design and implementation of the OpenMP specification v2.5 in GCC. The implementation supports all the languages specified in the standard (C, C++ and Fortran), and it is generally available on any platform that supports POSIX threads.

Emphasis is placed on the internal architecture and, in particular, the intermediate representation, which could be used in the implementation of automatic parallelization techniques. The paper also presents performance results on the SPEC OMP2001 benchmark.

1 Introduction

OpenMP defines language extensions to C, C++ and Fortran for implementing shared-memory multi-threaded applications [1]. Compiler pragmas are used to define parallel regions, data and work sharing attributes. A runtime library implements the actual mechanism for creating threads, synchronization and data sharing.

This paper describes GOMP (GNU OpenMP), an OpenMP implementation for GCC. There are four main components: parser, intermediate representation, code generation and the runtime library (`libgomp`). The parser identifies and validates the OpenMP pragmas and emits the corresponding GENERIC representation. The IR used to represent OpenMP is an extension to GENERIC and GIMPLE. It serves a dual purpose: as an interface to `libgomp` and as a code generation target for auto-parallelization transformations.

2 Parser

OpenMP defines a collection of compiler pragmas for C, C++ and Fortran. As such, three separate implementations were required for each of the front ends. The new pragmas are categorized in two groups: **directives** for specifying parallelism and work-sharing, and **clauses** for specifying data sharing and thread scheduling properties.

Every OpenMP command starts with `#pragma omp` and though the standard defines quite a few of them, they are mostly straightforward to recognize in a recursive descent scan. The recognition code is hooked into the standard pragma processing code in each of the front ends: `c-parser.c:c_parser_omp_*` for C, `cp/parser.c:cp_parser_omp_*` for C++ and `fortran/parse.c:parse_omp_*` for Fortran.

Once recognized, the front ends generate the corresponding GENERIC representation as described in the next section. Some of the semantic analysis and validation is also done during parsing. Structural diagnostics such as nesting of directives is done after the representation is in GIMPLE form (`omp-low.c:diagnose_omp_structured_block_errors`). Other common diagnostics are emitted during the conversion into GIMPLE (`gimplify.c:gimplify_omp_*` and `gimplify.c:omp_*`).

3 Intermediate Representation

Most directives and clauses have a corresponding GENERIC node defined in `tree.def`. The basic code generation strategy is to outline the body of parallel regions into functions that are used as arguments to the `libgomp` thread creation routines. Data sharing is implemented by passing the address of a local structure with all the data items marked for sharing. Copy-in data is passed by value, while copy-in/copy-out data and variables that are bigger than a certain threshold are passed by address.

To illustrate at a high-level how OpenMP programs are compiled, consider the program in Figure 1 to compute the sum of all the thread IDs in parallel.[1]

Figure 2 shows the corresponding High GIMPLE representation. Note that for debugging convenience, the IL pretty-printer renders OpenMP statements using the `#pragma omp` syntax. Some transformations and mappings are done during parsing and gimplification. For instance, all predetermined or implicitly determined sharing attributes are made explicit for the benefit of code generation. In the case of Figure 2, variable `sum` is predetermined shared. Also, the atomic add operation is mapped into the corresponding `__sync` built-in.

[1] Yes, the program makes absolutely no sense.

```
main()
{
  int sum = 0;
  #pragma omp parallel
    {
      #pragma omp atomic
      sum += omp_get_thread_num ();
    }
  printf ("sum = %d\n", sum);
}
```

Figure 1: OpenMP program to compute a sum.

```
main ()
{
  sum = 0;
  #pragma omp parallel shared(sum)
    {
      D.1324 = omp_get_thread_num ();
      D.1325 = (unsigned int) D.1324;
      __sync_fetch_and_add_4 (&sum, D.1325);
    }
  sum.0 = sum;
  printf ("sum = %d\n", sum.0);
}
```

Figure 2: High GIMPLE form for Figure 1.

The next lowering stage (`omp-low.c:pass_lower_omp`) sets up mappings for satisfying data sharing attributes and linearizes the bodies of the OpenMP directives. Converting the code into linear form, requires the addition of `OMP_RETURN` markers that indicate the end of each body. This becomes important later when the parallel work-sharing regions are expanded into the corresponding `libgomp` calls. In Figure 3, the `OMP_RETURN` at line 9 marks the end of the parallel region starting at line 3.

Data sharing is implemented using an artificial data structure (`struct .omp_data_s`) whose fields are all the variables included

```
main ()
{
 1  sum = 0;
 2  .omp_data_o.sum = &sum;
 3  #pragma omp parallel shared(sum)
 4  .omp_data_i = &.omp_data_o;
 5  D.1324 = omp_get_thread_num ();
 6  D.1325 = (unsigned int) D.1324;
 7  D.1334 = .omp_data_i->sum;
 8  __sync_fetch_and_add_4 (D.1334, D.1325);
 9  OMP_RETURN
10  sum.0 = sum;
11  printf (&"sum = %d\n"[0], sum.0);
12  return;
}
```

Figure 3: Low GIMPLE form for Figure 1.

in data sharing clauses like `shared` and `copyin`. This is why the front end is required to explicitly indicate all the variables with sharing semantics. In general, variables with sharing or copy-in/copy-out semantics are passed by reference while variables with copy-in semantics are passed by value. However, if a copy-in variable is too large, it will also be passed by reference. This is controlled by `omp-low.c:use_pointer_for_field`.

Two local variables are created: `.omp_data_o`, which is filled in with the addresses and values of every shared variable to be sent to the children threads (line 2 in Figure 3), and `.omp_data_i`, which will hold the address of `.omp_data_o` (line 4 in Figure 3). This way, every reference to variable `sum` inside the body of the `omp parallel` directive, is rewritten to use `.omp_data_i->sum`.

This seemingly convoluted rewriting is necessary for outlining the body of the `omp parallel` into a separate function as shown in Figure 4. The new function `main.omp_fn.0` receives `&.omp_data_o` in its argument `.omp_data_i`. Final expansion replaces the parallel body with calls into

```
main ()
{
 1  # BLOCK 0
 2  # PRED: ENTRY (fallthru)
 3  sum = 0;
 4  .omp_data_o.sum = &sum;
 5  __builtin_GOMP_parallel_start(main.omp_fn.0,
 6                                &.omp_data_o, 0);
 7  main.omp_fn.0 (&.omp_data_o);
 8  __builtin_GOMP_parallel_end ();
 9  sum.0 = sum;
10  printf (&"sum = %d\n"[0], sum.0);
11  return;
12  # SUCC: EXIT
}


main.omp_fn.0 (.omp_data_i)
{
13  # BLOCK 0
14  # PRED: ENTRY (fallthru)
15  D.1324 = omp_get_thread_num ();
16  D.1325 = (unsigned int) D.1324;
17  D.1334 = .omp_data_i->sum;
18  __sync_fetch_and_add_4 (D.1334, D.1325);
19  return;
20  # SUCC: EXIT
}
```

Figure 4: Final expansion for Figure 1.

`libgomp` to launch children threads and execute `main.omp_fn.0` (lines $5-8$ in Figure 4).

The sequence of transformations proceeds as follows:

1. The front end parses the OpenMP pragmas and emits the corresponding GENERIC statements as described in Section 3.1.

2. The gimplifier determines which variables are used inside parallel regions and establishes mappings according to the data sharing clauses. It also tries to replace `omp atomic` directives with corresponding atomic update functions.

3. `pass_lower_omp` creates the artificial data structure to implement the data sharing mappings, rewrites variables to use the fields in `struct .omp_data_s`, expands some forms of synchronization and adds `OMP_RETURN` markers for directive bodies.

4. `pass_lower_cf` linearizes the directives and their bodies to remove the nested property and prepare the IL for building the flow graph.

5. `pass_build_cfg` builds the control flow graph, making sure that incoming edges into parallel regions are marked abnormal to avoid CFG cleanups from making any assumptions that may violate parallel semantics. This is mostly a precautionary measure, as no such cleanups are currently implemented that may cause these problems.

   One important property about `omp parallel` regions is that they are guaranteed to be single-entry, single-exit. This is exploited by the expansion phase.

6. `pass_expand_omp` runs just before the code is put into SSA form. With the existing implementation, `omp parallel` regions cannot be put into SSA form because it does not support concurrency semantics.

   This pass outlines the single-entry, single-exit region of every `omp parallel` into a new function and expands all the other directives into calls to `libgomp` or the corresponding GIMPLE expansion. For instance, the computations needed to calculate iteration space bounds for statically scheduled parallel loops are expanded inline (Figures 5(a) and 5(b)).

3.1 Directives

Most OpenMP directives and clauses have a corresponding GENERIC and GIMPLE code. The exception are those that can be represented with built-in function calls (e.g. `omp barrier`, `omp flush`) or attributes (e.g. `omp threadprivate` are handled with the standard the thread-local storage attributes).

Calls to `libgomp` are encoded as built-in functions in `omp-builtins.def`. Directives and clauses encoded as IL statements are defined in `tree.def`. All the front ends emit the statements and built-ins defined in these files.

The C and C++ front ends share common code generation routines in `c-omp.c` while the Fortran front end converts its parse trees into GENERIC in `fortran/trans-openmp.c`.

`OMP_PARALLEL`

Represents `#pragma omp parallel [clause1 ... clauseN]`. It has four operands:

Operand `OMP_PARALLEL_BODY` is valid while in GENERIC and High GIMPLE forms. It contains the body of code to be executed by all the threads. During GIMPLE lowering, this operand becomes `NULL` and the body is emitted linearly after `OMP_PARALLEL`.

Operand `OMP_PARALLEL_CLAUSES` is the list of clauses associated with the directive.

Operand `OMP_PARALLEL_FN` is created by `pass_lower_omp`, it contains the `FUNCTION_DECL` for the function that will contain the body of the parallel region.

Operand `OMP_PARALLEL_DATA_ARG` is also created by `pass_lower_omp`. If

```
foo ()
{
  #pragma omp for
  for (i = 0; i <= 8; i = i + 1)
  do_work (i);
  OMP_CONTINUE
  OMP_RETURN
  return;
}
```

(a) Low GIMPLE form.

```
foo ()
{
      /* Lines 3-14 compute the iteration space for
         each thread.     */
 3    D.1330 = __builtin_omp_get_num_threads ();
 4    D.1331 = (unsigned int) D.1330;
 5    D.1332 = __builtin_omp_get_thread_num ();
 6    D.1333 = (unsigned int) D.1332;
 7    D.1334 = 9 / D.1331;
 8    D.1335 = D.1334 * D.1331;
 9    D.1336 = D.1335 != 9;
10    D.1337 = D.1334 + D.1336;
11    D.1338 = D.1337 * D.1333;
12    D.1339 = D.1338 + D.1337;
13    D.1340 = MIN_EXPR <D.1339, 9>;
14    if (D.1338 >= D.1340) goto <L3>; else goto <L0>;
      /* Lines 20-25 compute the first and last value of
         'i' taking the loop increment value into
         consideration.      */
17    # BLOCK 1
19  <L0>:;
20    D.1341 = (int) D.1338;
21    D.1342 = D.1341 * 1;
22    i = D.1342 + 0;
23    D.1343 = (int) D.1340;
24    D.1344 = D.1343 * 1;
25    D.1345 = D.1344 + 0;
      /* Lines 31-34 are the actual loop.       */
28    # BLOCK 2
30  <L1>:;
31    do_work (i);
32    i = i + 1;
33    D.1346 = i < D.1345;
34    if (D.1346) goto <L1>; else goto <L3>;
      /* This barrier is emitted because the loop
         was not marked with the 'nowait' clause.       */
37    # BLOCK 3
39  <L3>:;
40    __builtin_GOMP_barrier ();
41    return;
}
```

(b) Corresponding expansion.

Figure 5: Expansion of a statically scheduled parallel loop.

there are shared variables to be communicated to the children threads, this operand will contain the VAR_DECL that contains all the shared values and variables.

OMP_FOR

Represents #pragma omp for [clause1 ... clauseN]. It has 5 operands:

Operand OMP_FOR_BODY contains the loop body.

Operand OMP_FOR_CLAUSES is the list of clauses associated with the directive.

Operand OMP_FOR_INIT is the loop initialization code of the form VAR = N1.

Operand OMP_FOR_COND is the loop conditional expression of the form VAR {<,>,<=,>=} N2.

Operand OMP_FOR_INCR is the loop index increment of the form VAR {+=,-=} INCR.

Operand OMP_FOR_PRE_BODY contains side-effect code from operands OMP_FOR_INIT, OMP_FOR_COND and OMP_FOR_INC. These side-effects are part of the OMP_FOR block but must be evaluated before the start of loop body.

The loop index variable VAR must be a signed integer variable, which is implicitly private to each thread. Bounds N1 and N2 and the increment expression INCR are required to be loop invariant integer expressions that are evaluated without any synchronization. The evaluation order, frequency of evaluation and side-effects are unspecified by the standard.

OMP_SECTIONS

Represents #pragma omp sections [clause1 ... clauseN].

Operand OMP_SECTIONS_BODY contains the sections body, which in turn contains a set of OMP_SECTION nodes for each of the concurrent sections delimited by #pragma omp section.

Operand OMP_SECTIONS_CLAUSES is the list of clauses associated with the directive.

OMP_SINGLE

Represents #pragma omp single.

Operand OMP_SINGLE_BODY contains the body of code to be executed by a single thread.

Operand OMP_SINGLE_CLAUSES is the list of clauses associated with the directive.

OMP_MASTER

Represents #pragma omp master.

Operand OMP_MASTER_BODY contains the body of code to be executed by the master thread.

OMP_ORDERED

Represents #pragma omp ordered.

Operand OMP_ORDERED_BODY contains the body of code to be executed in the sequential order dictated by the loop index variable.

OMP_CRITICAL

Represents #pragma omp critical [name].

Operand OMP_CRITICAL_BODY is the critical section.

Operand OMP_CRITICAL_NAME is an optional identifier to label the critical section.

OMP_ATOMIC

Represents #pragma omp atomic.

Operand 0 is the address at which the atomic operation is to be performed.

Operand 1 is the expression to evaluate. The gimplifier tries three alternative code generation strategies. Whenever possible, an atomic update built-in is used. If that fails, a compare-and-swap loop is attempted. If that also fails, a regular critical section around the expression is used.

`OMP_RETURN`

This does not represent any OpenMP directive, it is an artificial marker to indicate the end of the body of an OpenMP. It is used by the flow graph (`tree-cfg.c`) and OpenMP region building code (`omp-low.c`).

`OMP_CONTINUE`

Similarly, this instruction does not represent an OpenMP directive, it is used by `OMP_FOR` and `OMP_SECTIONS` to mark the place where the code needs to loop to the next iteration (in the case of `OMP_FOR`) or the next section (in the case of `OMP_SECTIONS`).

In some cases, `OMP_CONTINUE` is placed right before `OMP_RETURN`. But if there are cleanups that need to occur right after the looping body, it will be emitted between `OMP_CONTINUE` and `OMP_RETURN`.

3.2 Clauses

Clause codes are defined in `tree.h` as subcodes for the main `OMP_CLAUSE` code. This was necessary because of code space overflow in `tree.def`. GCC does not support more than 256 IL codes, so clauses are all represented by a main code (`OMP_CLAUSE`) and a sub-code, which can be one of `OMP_CLAUSE_PRIVATE`, `OMP_CLAUSE_SHARED`, `OMP_CLAUSE_FIRSTPRIVATE`, `OMP_CLAUSE_LASTPRIVATE`, `OMP_CLAUSE_COPYIN`, `OMP_CLAUSE_COPYPRIVATE`, `OMP_CLAUSE_IF`, `OMP_CLAUSE_NUM_THREADS`, `OMP_CLAUSE_SCHEDULE`, `OMP_CLAUSE_NOWAIT`, `OMP_CLAUSE_ORDERED`, `OMP_CLAUSE_DEFAULT`, and `OMP_CLAUSE_REDUCTION`.

Clauses associated with the same directive are chained together via `OMP_CLAUSE_CHAIN`. Those clauses that accept a list of variables are restricted to exactly one, accessed with `OMP_CLAUSE_VAR`. Therefore, multiple variables under the same clause *C* need to be represented as multiple *C* clauses chained together. This facilitates adding new clauses during compilation.

4 Auto parallelization

The new GENERIC and GIMPLE codes used for OpenMP can also be the target for an auto parallelization pass. Although GCC does not currently implement such a transformation, all the necessary data dependency and code generation tools are already present.

It is possible to emit both task and data parallel code using `OMP_SECTIONS` and `OMP_FOR` respectively. Data sharing semantics can be implemented with the corresponding `OMP_CLAUSE_*` codes and synchronization needed to preserve sequential data dependency semantics may use the appropriate `OMP` directive or call the `libgomp` routines directly.

Once parallel GIMPLE code is generated, `pass_expand_omp` may be used to do the outlining and low-level expansion work, and schedule the new function into the call-graph. Currently, care should be taken to take the function out of SSA form prior to these transformations because the call graph manager currently expects functions to be in normal form. However, this limitation may be lifted in the future.

5 Runtime Library

The runtime library (`libgomp`) is essentially a wrapper around the POSIX threads library, with some target-specific optimizations for systems that support lighter weight implementation of certain primitives. For instance, locking primitives in some Linux targets are implemented using atomic instructions and futex system calls. To support `libgomp`, the target must also implement thread-local storage.

The implementation is in `gcc/libgomp` and most entry points into the library are defined as built-in function calls inside the compiler.

5.1 Thread creation

The main entry point is `GOMP_parallel_start`, which takes as arguments the function to run on each thread, a pointer to the `.omp_data_s` structure as described earlier and the number of threads to be launched. If the specified number of threads is 0, the number of threads is computed automatically.

Once the parallel region ends, threads are docked so that they can be re-used at a later time. The master thread keeps executing the code after `GOMP_parallel_start`, which in this case is just another invocation to the same function that the children threads are executing. A call to `GOMP_parallel_end` Tears down the team and returns to the previous parallel state.

There are alternate entry points for combined parallel and work-sharing constructs that avoid one extra synchronization at the start of the work-sharing construct. The compiler tries to emit these combined calls whenever possible (`omp-low.c:determine_parallel_type`).

5.2 Synchronization

With few exceptions, most synchronization is just a direct mapping to the underlying POSIX routines. The exceptions are `omp master` and `omp single`:

`omp master` simply blocks the thread with a thread-id different than 0.

`omp single` has two alternate entry points, with and without the `copyprivate` clause. Since `copyprivate` is used to broadcast the values computed inside the `omp single` body, the compiler emits a call to `GOMP_single_copy_start`, which will block all the threads except one. On return, the blocked threads receive a pointer into a common area which will have been filled by the thread that entered the region. That area contains the broadcast data. See `omp-low.c:lower_omp_single_copy` for details.

5.3 Work sharing

Every scheduling variant of `omp for` has been implemented in the library. There are three main functions, `GOMP_loop_*_start` to initialize the loop bounds, `GOMP_loop_*_next` to get the next chunk of iteration space to work on, and `GOMP_loop_*_end` to finalize the parallel loop.

The `omp sections` construct is simpler. The compiler transforms the construct into a `switch` statement using the section id as index. The call to `GOMP_sections_start` sets up the work-share construct and record the number of sections found in the body. `GOMP_sections_next` returns the next section id to execute. Once all the sections have been executed, a barrier after the `switch` synchronizes all the threads.

Benchmark	ICC 9.0	GCC 4.2.0	% Diff
wupwise	227.0	224.0	-1.3%
swim	140.0	138.0	-1.4%
mgrid	146.0	140.0	-4.1%
applu	154.9	147.3	-4.9%
equake	267.2	264.5	-1.0%
apsi	179.0	179.0	0.0%
fma3d	139.0	133.0	-4.3%
ammp	140.0	153.0	9.3%
Mean	**169.11**	**167.31**	-1.1%

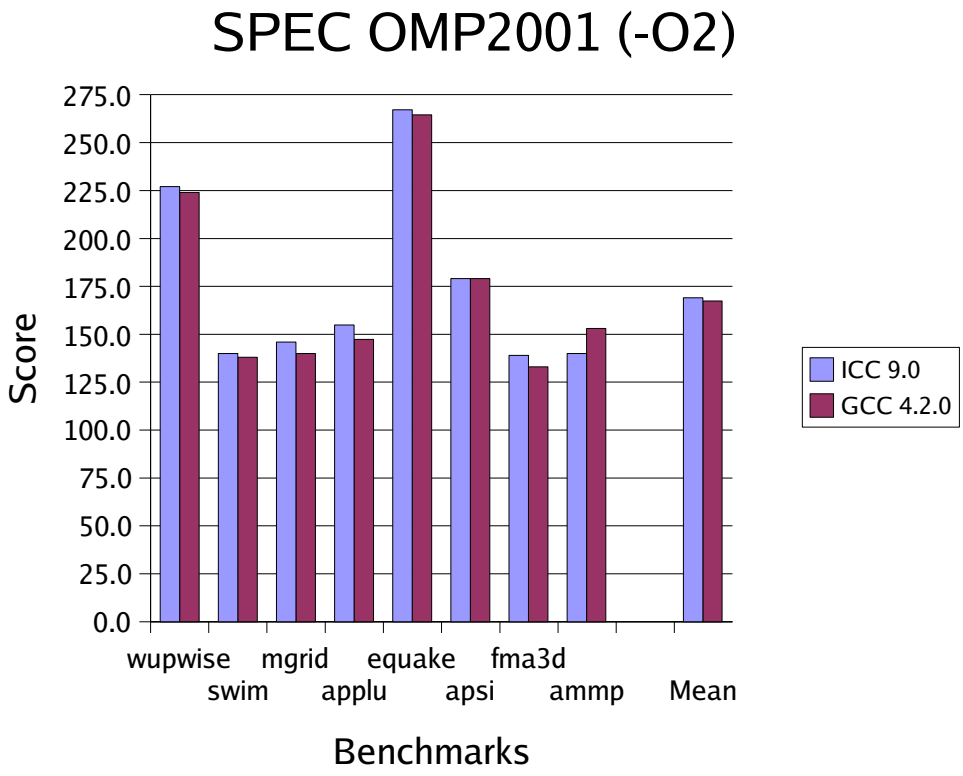


Figure 6: SPEC OMP2001 scores for GCC and ICC on a dual processor EM64T. Higher scores indicate better performance.

6 Implementation Status

At the time of this writing, the implementation is feature complete for all the three languages defined in the standard and scheduled to be released with GCC 4.2. It has also been ported to the GCC 4.1 version included in Fedora Core 5.

The focus over the next few months will be bug fixing and performance tuning. No firm plans exist yet for an auto-parallelization pass as described in the previous section, but it should not be an exceedingly complex project to implement.

To assess the performance of the code generated by GCC, I used SPEC OMP2001 on a dual processor Intel EM64T at 3.4Ghz with 2Gb of RAM, running Fedora Core Linux 3. The compilers tested were GCC v4.2.0 20060406 (experimental) and ICC v9.0 20050914.

As shown in Figure 6, the performance differences between the two compilers are negligible. GCC has a slight edge in some tests and vice versa, but the geometric mean is almost identical.

Both compilers used the standard `-O2` optimization level. Note that the goal was to get a rough idea on how the GCC implementation compares to other compilers. This was not a valid SPEC run as neither GCC nor ICC were able to run all the benchmarks without errors. GCC failed to execute `gafort` and `art`, while ICC failed to build `galgel` and failed to execute `gafort`. Tests that failed in either compiler were taken out of the chart.

References

[1] OpenMP Architecture Review Board. Openmp application program interface v2.5. May 2005. `http://www.openmp.org/drupal/mp-documents/spec25.pdf`.

Autovectorization in GCC—two years later

Dorit Nuzman
IBM Haifa Research Lab - HiPEAC Member
dorit@il.ibm.com

Ayal Zaks
IBM Haifa Research Lab - HiPEAC Member
zaks@il.ibm.com

Abstract

The first version of auto-vectorization was contributed to the GCC lno-branch on January 1st, 2004. Later that year it was presented at the second GCC summit, featuring basic capabilities and preliminary experimental results on PowerPC970. Since then, the vectorizer has made a long way, starting from its acceptance to the GCC 4.0 release, and gradually increasing its applicability both in terms of the application domain it can address and the range of platforms it can target.

This paper overviews the evolvement of the vectorizer in the past two years. This includes support for pointer based and unaligned references, conditional operations, reductions, special idioms, type conversions, and a novel generic vectorization of accesses with power-of-2 strides. Some of these features required new abstractions to express vector operations. It took a collaborative effort to devise abstractions that are general enough, applicable to existing architectures, and fit GCC conventions. This collaboration yielded a vectorization scheme that balances the conflicting needs of different platforms while efficiently supporting each individual target. This is the most comprehensive effort that considers the multi-platform aspect of vectorization, demonstrating applicability on diverse SIMD platforms by one compiler. We also present experimental results on a wide range of key kernels and on several different SIMD platforms, and conclude with directions for future work.

1 Introduction

In early June 2004 we introduced the initial implementation of GCC's automatic vectorization optimization at the second annual GCC Developers Summit. At that stage, the vectorizer was still in its infancy on a development branch and was capable of handling simple constructs only: loops with a single basic block that contain unit-stride accesses to aligned memory locations, all accessing data types of the same size, and no loop-carried dependencies. The vectorizer has enhanced constantly in the past two years starting from its acceptance into GCC mainline version 4.0, followed by new features including vectorizing reduction operations (GCC 4.1), reduction pattern recognition (committed to GCC 4.2), and additional enhancements to support non unit stride accesses and multiple types (submitted

to GCC 4.2). In this paper we describe these enhancements and show how collectively they facilitate vectorizing important multimedia kernels. In concert with these functionality enhancements, the vectorizer has also been extended to support many different vector targets [7]. Continued development of the vectorizer takes place on the autovect development branch; the interested reader is referred to the Free Software Foundation website [2] for access to source files. For general background on automatic vectorization, the reader is referred to our previous paper [6].

The main achievements of the vectorizer over the past two years are in three areas:

More loops can now be vectorized, thanks to efficient support for alignment, pointer accesses, conditional operations, reductions, pattern recognition, multiple types and non unit strided accesses.

More platforms are now supported, due to collaborative design of new vector abstractions.

Real performance improvements can now be obtained by vectorizing real world code for a multitude of SIMD platforms.

These achievements provide the first industry-strength framework capable of considering the multi-platform aspect of vectorization [7]. Many users are exercising the vectorizer (see, e.g., [4]), whose patches are now part of official GCC releases (or are pending approval for inclusion in future releases), thereby helping to reveal various issues [5].

The paper is organized as follows. Section 3 explains the major tradeoffs that we faced when introducing new idioms to the intermediate representation of GCC. In section 4 we describe ability of the vectorizer to handle loops with reduction operations which involve loop-carried dependencies. Section 5 shows how the basic infrastructure was seamlessly extended to support non-unit stride accesses to memory, recognizing and exploiting spatial locality efficiently. In section 6 we describe the ability of the vectorizer to handle loops with accesses to multiple data types and sizes, effectively requiring a restricted form of unrolling. Section 7 provides experimental results that exercise the new capabilities and support for different SIMD targets. We present our conclusion in Section 8.

2 Back to the Future

The starting point for this paper is the directions for further development projected two years ago [6]. These were organized into four categories:

*"**Support additional loop forms**. Support for unknown loop bounds and if-then-else constructs is nearly complete. The major remaining restriction on loop form is the nesting level. Vectorization of nested loops will be considered in the future."*

Indeed, the vectorizer now handles counted loops with arbitrary bounds using loop peeling (contributed by Olga Golovanevsky), based on a generic utility to compute number of iterations (contributed by Sebastian Pop and Zdenek Dvorak). An if-conversion pass is also in place prior to vectorization to collapse simple if-then-else constructs (contributed by Devang Patel). Further extension to the if-converter are currently under development, including the handling of loads/stores and collapsing multiple if-then-else constructs. Vectorization of nested loops is yet to be addressed.

*"**Support additional forms of data references**. Potential extensions in this category include en-*

hancements to the dependence tests (as discussed in Section 5) and support for additional access patterns (reverse access, and accesses that require data manipulations like strided or permuted accesses). Exploiting data reuse as in [9] is an optimization related to data references that we plan to consider in the future."

The major enhancement in this category is the support for additional access patterns in the form of non unit strided accesses whose stride is a power of 2, which has been submitted to mainline and is pending review (for inclusion in GCC 4.2; available on the autovect branch). Handling other access patterns such as reverse or permuted accesses has not been addressed yet. The dependence tests have been enhanced to consider dependencies of distance greater than the vectorization factor (as discussed in the above mentioned Section 5), and there have been enhancements to the generic dependence-tests engine (e.g., Omega tests, contribution of Sebastian Pop). The dependence test used by the vectorizer now handles pointers references as well, in addition to the originally supported array references. Using loop distribution to resolve dependencies has not been addressed yet, and neither has the issue of exploiting outer-loop related reuse.

*"**Support additional operations**. Vectorization of loops with multiple data-types and type casting is the first extension expected in this category. This capability requires support for data packing and unpacking, which breaks out of the one-to-one substitution scheme, and cannot be directly expressed using existing tree-codes. The next capabilities to be introduced will be support for vectorization of induction, reduction, and special idioms (such as saturation, min/max, dot product, etc.), using target hooks or adding new tree-code as necessary."*

Much progress has been achieved in this category. Specifically, support for reduction operations (including regular summation, min/max) has been incorporated into GCC 4.1, and support for reduction patterns including dot-product and widening summation has been committed to mainline (for GCC 4.2). Patches that handle multiple data-types and type casting are available on the autovect branch and have been submitted to GCC mainline (pending review). Support for induction is under development.

*"**Other enhancements and optimizations**. Two general capabilities that we are planning to introduce are support for multiple vector lengths for a single target, and the ability to evaluate the cost of applying vectorization. This will require some form of cost modelling for the vector operations. Interaction with other optimization passes should also be examined, and in particular, potential interaction with other (new) passes that might also exploit data parallelism. One example could be loop parallelization (using threads). Another example could be straight-line code vectorization (as opposed to loop based), such as SLP [3]."*

This category of 'other enhancements' is again a subject of future work.

In addition to the above categories, the handling of alignment has been improved considerably over the last two years, with the ability to perform loop versioning (contributed by Keith Besaw) and loop peeling for multiple loads/stores known to have the same misalignment value. Some of the major issues involving the vectorizer have been presented in other forums [7] [8]. This paper provides a summary of the work achieved in the last two years, with additional more recent details not conveyed by prior publications, focusing on mature patches that have been incorporated or at-least submitted to mainline.

The capabilities of the vectorizer two years ago relied on the available idioms of GCC's GIMPLE intermediate representation, and were

limited accordingly. The major enhancements mentioned above involve vector operations that were not previously expressible in GIMPLE. One of the key factors to the success of the recent enhancements was the introduction of appropriate vector abstractions to GCC. This required a collaborative effort of different vendors and individuals, in order to come up with abstractions that are general enough, applicable to existing architectures, and comply with GCC conventions. This collaboration yielded a vectorization scheme that is able to balance the conflicting needs that arise from the diverse nature of SIMD architectures while supporting each individual target efficiently, as we explain next.

3 New vector abstractions

Vector operations are generally represented in GIMPLE like scalar operations: the same operation codes are used, but the arguments are of vector type. This is suitable for 'pure SIMD' operations, in which the functionality represented by the operation code (e.g., addition) is performed on each element of the vector. Other vector operations like reductions, alignment-support mechanisms, and vector element shuffling operations are meaningless in the context of scalar computations and are therefore unavailable in GIMPLE. In order to express these mechanisms in the vectorized GIMPLE IL, we introduce some new high-level platform-independent abstractions during the last two years.

The general issues and considerations involved in introducing new vector operations to GCC and the GIMPLE IL are discussed in detail in [7], especially in the context of alignment and reduction mechanisms. These considerations include: (1) weighing the benefits of compound abstractions with the advantages of using simpler more basic abstractions; (2) balancing the generality of abstractions with the applicability of existing architectures; (3) considering existing GCC conventions, and; (4) performance considerations — striving to use abstractions that translate into the most efficient instructions available on targets.

For example, consider the issue of non unit stride accesses. Most SIMD architectures provide access only to contiguous memory items, from base addresses that are aligned on a natural vector size boundary. Computations, on the other hand, may access data elements in an order which is neither contiguous nor adequately aligned. Special data reordering mechanisms are provided to help cope with such situations. These mechanisms usually involve additional memory accesses and shuffling instructions for combining data elements from different vectors. The vectorizer must be aware of the available data reordering mechanisms in order to determine whether vectorization of a given computation on a given platform is possible and profitable. It also needs to generate code that correctly and efficiently accesses data located at disjoint and potentially unaligned memory addresses.

We therefore need to abstract low-level data shuffling and alignment handling constructs, and express them in the GIMPLE IL. However, these data shuffling mechanism often differ widely from one SIMD platform to another, posing a challenge to formulate adequate abstractions. A similar situation involving data shuffling arises when vectorizing computations that contain type conversions. In such cases, data elements that reside in one vector need to be expanded and placed in two or more vectors, and vice-versa.

Data shuffling can be accomplished using a general "`permute`" operation, which selects an arbitrary set of elements from two vectors (possibly restricted to only one) and packs

them in one vector. Most platforms, however, do not support such a powerful permute idiom in its most general form. Only the AltiVec `vperm` instruction allows arbitrary, variable permutation. Other platforms (MMX, SSE, MIPS, IA-64 and SPE) either have instructions to merge the high or low halves of two vectors, or something akin to the AltiVec `vperm` that accepts fixed (compile-time constant) permutations. This is a perfect example of the conflict between the desire to introduce general, powerful abstractions and the need to consider what the target platforms actually support. Instead of using a general permute abstraction, the different flavors of data shuffling mechanisms are more appropriately addressed in GCC via simpler, specialized idioms that better match the available technology: the `vec_extract_even` and `vect_extract_odd` abstractions extract elements at even/odd indices, respectively, from two vectors, treated as one stream of elements; the `vec_interleave_hi` and `vec_interleave_lo` abstractions extract the high/low-order elements of two vectors, respectively, and merge them together. These simple abstractions are supported efficiently on most SIMD targets, and provide the means to handle power-of-2 strided accesses [8], as explained in Section 5.

A similar case of data shuffling is related to handling accesses to unaligned addresses, where data elements are to be extracted from two vectors according to the misalignment of the address. Here too, instead of using a general permute abstraction, we created the specialized `realign_load` idiom [7] which takes three arguments: two vectors and a `realignment_token`. The `realignment_token` can be an address, a bit mask, a vector of indices, an offset, or anything that can be generated as a function of the respective address. The `realignment_token` hides low-level details, allowing each target to express its best alignment handling capabilities thereby keeping the `realign_load` idiom general enough yet not too general.

4 Reduction

The basic support for vectorizing loops with reduction operations is provided in GCC version 4.1. Enhanced support for more complex reduction patterns was committed to mainline and will become part of GCC version 4.2.

4.1 Basic reduction

A loop containing a basic reduction operation is depicted in Figure 1(a), where `op` is an associative and commutative operation (such as addition or min/max) which creates a cross iteration data flow dependence cycle, formed by the reduction variable (`a` in the example). If there are no other uses of `a1` and `a2` in the loop, and if it's ok to change the computation order, the vectorizer transforms code by having the loop compute a vector of partial results followed by a loop epilog which reduces this vector to the desired scalar result, thereby altering the order of operations. This is shown in Figure 1(b), where the loop executes parallel independent `op` operations (`vop`) and the loop epilog (labelled `loop_exit`) reduces the vector of partial results using a `reduc_op`, from which the final scalar element desired is extracted (`bit_field_ref`).

Specific details involving epilog code generation and accumulator initialization are described by Nuzman and Henderson [7].

(a) scalar :

```
loop:
      a1 = phi <a0, a2>
s1:   x = ...
s2:   a2 = op <x, a1>

loop_exit:
      a3 = phi <a2>
s3:   use <a3>
s4:   use <a3>
```

(b) vector :

```
loop:
      va1 = phi <va0, va2>
vs1:  vx = ...
vs2:  va2 = vop <vx, va1>

loop_exit:
      va3 = phi <va2>
vs3:  va4 = reduc_op <va3>
vs4:  a5=bit_field_ref<va4,0>
s3:   use <a5>
s4:   use <a5>
```

Figure 1: Basic reduction

4.2 Reduction patterns

Vector targets often have efficient support for complex operations that involve reduction operations. For example, a target may provide an instruction that both multiplies two vectors of multipliers and adds (some of) the products together, to support efficient dot product computations. This example also helps to deal with multiple types (see Section 6), as fewer results of a wider type need to be recorded. It is therefore important for the vectorizer to detect when such complex operations can be applied.

We implemented a pattern recognition engine (vect_pattern_recog) for this purpose, which has been committed to mainline (planned for GCC version 4.2). This engine searches for a sequence of statements that follows a certain idiom; if such a sequence is found, a new 'pattern' statement `ps` representing the idiom is added before the last statement of the sequence `ls` (with cross links between the two), and `ls` is marked as 'in_pattern'. Later on, if we reach `ls` during the bottom-up scan to mark statements relevant for vectorization (vect_mark_relevant), we mark the new `ps` statement instead of `ls`. This way we detect if all intermediate statements of the sequence are used only by `ls`, and if so vectorize only `ps`.

```
sum0 = phi <init, sum1>
dx = (type1) x;
dy = (type1) y;
dprod = dx * dy;
[dprod = (type2) dprod;]
sum1 = dprod + sum0;
```

Figure 2: Reduction pattern example: dot product

The original support for reduction operations handles single statements of the form `a = op <x, a>` having two arguments, where the first (`x`) is defined in the loop and the second (`a`) is the reduction variable defined by the loop-header ϕ, and both have the same type. The extension to handle reduction patterns considers more than one `x` argument defined in the loop, keeping the last `a` argument as the reduction variable, and allowing the type of `a` to be wider than the types of the `x` arguments. This extension captures the dot-product and widening-summation reduction patterns.

The pattern for dot product, for example, is given in Figure 2, where `dx` is double the size of `x`, `dy` is double the size of `y`, `dx`, `dy`, `dprod` all have the same type, `sum` is the same size of `dprod` [or wider], and `sum` has been recognized as a reduction variable. The pattern statement for such dot product patterns is denoted by a new DOT_PROD_EXPR tree-code taking three arguments `<x,y,sum0>`. It is equivalent to a WIDEN_MULT_EXPR statement of `<x,y>` (see Section 6) followed by a regular PLUS_EXPR or a WIDEN_SUM_EXPR state-

ment.

When vectorizing reduction patterns that involve multiple types, care must be taken to select the appropriate types in the final loop epilog code; inside the loop we use statements with complex tree-codes such as DOT_PROD_EXPR, but at the epilog we may use the simple PLUS_EXPR statements to combine the partial results together, using both the wider type and the tree-code of the original scalar reduction operation. This is contrary to simple reductions in which the types of all arguments (including that of the reduction variable) are the same, allowing us to use the same vector type and tree-code for the epilog code and for the code inside the loop.

5 Non Unit Stride Accesses

Work on vectorizing loops which access data in a non-consecutive strided pattern has been presented recently by Nuzman, Rosen and Zaks [8]. This work has been submitted to GCC mainline for inclusion in version 4.2, and is pending review (at the time of writing; contribution of Ira Rosen). The challenge in vectorizing accesses to non-consecutive addresses is twofold: first, the appropriate sets of loads/stores and extract/interleave instructions have to be established, and second, spatial reuse opportunities should be identified and exploited. In terms of the underlying infrastructure, a new type of analysis which groups together independent statements was devised to represent spatial reuse partners. In contrast, previous analyses dealt with individual data references, dependent statements or the entire loop.

Despite the above challenges, integrating the support for non-unit stride accesses with the existing framework was pretty smooth. We extended the existing dependence resolution traversal over pairs of load/store statements, to construct groups of interleaved loads or stores that have the same stride and adjacent base addresses. The data is then provided to each group using a set of loads and extract_even/odd operations, or a set of interleave_low/high operations and stores. This mechanism supports strides that are a power of 2, which are the more common strides and are efficiently handled on most platforms using these abstractions.

For example, if we have a group of four loads with stride four and consecutive base addresses, and VF=8, we will generate the following instructions: a set of four vector loads that load consecutive elements into vectors (0...7), (8...15), (16...23), (24...31) followed by a set of four (intermediate) extract_even/odd operations that produce the vectors (0,2...14), (16,18...30), (1,3...15), (17,19...31) and a set of four (final) extract_even/odd operations that produce the desired vectors: (0,4...28), (1,5...29), (2,6...30), (3,7...31). The case of strided stores is analogous, using interleave_lo/hi instead of extract_even/odd. Additional examples and details are available in previous publications [8, 1].

6 Multiple Types

When the computations inside a loop operate on data-types of different sizes, the vectorization scheme can no longer be a simple 'strip-mine by a single vectorization factor and replace 1-to-1' [6]. This is because different operations prefer different vectorization factors. In general, we set the vectorization factor according to the smallest data type in the loop; operations that operate on larger data types will be replicated similar to loop unrolling. Peeling the loop to align accesses was also updated to

consider multiple data types; the original misalignment values in byte units had to be augmented with the corresponding data type size, to deduce the correct number of elements to handle in the loop prolog.

The overall transformation scheme is as follows. First, as mentioned above, we set the vectorization factor (VF) to the largest value according to the smallest data type in the loop. Then when we vectorize a statement that operates on a data-type of which VF elements cannot fit in one vector word, multiple vector statements are generated to compute VF results. For example, say a loop contains (1 byte) chars and (4 byte) ints, and the vector size (VS) is 16 bytes; the VF in this case will be 16 due to the operations on chars. Each scalar statement that deals with chars is vectorized as usual by replacing it with its vector counterpart. In contrast, each scalar statement that deals with ints is replaced by four vector statements (denoted VS1.0, VS1.1, VS1.2, VS1.3) that together compute 16 results in one iteration of the vectorized loop (see Figure 3). This is because only four ints fit in one vector word. When we continue and vectorize the statement that uses the value defined by S1 (denoted S2), each of the 4 vector statements that we generate (denoted VS2.0, VS2.1, VS2.2, VS2.3) will use the respective vector value defined by VS1.0, VS1.1, VS1.2, VS1.3. In order to be able to find these 4 vector definitions, we chain the vector statements together (via the `RELATED_STMT` field of the `stmt_info` struct: VS1.0→ VS1.1→ VS1.2 → VS1.3. For this reason a `stmt_info` struct is now created for the newly generated vector statements as well). The changes to the vector transformation routines therefore mainly consist of adding a loop around the original vector-statement-creation code to create multiple vector statements per scalar statement, and perform the bookkeeping described above to chain together these vector statements.

(a) scalar :

```
S1:     x = memref
S2:     z = x + 1
```

(b) after vectorizing S1 :

```
VS1.0:  vx0 = memref0
VS1.1:  vx1 = memref1
VS1.2:  vx2 = memref2
VS1.3:  vx3 = memref3
S1: x = memref
S2: z = x + 1
```

(c) after vectorizing S2 :

```
VS1.0:  vx0 = memref0
VS1.1:  vx1 = memref1
VS1.2:  vx2 = memref2
VS1.3:  vx3 = memref3
S1: x = memref
VS2.0:  vz0 = vx0 + v1
VS2.1:  vz1 = vx1 + v1
VS2.2:  vz2 = vx2 + v1
VS2.3:  vz3 = vx3 + v1
S2: z = x + 1
```

Figure 3: Multiple types: simple example

We now describe several issues that arise for certain kinds of computations operating on multiple data types. Many of these issues are related to loop unrolling in general, but are raised here in the context of vectorization.

6.1 Multiple load/stores

When vectorizing a load/store statement and replacing it with multiple vector load/store statements, one issue that comes up is the creation of the vector pointer: we always create a single vector pointer per scalar load/store that we vectorize; all copies of the corresponding vector loads/stores that we generate use the same vector pointer, which is bumped by VS bytes between each pair of consecutive vector loads/stores.

An alternative approach is to bump the mutual vector pointer only once in each iteration of

the vectorized loop, and use a different offset for each of the multiple vector load/store statements involved. It is possible for a subsequent pass to introduce such multiple offsets (which may require more registers) in order to reduce dependencies.

6.2 Multiple reductions

One issue that comes up when generating multiple vector statements for a scalar reduction statement, is how to combine the multiple accumulators. Suppose we vectorize a summation of ints in a loop that also operates on chars, and the VS is 16 bytes. We generate four vector statements to add 16 int elements in each iteration of the vectorized loop. One option is to use four independent accumulators, and combine them at the loop epilog (Figure 4(b)); another option is to have each of the four vector statements feed the next, effectively using a single accumulator (Figure 4(c)).

We chose to implement the latter option of the single accumulator primarily because it uses fewer registers. A subsequent accumulator-expansion pass could in the future replace the single accumulator with multiple accumulators, similar to GCC's current modulo-variable-expansion optimization in the loop unroller (except that in our case we unroll the relevant statements ourselves). In contrast, the converse operation of collapsing multiple independent accumulators feeding a final reduction at the loop epilog into a single accumulator, primarily to save registers, seems more difficult to recognize and realize.

(a) scalar :

```
x = memref
z = z + x
```

(b) multiple vector accumulators :

```
loop:
  vz0 = phi (init0, vz0)
  vz1 = phi (init1, vz1)
  vz2 = phi (init2, vz2)
  vz3 = phi (init3, vz3)
  vx0 = memref0
  vx1 = memref1
  vx2 = memref2
  vx3 = memref3
  vz0 = vz0 + vx0
  vz1 = vz1 + vx1
  vz2 = vz2 + vx2
  vz3 = vz3 + vx3
epilog:
  vz3 = (vz0 + vz1 + vz2 + vz3)
```

(c) single chained vector accumulator :

```
  vz = phi (init, vz)
  vx0 = memref0
  vx1 = memref1
  vx2 = memref2
  vx3 = memref3
  vz = vz + vx0
  vz = vz + vx1
  vz = vz + vx2
  vz = vz + vx3
```

Figure 4: Multiple types: reduction

6.3 Type demotion

When operations involving different data types are not independent, data is transferred from one type to another. Type demotion refers to

the case where data is transferred from a larger type to a smaller type. In this case, one usually either disregards the excessive bits (modulo demotion) or uses them to perform saturation if needed (saturating demotion). We added new tree-codes to support both options, namely `VEC_PACK_MOD_EXPR` and `VEC_PACK_SAT_EXPR`, and accompanying optabs. The term 'pack' stems from the fact that we can place more elements in a vector word after demoting them.

The current implementation of type-demotion is restricted to 'half-demotion' cases where the wider-type (the type of the arguments) is twice that of the smaller type (the type of the result). Note that most targets provide instructions that directly support half-demotion, where the contents of two source operands are packed into a single destination operand; for this reason, a scalar half-demoting operation may be replaced by a single vector statement (which is fed by multiple vector statements).

6.4 Type promotion

Type promotion refers to the case where data is transferred from a smaller type to a larger type. This often occurs when an operation produces a result that can be larger than its operands, the prominent case being multiplication. In general, type promotion appears as a cast operation; the operands are first casted and then the operation is performed on the wide data type. However, most targets provide vector support that combines certain operations with type promotion. It is important for the vectorizer to make use of such combined operations because of their efficiency and because targets may not support the same operation on wider operands.

In addition to combining operations with type promotion, targets typically provide instructions that produce only a subset of the results, each result being wider than the input arguments of the operation. In some cases, as in the case of widening multiplication, the subset of products may be noncontiguous, requiring subsequent shuffling statements to sort the obtained products according to the original (multiplier) order, if needed. Again, the vectorizer should identify when such shuffling code needs to be generated.

Type promotion involves taking one vector of elements (or several, as in the case of widening multiplication) and producing a vector in which each element is of larger size; the resulting vector therefore typically does not fit in a single vector register, so several vector statements are used to replace a single scalar statement (chained together as described earlier in this section).

We implemented support for both general cast promotion and for widening multiplication. General cast promotions are represented using the existing `NOP_EXPR` tree-code, and are vectorized using new `vec_unpack_hi/lo` idioms. Widening multiplication promotions are represented using the new `WIDEN_MULT_EXPR` tree-code, and are vectorized using the `vec_widen_mult_hi/lo` idioms, which preserve the order of the products. If the vectorizer can prove that the order of the products does not have to be preserved (e.g., when the products are used only to feed a reduction computation) then the target hooks builtin_mul_widen_even/odd are used if available; they produce the products of even and odd elements in two separate vectors.

The current implementation of type-promotion is restricted to 'double-promotion' cases where the wider-type (the type of the result) is twice that of the smaller type (the type of the arguments). Note that most targets provide instructions that directly support double-promotion, where half of the elements in the source

operand(s) are expanded to fill the destination operand; for this reason, scalar double-promoting statement are often replaced by pairs of vector statements.

The following new tree-codes and accompanying optabs were added to express vectorized widening operations. To double-promote the low/high half of a vectors elements, using sign-extension for signed and zero-extension for unsigned data types: `VEC_UNPACK_[LO, HI]_EXPR` To produce double-width products of the low/high half of (signed/unsigned) vector multipliers and multiplicands: `VEC_WIDEN_MULT_[LO,HI]_EXPR`.

The `TARGET_VECTORIZE_BUILTIN_MUL_WIDEN_[ODD,EVEN]` target hooks were added, to produce double-width products of odd/even half of vector multipliers and multiplicands. These hooks are used only when we know that the order of products can be altered. We currently detect such situations during the '`mark_stmts_to_be_vectorized`' scan, which was augmented to indicate if a statement is used (only) by reduction operations (in addition to the original indication if it is used in the loop or not).

7 Experimental Results

The results we present in this section exemplify how the enhancements developed in the past two years can be put to work, and successfully vectorize important kernels on a multitude of platforms. We generated the results automatically using the autovect-branch which contains the enhancements of GCC 4.1 and those submitted to GCC 4.2, available from [2]. Experiments were performed on an IBM PowerPC970 processor with Altivec, an AMD Athlon processor with SSE2, an Intel PentiumD (dual-core Pentium 4) processor with SSE2, an Intel Itanium2, and a MIPS64 instruction-accurate simulator with paired-single-fp support. For the AMD/Intel and MIPS platforms, the measurements as well as the required platform-specific development were performed by Richard Henderson and Chao-Ying Fu, respectively.

Name	Description
saxpy_fp	constant times a vector plus a vector
sdot_fp	dot product of two vectors
vecsum_fp	sum elements of a vector
vecmax_fp	find maximum over elements of a vector
cond_replace_fp	copy selective vector elements
add_sat_fp	add two vectors and clip the result
tcpip_checksum_s16	tcpip checksum
audio_dissolve_s16	audio steam fade away
dot_s16	dot product (used in audio FIR filters)
eudist_s16	euclidian distance of vectors (GSM EFR)
linear_comb_s16	linear combination of two vector
vecmax_s16	find maximum over elements of a vector
video_dissolve_u8	image fade away
linear_comb_u8	linear combination of two vector
vecsum_u8	summation of a vector elements
vecmax_u8	find maximum over elements of a vector
i2_cxdot-fp	complex dot product
i4_mixStreams-s16	mix two 4-channel audio streams
i8_cvt_codec-u8	convert rrggbbaa codec to aarrggbb

Table 1: Benchmark description

Table 1 provides a brief description of the kernels used in our experiments. The kernels we use are representative of the main computations in important applications from different domains—linear algebra, video and audio processing, and networking and cover all the features discussed in previous sections (if-conversion, multiple types, reduction, and reduction patterns). The last set of kernels also includes interleaved data (strided accesses), with strides 2, 4, and 8 in `i2_cxdot`, `i4_mixStreams`, and `i8_cvs_codec`, respectively. We report the speedup factors achieved by an automatically vectorized version over the sequential version of the benchmark, compiled with the same optimization flags. Time is measured using the `getrusage` routine (except for MIPS speedups that measure dynamic instruction count instead) and includes any overheads introduced by vectorization. Figures 5–9 summarize the speedup factors obtained for

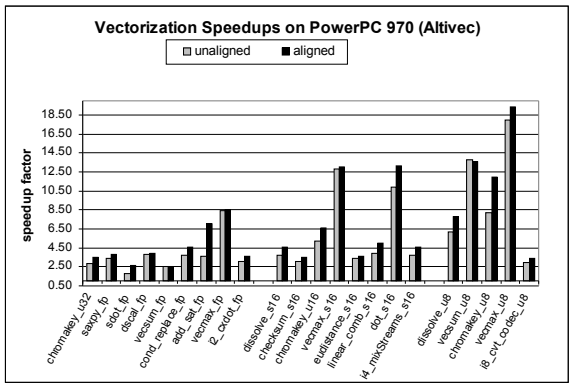


Figure 5: PowerPC970 (Altivec) speedups

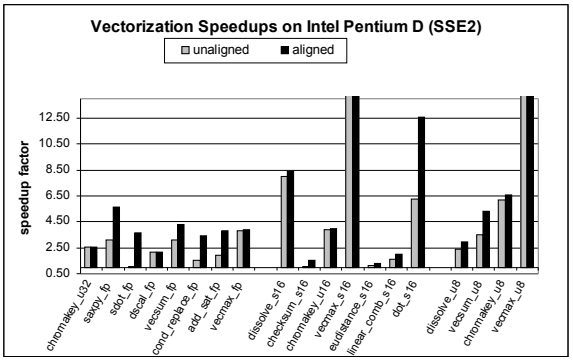


Figure 7: PentiumD (SSE2) speedups

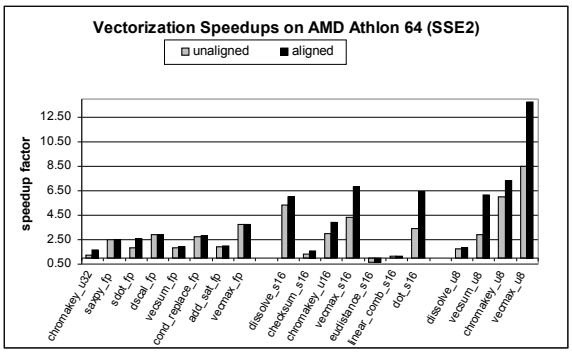


Figure 6: Athlon (SSE2) speedups

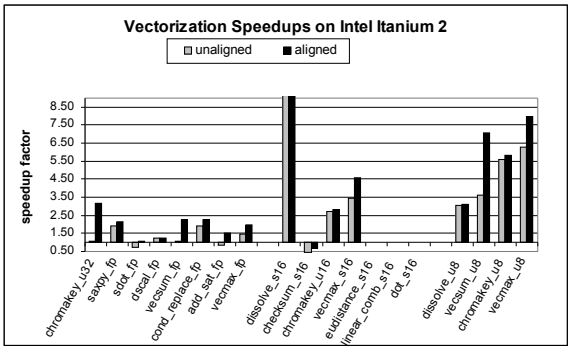


Figure 8: Itanium2 speedups

two versions of each testcase—one in which the alignment of the data is unknown and the other when the data is aligned.

PPC970 Speedups (Altivec): On PowerPC970 we expect an improvement factor between 2 – 4 on floating point benchmarks (one SIMD unit vs. two scalar units), speedups between 4 – 8 on the short benchmarks, and between 8 – 16 on the char benchmarks (minus realignment overhead on the unaligned versions). The speedups obtained are generally within these ranges. Super-linear speedups on dot_s16, vecmax_u8/s16/fp, add_sat_fp, and cond_replace_fp are due to the availability of specialized instructions on the vector unit, while they are absent from the scalar unit

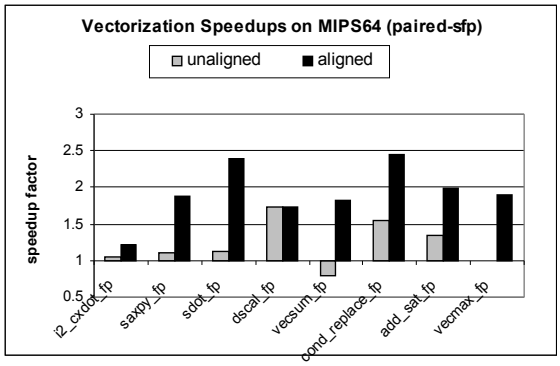


Figure 9: MIPS64 speedups

(`mulsum` (dot), `max`, and conditional instructions). Lower speedups on dissolve_s16/u8, checksum, and linear_comb are due to additional overheads incurred by type conversion and widening multiplication, which take twice as many vector operations to accomplish. Lower speedups on i2_cxdot, i4_mixStreams, and i8_cvs_codec are due to data reordering overheads (usage extracts nad interleaves) to cope with interleaving levels (strides) 2, 4, and 8, respectively.

x86 (Athlon and Pentium) Speedups (SSE2): While the SSE2 architecture supports 128-bit vectors, the current implementations can only operate on 64-bit simultaneously. Therefore, the expected improvement on the benchmarks is between `VF`$/2$ and `VF`, closer to `VF`$/2$ (2 for the fp benchmarks, 4 for the short benchmarks, and 8 for the char benchmarks), minus alignment handling overhead on the unaligned versions. The super-linear speedup on vecmax_s16/u8and dot_s16 is due to inefficient code that is generated for the scalar version.

Itanium2 speedups: We expect an improvement factor of 2 for floats, as the SIMD float instructions run on the same functional units as the scalar float instructions. For integer data, we expect the improvement to be correlated with the number of elements packed in the word size. The high speedup in dissolve_s16 is due to the slower 32-bit integer multiply used in the scalar version as opposed to the parallel 16-bit multiply used for dissolve_u8. The eudist_s16, linear_comb_s16 and dot_s16 do not get vectorized on this platform due to the lack of a 32-bit vector integer multiply.

MIPS64 Speedups: The latencies on the paired-single-fp instructions for MIPS64 are similar to the latencies of scalar instructions, and both share the single fp functional unit. The expected speedup is therefore `VF` $= 2$ (minus the alignment handling overhead). Super-linear improvements on dot_fp and cond_replace_fp are due to inefficient addressing in the scalar code compared to the vector code, and lack of conditional move in the scalar unit, respectively. Lower speedups on i2_cxdot are due to data ordering overheads to handle a strided access.

8 Conclusions

The auto-vectorization optimization of GCC has been enhanced in the past two years in three main areas: (1) adding support to vectorize more complex computations involving in particular reductions, multiple types and non unit strides; (2) extending the GIMPLE intermediate language to represent the desired vector abstractions, through active collaboration that addressed the needs of different platforms and resulted in acceptance into GCC's mainline code base; and (3) applying these stable functional enhancements to important kernels on a multitude of platforms to achieve significant performance improvements. More work lies ahead, including items suggested two years ago and refinements of currently supported features.

9 Acknowledgments

Many people have contributed to the vectorizer over the past two years. We tried to specify major individual contributions throughout the paper. Richard Henderson has reviewed our patches, and together with Ira Rosen, Daniel Berlin, Sebastian Pop, and Zdenek Dvorak have provided continuous support; thanks to all!

References

[1] Free Software Foundation. Auto-Vectorization in GCC,

http://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[2] Free Software Foundation. GCC, http://gcc.gnu.org.

[3] S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 145–156, June 2000.

[4] T. Moene. GFortran: Compiling a 1,000,000+ line NumericalWeather Forecasting System. In *Proc. of the GCC Developers Summit*, pages 159–164, June 2005.

[5] T. Moene. Public and private communication. 2005 and 2006.

[6] D. Naishlos. Autovectorization in GCC. In *Proc. of the GCC Developers Summit*, pages 105–117, June 2004.

[7] D. Nuzman and R. Henderson. Multi-platform Auto-vectorization. In *Proc. of the 4th Annual International Symposium on Code Generation and Optimization (CGO)*, March 2006.

[8] D. Nuzman, I. Rosen and A. Zaks. Auto-Vectorization of Interleaved Data for SIMD In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2006.

[9] J. Shin, J. Chame and M. W. Hall. Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures. In *Proc. of the 11th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 45–55, September 2002.

Speeding Up Thread-Local Storage Access in Dynamic Libraries

Alexandre Oliva
Red Hat and IC-Unicamp
aoliva@redhat.com, oliva@lsd.ic.unicamp.br

Guido Araújo
IC-Unicamp
guido@ic.unicamp.br

Abstract

As multi-core processors become the norm rather than the exception, multi-threaded programming is expected to expand from its current niches to more widespread use, in software components that have not traditionally been concerned about exploiting concurrency.

Accessing thread-local storage (TLS) from within dynamic libraries has traditionally required calling a function to obtain the thread-local address of the variable. Such function calls are several times slower than typical addressing code that is used in executables. While instructions used in executables can assume thread-local variables are at a constant offset within the thread Static TLS block, dynamic libraries loaded during program execution may not even assume that their thread-local variables are in Static TLS blocks.

Since libraries are most commonly loaded as dependencies of executables or other libraries, before a program starts running, the most common TLS case is that of constant offsets. This paper proposes an access model that enables dynamic libraries to take advantage of this fact, without giving up the ability to be loaded during program execution. This new model was implemented and tested on GNU/Linux systems, initially on the Fujitsu FR-V architecture, and later on IA32 and AMD64/EM64T, such that performance could be compared with that of the existing models.

Experimental results revealed the new model consistently exceeds the old model in terms of performance, particularly in the most common case, where the speedup is often well over 2x, bringing it nearly to the same performance of access models used in plain executables.

1 Introduction

As mainstream microprocessor vendors turn to multi-core processors as a way to improve performance[1, 2], the relevance of multi-threaded programming to leverage on such potential performance improvements grows.

Besides the common difficulty multi-threaded programs run into, namely the need for synchronization between threads, it is often the

case that a thread would like to use a global variable,[1] for extended periods of time, without other threads modifying its contents, and without having to incur synchronization overheads.

Using automatic variables to achieve this is a possibility, since each thread has its own stack, where such variables are allocated. However, if multiple functions need to use the same data structure within a thread, a pointer to it must be passed around, which is cumbersome, and might require reengineering the control flow so as to ensure that the stack frame in which the data structure is created is not left while the data is still in use.

Widely-used thread libraries have introduced primitives to overcome this problem, enabling threads to map a global handle, shared by all threads, to different values, one for each thread. This feature is offered in the form of function calls (`pthread_getspecific` and `pthread_setspecific`, in POSIX[3] threads), that are far less efficient than access to global variables and even less efficient than access to automatic variables. Besides the efficiency issues, they are syntactically far more difficult to use than regular variables. These were the main motivations for the introduction of Thread Local Storage (henceforth, TLS[4, 5]) features in compilers, linkers and run-time systems, that enable selected global variables to be marked with a `__thread` specifier or a `threadprivate` pragma, indicating that, for each thread, there should be a separate, independent copy of the variable.

By using custom low-level thread-specific implementations[6], or with cooperation from the compiler and the linker, access to thread-local variables can be far more efficient than using the standard functions that offer abstractions of thread-specific data. In some cases, such as when generating code for dynamic libraries, the compiler-generated code is still very inefficient, for reasons detailed in Section 2; for main executables, access can sometimes be just as efficient as accessing automatic or global variables. The mechanisms introduced in Section 3, based on the novel concept of TLS Descriptors[7, 8], yield a major speedup, that brings the performance of TLS access in dynamic libraries close to that of executables, as shown in Section 4. Section 5 summarizes the results with some final remarks and future directions.

[1] The strictly-correct term here would be variable whose storage has static duration.

2 Background

In this paper, we use the term *loadable module*, or just *module*, to refer to executables, dynamic libraries and the dynamic loader. A process may consist of a set of loadable modules consisting of exactly one executable, a dynamic loader (for dynamic executables) and zero or more dynamic libraries. We call *initial modules* the main executable, any dynamic libraries it depends upon (directly or indirectly) and any other dynamic libraries the dynamic loader chooses to load before relinquishing control to the main executable. Moreover, we use the term *dlopened modules* to refer to modules that are loaded after the program starts running, typically by means of library calls such as `dlopen`.

Every loadable module may define a memory address range delimiting its TLS segment. This range, after relocation processing, contains the memory image to be used to initialize the TLS block associated with that module, for each different thread.

For every thread, two data structures are allocated: a Static TLS Block and a Dynamic Thread Vector (DTV), as depicted in Figure 1.

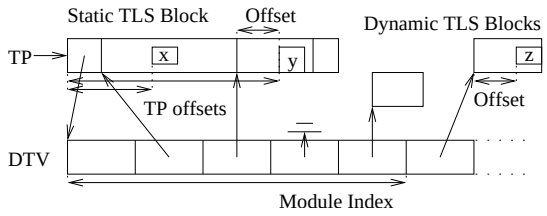


Figure 1: Per-thread data structures used to support TLS.

A reserved register, called the Thread Pointer (TP, for short), points to a base address within that thread's Static TLS Block. At a fixed relative location within the Static TLS Block lies a pointer to the DTV. The DTV, in turn, starts with a generation counter, followed by pointers to TLS Blocks. For every module containing a TLS segment, a module index is assigned, that indicates the entry in each thread's DTV reserved to hold a pointer to the TLS Block corresponding to that module.

The dynamic loader can use information about the TLS segments of all initial modules to lay out the Static TLS Block. Each thread's static block will contain TLS blocks for all initial modules. Using the same layout for all threads implies that the relative locations, in the Static TLS Block, of the initial modules's TLS blocks's are the same across all threads, thus enabling not only efficient code generation for some TLS access models, but also the optimization proposed in Section 3.

2.1 Access Models

If a main executable contains a TLS segment, the dynamic loader not only reserves the first entry in the DTV for it, but also lays out the Static TLS Block in such a way that the offset from the TP to the executable's TLS block is a constant computable at link time. The exact location of the executable's TLS block within the Static TLS Block only depends on the size and alignment requirements of the executable's TLS segment, and conventions set by the Application Binary Interface (ABI) of the hardware architecture and operating system. Since the linker can compute the offset from the TP to the executable's TLS block, and the relative location of a variable defined within this block, it can compute the exact TP offset of such a variable (say, variable `x` in Figure 1), and use that as a displacement from the TP to access the variable. This access model is known as Local Exec. It is the most efficient, but least general, access model, since only the main executable can use it. In theory, all initial modules could use it, but this would require text segments to be modified at dynamic relocation processing time, and modifying text segments is highly undesirable, mainly because it prevents page sharing across multiple processes, which is what shared libraries are supposed to enable.

An example of computing the address of a variable `var` into register `reg` using the Local Exec access model, in low-level pseudo code, is given below. `TPoff` is a functional notation to denote the TP offset of a variable.

let reg ← TP + TPoff(var)

Accessing thread-local variables that are not defined in the main executable preclude the use of the Local Exec access model. The main executable, however, can still take advantage of the fact that every dynamic library it depends on, that might provide the variable it wants to access, is an initial library, and therefore its relative location within the Static TLS Block is a run-time constant, which holds for variables `x` and `y` in Figure 1. Emitting a relocation to get the dynamic loader to compute this run-time constant and store it into a Global Offset Table (GOT) entry, and using this constant, loaded from the GOT, as an offset from the TP

to access the variable, is called the Initial Exec access model. Under certain circumstances, it may be used in dynamic libraries as well, but it may come at the cost of being unable to dlopen such libraries. The use of indirection through the GOT, allocated in the data segment, not only retains the ability to share pages of code, but also merges all the dynamic address computation related with a symbol into a single location, reducing the number of dynamic relocations needed.

An example of computing the address of variable `var` into register `reg` using the Initial Exec access model follows. `GOT`, in such low-level pseudo code, denotes a reserved register or some PC-relative addressing mode that yields the GOT base address. `GOTTPoff` denotes the offset of a GOT entry that, at run time, will hold the TP offset of a variable.

```
load reg, GOT[GOTTPoff(var)]
let reg ← TP + reg
```

The other two access models, General Dynamic and Local Dynamic, require the (implicit) use of the DTV. Both access models involve calling a function, normally called `__tls_get_addr`, to obtain a thread-local address. Function `__tls_get_addr` requires two pieces of information to compute the requested address: a Module Index and an Offset within the module's TLS segment, as depicted in Figure 1 for variable `z`. These two pieces of information are normally computed by the dynamic loader, in response to relocation entries that request them to be stored in the GOT. An example of the use of the General Dynamic access model is given below, using adjacent GOT entries and passing it by reference in a register. Other implementations use independent GOT entries for the two values, and/or pass them by value. `GOTModIdx\&Off` is a functional notation to denote the offset of a GOT entry that, at run time, will hold a Module Index followed by a corresponding Offset.

```
let reg ← GOT + GOTModIdx&Off(var)
call __tls_get_addr
```

Local Dynamic is a variant of General Dynamic that calls the function to compute a base address, normally by passing the function a zero offset. Having obtained the base address of a module's TLS block with a single call, the Local Dynamic access model then uses variables's offsets to access them using the same base address. The offsets can all be computed by the linker, since they are a local property of the module. An example follows, in which `GOTModIdx` denotes the GOT offset for an entry that, at run time, will hold the Module Index and a zero offset, and `ModOff` represents the Offset of a given variable.

```
let reg ← GOT + GOTModIdx()
call __tls_get_addr
let reg1 ← reg + ModOff(var1)
let reg2 ← reg + ModOff(var2)
```

2.2 Dynamic behavior

At thread creation time, the DTV is initialized such that every entry corresponding to an initial module points to a TLS block within the Static TLS Block, like the second and third slots in the DTV in Figure 1, and all other entries are marked as not allocated, like the fourth slot. Entries for dlopened modules have to be assigned on demand to TLS blocks allocated dynamically, as depicted by the two Dynamic TLS Blocks in the figure. Dynamic allocation is necessary because multiple threads may already be running at the time a new module is loaded into a process. Function `__tls_get_addr` is responsible for the run-time maintenance of the DTV.

The generation counter in the DTV is used to keep track of such dynamically-allocated TLS blocks: every time a dlopened module with a TLS segment is loaded or unloaded, a global generation counter is incremented. Function `__tls_get_addr` checks whether the DTV generation counter is up to date every time it is called. If the DTV is found to be out of date, the function may have to release the memory associated with its outdated entries, to dynamically resize it, and to set any released or newly-created entries to the unallocated state.

Once the DTV is up to date, if function `__tls_get_addr` finds that the requested DTV entry is not allocated, it allocates the necessary storage, initializes it with the contents of the TLS segment from the corresponding module and sets the DTV entry to the allocated address. At last, it loads the module's TLS block's base address from the corresponding DTV entry and adds to it the variable offset it was passed as argument, returning the result.

3 Optimization

Let us first investigate why `__tls_get_addr` is perceived as so slow, and then proceed to introducing the optimization subject of this paper.

3.1 Inefficiencies in __tls_get_addr

It might seem that the dynamic access models should not be so expensive, since in the most common case, the run-time behavior of function `__tls_get_addr` will involve two test-and-branch sequences, with branches predicted not taken, followed by offsetting the base address already loaded for the second test by the amount given as an argument, as in the low-level pseudo code below. `DTVoff` denotes the offset from the TP to the DTV address stored in the Static TLS block; `DTVGCoff`, the relative location of the generation counter in the DTV, normally 0; `DTVentrysize`, the size of a DTV entry; `arg1` and `arg2`, the module index and the offset, respectively; `result`, the register in which `__tls_get_addr` returns its result.

```
load reg1 ← TP[DTVoff]
load reg2 ← generation_counter
branch to slow path 1 if reg1[DTVGCoff] < reg2
load reg2 ← reg1[arg1 × DTVentrysize]
branch to slow path 2 if reg2 == UNALLOCATED
let result ← reg2 + arg2
return
```

The first test, however, involves a global variable, the global generation counter. Accessing a global variable can be relatively expensive in such a simple function, since it may require setting up the GOT register to compute its address, if PC-relative addressing is not available.

A bigger performance penalty follows from the compiler's inability to shrink-wrap functions[9, 10], namely, to avoid saving and restoring registers, and even setting up a stack frame, in the fast path that issues no function calls and needs only two scratch registers. Since the slow paths issue function calls, compilers will generally set up a stack frame for the entire function, and since such paths are complex, possibly requiring multiple registers, several such registers have to be saved and restored every time the function is called, even though they are seldom actually used.

Although some register saving and restoring performance can be recovered by means of shrink-wrapping, compilers cannot help the fact that the definition of `__tls_get_addr`, in the dynamic loader, is publicly visible and not actually known before run time, so the compiler must assume it complies with the

platform-defined calling conventions. ABI-defined custom calling conventions for this function could shift into the `__tls_get_addr` slow path the penalties involved with preserving registers that would otherwise have to take place in its callers.

Yet another performance penalty is related with the fact that `__tls_get_addr` is always called through Procedure Linkage Table (PLT) entries. Since it is defined in the dynamic loader, calls to it in other modules have to go through such an entry that loads the actual function address from the GOT and then jumps to it.

Without such inefficiencies, the instruction sequence above would be observed at run time. However, with all the inefficiencies, the dynamic instruction trace after an instructions that calls `__tls_get_addr` is as follows. Additional instructions, not present above, are *emphasized*. `GOToff(sym)` denotes the offset from the GOT to the address of symbol `sym`.

jump to address loaded from PLT GOT entry
set up stack frame
save call-preserved registers used in slow path
save and set up GOT register if needed
load reg1 ← TP[DTVoff]
load reg2 ← GOT[GOToff(generation_counter)]
branch to slow path 1 if reg1[DTVGCoff] < reg2
load reg2 ← reg1[arg1 × DTVentrysize]
branch to slow path 2 if reg2 == UNALLOCATED
let result ← reg2 + arg2
restore registers
destroy stack frame
return

Even if the compiler could be improved so as to avoid setting up a stack frame, the GOT-relative addressing mode to access the generation counter is unavoidable. As for the PLT entry, the additional jump could be avoided by using a call sequence in `__tls_get_addr` callers that referenced its GOT entry directly, precluding lazy relocation of this reference and, most often, requiring larger code size at all call sites, negatively impacting the instruction cache efficiency.

3.2 TLS Descriptors

From the previous paragraph, it would seem that improving the performance of the dynamic access models would not involve a change in the access models themselves, but rather in the compiler used to compile `__tls_get_addr`.

It is possible, however, to make them more efficient, by introducing specialized versions thereof for different situations, and by providing such specialized versions with additional information. Let us put aside for a moment the issue of how to get the most appropriate specialized version selected efficiently, and concentrate on the potential benefits first.

3.2.1 Improving Static TLS

One major shortcoming of `__tls_get_addr` is that it fails to take advantage of the fact that, to access the TLS block for an initial module, no tests are necessary. Since initial modules' TLS blocks are laid out as part of Static TLS Blocks, all threads' DTVs already contain the correct addresses in the entries corresponding to such modules, so it would suffice to dereference the DTV and add the variable offset.

However, it is possible to do even better in the Static TLS case: since the initial module's TLS block is at an offset from the TP that is the same for all threads, we can use the provision above of passing additional information to the specialized function and pass it this constant TP offset, instead of the then-unused module index. Thus, all this specialized function has to do is to add

the module's TP offset to the TP, and then to the variable offset.

In a further step, this specialized function could take as arguments, instead of the TP offset and the variable offset, the precomputed result of adding them together. This specialized function is thus reduced to the following pseudo code:

```
let result ← TP + arg
return
```

Selecting this specialized function reduces significantly the computation performed in the function, rendering its performance very similar to that of the Initial Exec or even Local Exec models, discounting the function call overhead. The use of this specialized version is the most significant improvement we have introduced, but there are additional minor improvements to follow.

One important point to consider is that all specializations must present the same interface, such that callers are totally unaware of which specialization is selected; such selection takes place at run time, at which point it is undesirable to modify code. Therefore, when we modify the interface of a specialization so as to take a single argument, we are either determining that none of the specializations can take more than one argument, or that this one specialization will ignore any additional arguments other specializations might require.

3.2.2 Returning TP offsets

On some architectures, register-plus-register indirect addressing modes is little or no more expensive than indirect addressing modes. On Fujitsu FR-V, for example, there is no single-register indirect addressing mode: loads and stores compute the address by adding a register to either another register or a constant displacement. On IA32 and AMD64/EM64T, on GNU/Linux, segment registers are used as TP, so an instruction with a single-register indirect addressing mode can be modified to use this register as an offset from the segment base address by using a 1-byte prefix, with no significant performance penalty.

On such architectures, it makes sense to arrange for the function to return not the address of the variable, but rather its TP offset. If it is also possible to arrange for the argument to be passed in the register used to hold return values, then the specialization optimized for Static TLS becomes a single return statement, as on FR-V. On IA32 and AMD64/EM64T, it could be possible to achieve the same, but at the expense of additional code at every call site to load the argument from memory. Thus, it is more efficient, in terms of code size, to leave it up to the specialized function to load it before returning.

3.2.3 Linker relaxations

TLS-related relaxations are always defined so as to turn accesses using dynamic access models into Initial Exec or Local Exec, when linking an executable. In general, the `__tls_get_addr` call sequence, including the instructions that set up the arguments, has to contain padding such that, if the linker relaxes the code to a more efficient access model, there is room for the instruction that adds the TP and the TP offset, regardless of whether it is the Local Exec link-time constant or the Initial Exec run-time constant loaded from the GOT.

The convention of returning the TP offset instead of the actual address simplifies linker relaxations, because the addition of the TP does not have to fit in the replacement sequence: it is

already there, after the call sequence. So it suffices to arrange for the value loaded from the GOT, or the fixed constant used in Local Exec, to make it to the register in which the call would have returned the TP offset. With the reduced padding, code size is reduced, improving the efficiency of the instruction cache.

3.2.4 Avoiding unnecessary DTV updates

The use of a global variable, namely the generation counter, when testing whether a DTV is up to date, is not only a bad idea because of the potential performance hit associated with saving, setting up and restoring the GOT register.

The fact that some thread *A* may choose to dlopen or dlclose a module *a* may slow down another thread *B* that accesses TLS variables from module *b*. This occurs because the test in `__tls_get_addr` checks whether the DTV is up to date, and not whether it is recent enough to access a variable in the requested module.

While indexing some TLS module table to determine the generation count associated with a module could be feasible, it would significantly slow down the fast path. However, with our provision of passing additional information to the specialized functions, we can arrange to have the minimum generation count needed to access a module's TLS passed to a specialized function used to handle Dynamic TLS.

Since we have arranged for the Static TLS specialization to use a single argument, we can do the same for the Dynamic TLS specialization at hand. Since there is no way to avoid the requirement for the module index and the offset, however, in order to fit all this information in a single argument, the only solution is to use indirection.

Since Dynamic TLS is designed to be the rare case, allocating additional storage for references to such variables is not deemed unacceptable, so what we do here is to arrange for the Dynamic TLS specialization to be passed, as its argument, a pointer to a data structure containing not only the module index and the offset, but also the generation counter needed by the module. The specialized function can thus avoid the need for the GOT register in the fast path, using for the test the generation counter stored in this data structure passed as its argument, also avoiding DTV updates that would not affect its ability to access the requested module.

On Fujitsu FR-V, a particular detail of the ABI[11] required an additional field in this data structure. The ABI requires the GOT register to be set up for a function not by the function itself, but rather by its caller. Since no specializations of TLS calls would require the GOT register in their fast paths, we have arranged for the argument data structure to contain the GOT pointer the specialization may need.

An additional micro-optimization, applied on FR-V, is to arrange for this data structure to contain not the module index, but rather the offset into the DTV where its entry is stored. This saves a shift-left instruction in the fast path of the specialized function, because FR-V does not have an addressing mode that adds an index register multiplied by a constant to a base register.

3.2.5 Specialized calling conventions

The IA32 version of `__tls_get_addr` on GNU/Linux has traditionally used custom calling conventions in that its arguments are not passed on the stack, as usual, but rather on registers. This should also be the case of specializations of this function.

Besides specifications of where arguments are passed and where return values are stored, another important aspect of calling conventions is that of defining which registers a function can modify without preserving (caller-saved or call-clobbered), and which have to be saved before they can be modified (callee-saved or call-preserved).

The most common TLS cases in code compiled for dynamic libraries, namely Static TLS specialization and relaxation for main executable, can assume that, in a TLS call instruction or its replacement, no register is modified other than the one holding the resulting address or TP offset.

Only the Dynamic TLS specialization needs a pair of temporary registers for the fast path, and potentially several other registers for the slow path.

Since in this work we are defining a new interface for `__tls_get_addr` specializations, we might as well define the conventions regarding preserved registers to privilege the most common cases. We have thus defined that the specializations are to preserve all registers other than the return value, such that TLS calls can be modeled like simple loads, enabling the full register set to be used without concerns about preserving registers across such calls. This requires that, when the slow path of the Dynamic TLS specialization issues calls to other functions, it preserves all registers that they might modify. Since it is the slow path, and it has so much work to do anyway, this additional work is insignificant. Unfortunately, this decision also affects the fast path, in that it has to preserve the two scratch registers it needs, but since Dynamic TLS is assumed to be the uncommon case, privileging the Static TLS case is a reasonable decision.

3.2.6 Selecting specializations at run time

Now that we have established that both specializations work with a single argument, and defined that they should use customized calling conventions to do their jobs, we are ready to specify how the appropriate specialization is to be selected and called.

In the existing dynamic access models, two GOT entries are needed to hold the arguments to `__tls_get_addr`. Since for the specialized versions we can use only one, we can use the other to hold the address of the specialized function. Then, we arrange for the code, that used to call `__tls_get_addr`, to call the function whose address is stored in that location.

As a general rule, we can store the function address at the GOT entry that would, in the traditional access model, contain the module index, and the argument to the function, in the GOT entry that would contain the variable offset. Since, for a given module, the decision on whether its TLS block can be accessed with the Static or the Dynamic specialization is the same for all variables in the block, this general rule works even for ABIs that enable the module index and the variable offset to be in non-adjacent entries, with potential use of the module index entry to access multiple variables.

The machines on which the new access model was implemented, however, all use adjacent GOT entries, since they make the code much simpler, at the expense of additional GOT space due to the multiple copies of the the same module index. Nevertheless, the absence of such sharing enables lazy processing of relocations, as detailed in Section 3.2.8. When the entries are adjacent, they form a data structure that we call TLS Descriptor, named after Function Descriptors, present in ABIs such as IA64's[12], PPC64's[13] and FR-V's[11], that contain a

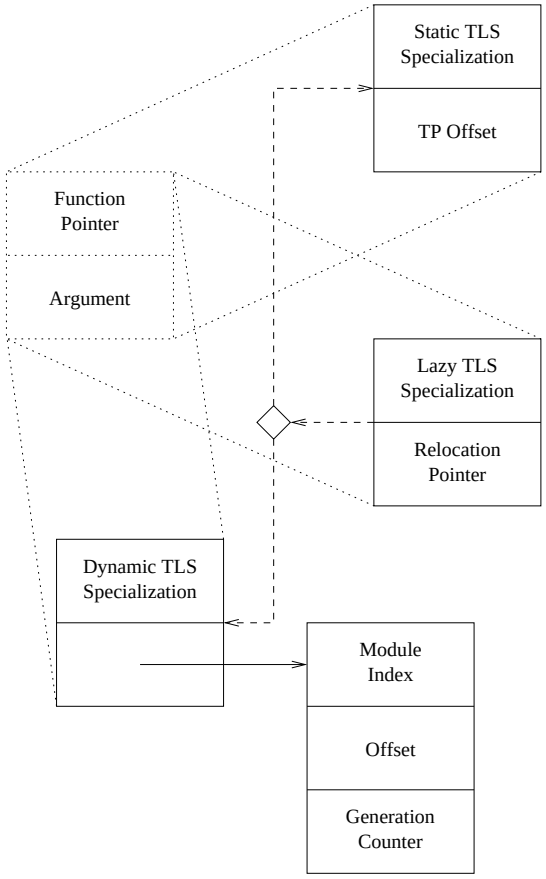


Figure 2: General structure of a TLS Descriptor, with 3 different specialization types, for Static and Dynamic TLS, and Lazy TLS that decays to one of the other two on the first use.

function's entry point and a context pointer, e.g., the GOT pointer to be used by the function. TLS descriptors also take two words, but, instead of a context pointer, their second word contains an argument to the function whose pointer is in the first word, as depicted in Figure 2.

Fujitsu FR-V has never had a traditional TLS ABI, since it was already designed taking advantage of the new access model, but we can imagine that, if it had, the instruction sequence would be as follows.

```
sethi.p #gottlsgdhi(var), gr8
setlo #gottlsgdlo(var), gr8
ldd #tlsgd(var)@(gr15, gr8), gr8
call __tls_get_addr
```

The `ldd` instruction loads into the pair of registers starting at `gr8` the pair of words starting at the address obtained by adding `gr15`, the GOT pointer, and `gr8`, whose value was set to the linker-computed displacement for the GOT entry containing the module index and the variable offset. In the actual FR-V TLS ABI, the call sequence is as follows.

```
sethi.p #gottlsdeschi(var), gr8
setlo #gottlsdesclo(var), gr8
ldd #tlsdesc(var)@(gr15, gr8), gr8
calll #gettlsoff(var)@(gr8, gr0)
```

The variation here is mainly from relocations that reference a TLS Global Dynamic GOT entry to those that reference a TLS Descriptor GOT entry, and the last instruction, that is a call to a named function in the former, that goes through a PLT entry, and a call to a given address in the latter, that goes straight to the specialization. The address was loaded into `gr8`; `gr0` is fixed at zero.

On GNU/Linux IA32, the difference is a bit more significant. The current TLS ABI specifies the following sequence for the General Dynamic access model.

```
leal var@TLSGD(,%ebx,1), %eax
call ___tls_get_addr@PLT
```

This uses an extraneous addressing mode for `leal`, equivalent to `(%ebx)`, but longer, making the instruction long enough for the relaxation replacement, that takes 12 bytes. Our version, however, is as short as 8 bytes for the call sequence, although it requires an additional byte for the segment prefix to the load or store instruction that uses the resulting offset.

```
leal var@TLSDESC(%ebx), %eax
call *var@TLSCALL(%eax)
```

Note that `var@TLSCALL` is just an annotation to aid linker relaxations, such that the two instructions can be scheduled apart. The actual instruction encoding is the two-byte indirect call, that calls the function at the address stored at the memory location whose address was computed into `%eax` by the `leal` instruction. The called specialization knows that, at its entry point, `%eax` points to the TLS descriptor, so it can load its argument from the descriptor.

On AMD64/EM64T, the original call sequence contains several meaningless padding prefixes to make room for relaxation substitutions, as follows.

```
.byte 0x66
leaq var@TLSGD(%rip), %rdi
.word 0x6666
rex64
call __tls_get_addr@PLT
```

Our improved call sequence follows the very same pattern as IA32, with the difference that GOT accesses do not involve a fixed register, but are PC-relative, and register and addresses are 64-bits wide. While the above takes 16 bytes, the following takes as little as 9, plus one for the byte prefix in actual accesses.

```
leaq var@TLSDESC(%rip), %rax
call *var@TLSCALL(%rax)
```

3.2.7 DTV compression

When this new access model is used, and the traditional one is not (i.e., `__tls_get_addr` is never called directly), it is possible to remove all static entries from the DTV, since they are never used. Since we know that every access to Static TLS will go through the static specialization, that does not use the DTV, entries for such modules can be entirely removed, enabling the initial DTV to be trivially set up.

This offers a slight speed up in thread creation for processes that have multiple initial modules with TLS segments, potentially saves memory by delaying the need for dynamically growing the DTV, and enables the DTV to be reduced by half, since its current definition reserves a word in every entry to tell whether it is static or dynamic when the time comes to release that entry and free up its storage.

Even when the traditional dynamic TLS access model is used, it is possible to enable this DTV compression, as long as the index range reserved for initial modules can be easily distinguinshed from that of dlopened modules, for example, by having the most significant bit set. `__tls_get_addr` would then have to recognize this case and use an alternate code path that, instead of relying on the DTV, obtained the module's constant TP offset from a separate table.

3.2.8 Lazy relocations

Processing relocation entries lazily enables significant speedups in start-up time for applications. The mechanism consists in performing a very quick pass over relocations that can be resolved lazily (something that can be determined by the linker), setting them up such that, only when they are used for the first time are they actually resolved.

This has traditionally been used to resolve function addresses in dynamic linking. A call to a function that does not bind locally (i.e., that may be resolved to a definition in a separate module) is directed to go through a PLT entry,

that loads an address from the GOT and jumps to it.

In the first pass, the dynamic loader sets these GOT entries to point to a stub that calls the dynamic resolver with enough information for it to identify the relocation that it should resolve at that time.

The dynamic resolver applies the relocation, modifying the GOT entry such that subsequent calls go straight to the actual function, and then transfers control to the function that should have been called, as if it had been called directly.

Although lazy relocation processing is very often applied to function calls, it is never applied to data accesses, since there is no transfer of control involved, and introducing it would render the access model too costly in terms of performance.

In our optimized dynamic access model, however, there is a control transfer, and we realized we could use that to enable lazy relocation processing. In the quick pre-relocation pass, the function address in the TLS descriptor is set to another specialization that handles lazy relocation, and the argument is set so as to point to the relocation itself.

When the function is called, it resolves the symbol the relocation refers to, decides whether to use the Static or Dynamic specialization and sets up the TLS descriptor according to the decision, such that subsequent calls involving the same TLS descriptors go straight to the most efficient specialization.

Care must be taken to ensure that the TLS descriptor is never in a state that, should another thread perform an access using it, will yield an incorrect result.

On FR-V, that is not very difficult, since the instructions that read and store a pair of words are atomic given sufficient alignment. On IA32 and AMD64/EM64T, however, there is no instruction that can read or modify a pair of words atomically. Since requiring every call site to use synchronization would be too costly, a solution was devised that requires synchronization only in the lazy relocation function itself.

The lazy relocation specialization first acquires a dynamic loader lock and verifies that the TLS descriptor still points to itself. If so, it modifies it so as to point to a hold function and reads the argument. At that point, it can release the lock and compute the final value of the TLS descriptor, using the argument read while the lock was held.

Before modifying the descriptor, it acquires the lock again, wakes up any threads that might be waiting for it in the hold function (using say a condition variable), finally releasing the lock and transferring control to the function whose address was stored in the TLS descriptor.

The hold function simply acquires the lock and, in a loop, tests whether the TLS descriptor still points to it and, if so, waits on the condition variable until it is signaled, otherwise, it releases the lock and transfers control to the function specified in the TLS descriptor.

A simpler, yet less scalable, alternate design for the hold function, that does not involve condition variables, relies on the lock alone: the lazy relocation function does not release the lock throughout its operation, and the hold function is as simple as acquiring the lock, releasing it and transferring control to the function specified in the TLS descriptor. This design is quite appropriate when the relocation-processing code in the dynamic loader already requires a lock to be held, as it is the case in GNU libc.

4 Performance

Verifying any actual performance improvements provided by the optimizations introduced herein proved to be a major challenge. To the best of our knowledge, the only library that makes heavy use of Thread Local Storage is GNU libc itself. To make matters worse, GNU libc takes advantage of the fact that its dynamic loader and C library are always loaded initially, and thus they use the Initial Exec access model throughout the libraries offered by GNU libc, ensuring that any thread-local variables accessed with this access model are located in one of these two libraries.

Even forcing GNU libc to not use the Initial Exec access model and running the Native Posix Thread Library (NPTL[14]) performance benchmark to evaluate the benefit of the optimization to this benchmark showed no difference whatsoever. Investigation showed that this benchmark called `__tls_get_addr` only a handful of times during a test run that took tens of seconds, so performance differences could not possibly be exposed by this benchmark.

The main reason as to why the thread performance test did not use dynamic access models very often is that, first of all, it did not exercise thread-local storage access itself and, even if it did, it is a main application, not a dynamic library, so dynamic models do not apply. As for the libraries it uses, GNU libc's C and thread libraries maintain information pertaining to threads in the thread's static TLS block, and access it using a model similar to Local Exec, so they are not affected by the choice to not use the Initial Exec model within libc.

Although Gomp[15], the implementation of OpenMP[16] for the GNU toolchain, has very recently become a viable platform for measuring TLS performance, the SPEC OMP2001 benchmark uses `threadprivate` variables in only one of its tests, and even then, not in a dynamic library, so using this benchmark was not viable either, and we were left with the need for creating synthetic microbenchmarks.

We have created a total of 40 tests for our benchmark, such that every test is represented as a function that returns a result that is somehow related with one or more thread-local variables, with variations in 4 different dimensions, described in the following paragraphs.

Operation Half of the tests compute the address of a thread-local variable (**addr**), whereas the other half computes the actual value stored in the thread-local copy of the variable (**load**). This exposes differences related with the efficiency of accessing a thread-local variable without explicitly adding the thread pointer to its relative location. On all tested CPUs, the TP register is a special register whose contents cannot be read or modified from userland. It can be used as a base register to read or modify a thread-local variable, but computing the address of a variable requires loading the register's value from a reserved location in the Static TLS block.

Timing All of the timing is performed using the clock tick counting instructions available on the CPUs we've used for testing. Half of the test functions time their operation by themselves (**Internal**), storing the number of clock ticks elapsed while performing the operation in a pointer passed in as an argument. The other half perform no timing whatsoever, relying on their callers to obtain the clock tick count for the entire call (**External**). Unlike the previous dimension, that intends to expose differences, this one intends to confirm the performance improvements we've achieved, by offering multiple performance measures of different but functionally-similar code.

The confirmation was not straightforward, though; the little room for scheduling in the internal timing variants and the high pressure on the registers used by both the timing instructions and function call return values would create pipeline bubbles that, without care to avoid such worst-case conditions (unlikely to occur in real life), would have made some tests that perform very little work appear to be slower than some that perform much more work.

Access model We have four different kinds of tests in this dimension, in which knowledge about the location of the thread-local variable used varies, plus one kind of test that combines access to multiple variables.

Half of the single-variable tests use Initial Exec access models, but in half of these, the compiler generated Initial Exec code because it was told the variable was in Static TLS (**OIE**, for original IE); in the other half, the compiler was told the variable was in Dynamic TLS, so it generated General Dynamic code, and then the linker relaxed that to Initial Exec, being aware of the Static TLS location of the variable (**RIE**, for relaxed IE).

The other half of the single-variable tests use General Dynamic access models. In half of these, the variable is in Static TLS, so our main optimization kicks in (**SGD**, for static GD); in the other half, the variable is in Dynamic TLS, so the main optimization does not apply (**DGD**, for dynamic GD).

The multi-variable tests (**Cmb**, for combined) subtract the values or addresses of the **RIE** and the **SGD** variables, and adds the value or address of the **DGD** variable, returning the result. All this work grants the compiler more opportunity to hide the latency of certain operations through instruction scheduling.

Local State Half of the test functions are so simple that, when they have to call `__tls_get_addr` or equivalent, any automatic variables of their own can easily be assigned to call-preserved registers, so the optimized calling conventions suggested in this paper show no benefit whatsoever (**Min St**). In order to expose such benefits, the other half of the test functions contain a large number of automatic variables (**Max St**) whose contents are forced into registers before and after the TLS operation, such that, with the standard calling conventions, almost all call-clobbered registers have to be spilled before the call and reloaded after it, whereas with our optimization, none of this takes place.

The number of variables is chosen such that all but one of the general-purpose registers are taken up by these variables. On IA32, we use 5 such variables, considering that `%ebx` is reserved as the GOT pointer, and that `%ebp` can be used as a general-purpose register, making up for 6 available registers, 3 call-saved, 3 call-clobbered. On AMD64, we use 14 such variables, since `%esp` is not really usable in the 16-register set. On both CPUs, we keep one register available to hold the result of the TLS operation, with the explicit intention of showing a worst-case scenario for the traditional code, where the advantages of the custom calling conventions would be greatest. The actual benefit from this change will be somewhere in between the two variants in this dimension.

The 40 combinations of the above variations are all located in a dynamic library that is dlopened by the main benchmark program. This ensures that the test functions do not get inlined into the main benchmark loop, which might enable hoisting of operations, making operations look faster than they are.

We build two such dynamic libraries for each tested architecture: one created with the com-

piler configured to generate code in the traditional way (**Ol**), another following our new proposed method (**Nu**). A full test run goes through all 40 tests for each of the 2 libraries, which makes up for the 80 tests total.

Every test is run a large number of times, in two different configurations. In one configuration, we run each one of them in a tight loop to then proceed to the next test; in the other, each test is run once in a randomized sequence generated for every iteration in a loop. More details are given below.

Although running the tests in a tight loop has enabled us to initially measure a lower bound for the execution time of each test, such lower bound was initially not thought to be very representative of real-life performance, since it depends heavily on hot caches and nearly-perfect branch and call/return prediction, something that is not necessarily expected in practice.

In order to try to obtain more representative results, we collect all of the tests into a vector and then, for every iteration in the main benchmark loop, we get the vector sorted at random and then iterate over the randomized vector, running each test once per iteration in the main loop. Each test run produces a time result that is immediately logged to a file. This logging and randomization helps avoid getting cache, branch and call/return prediction hits too common for any single test, which enables us to achieve moderately reproducible results with thousands of runs of each test, as opposed to hundreds of millions that we needed in the tight-loop test. It often (but not always) gets us identical per-iteration lower bounds, but the average run times no longer tend to the lower bound as the iteration count increases.

Unfortunately, this randomization, and the possibility of long interrupts and context switches that could skew averages up at random, have caused average times over 1 million runs to vary by as much as 30%, even after discarding values that appear to be too high.

That said, in spite of the significant error margin in the exact averages, we've verified that there appears to be a strong correlation between improvements in minimum times, as measured in the tight loop, and improvements in the average times, although speedups tend to be smaller for averages than for minimums.

Given this correlation and the irreproducibility of the exact average results, we've decided to not include the average times in the paper. Since binaries and the complete source code of the implementation, including the benchmark program that can generated them, are available for download at `http://www.lsd.ic.unicamp.br/~oliva/`, publishing only the minimum times, that are perfectly reproducible, was deemed enough.

4.1 Analysis

Testing procedure was as follows. A toolchain was built on Fedora Core 4, based on snapshots of the GCC and GNU binutils development trees taken on Oct 30, 2005. This toolchain was capable of generating code for both IA-32 and AMD64, selecting the old or the new TLS call sequences through a command-line switch. A development snapshot of GNU libc, taken on the same day, was built using this compiler for both IA-32 and AMD64. The IA-32 version was built with optimizations for Pentium II or newer; the AMD64 version was built with default settings. The benchmark program and libraries were built with the same settings.

The benchmark program was run on 3 different environments, each one described in the caption of the corresponding table: a Pentium III processor ran the 32-bit benchmark (Table 1),

Acc Mod	Op	Internal Timing Min St Ol	Min St Nu	Max St Ol	Max St Nu	External Timing Min St Ol	Min St Nu	Max St Ol	Max St Nu
OIE	load	33	33	37	37	48	48	58	58
	addr	33	33	38	38	45	45	55	55
RIE	load	35	35	40	38	50	50	61	60
	addr	34	34	39	37	48	50	62	59
SGD	load	64	39	67	43	77	53	88	64
	addr	63	39	67	43	76	53	87	64
DGD	load	64	58	67	58	77	67	90	78
	addr	63	53	68	58	76	67	87	76
Cmb	load	104	63	108	77	110	78	131	100
	addr	94	64	101	69	113	78	123	90

Table 1: Minimum run times, in CPU cycles, over 100000000 iterations on a Pentium III Speedstep 1.0GHz (32-bit only). The timing overhead, included in the figures above, was measured as 33 CPU cycles.

and an Athlon64 processor ran both the 32-bit (Table 2) and the 64-bit (Table 3) benchmarks. In all cases, the processors were configured to avoid clock speed switching, and the boxes were very lightly loaded, except for the benchmark program. The results were mechanically converted to LaTeX tables.

Figures 3, 4, 5, and 6, also generated mechanically, display information from the **SGD** and **DGD internal-timing** tests in the tables. In each chart, the left cluster of bars is for **Min St** tests; that on the right is for **Max St** tests. Within each cluster, the bars represent each of the tested machines, in the same order that their tables appear. Within each bar, the dotted line represents the timing overhead (see below), the lower bar is the **Nu** time and the upper bar is the **Ol** time. Speedups are computed in each bar; the lower speedup is computed as a fraction of the **Ol** and **Nu** numbers directly from the table, the upper speedup is computed by first subtracting the timing overhead from the dividend and the divisor. The real speedup in practice ought to be between the two figures.

The timing overhead is the difference in the clock tick count between two subsequent ex-

Acc Mod	Op	Internal Timing Min St Ol	Min St Nu	Max St Ol	Max St Nu	External Timing Min St Ol	Min St Nu	Max St Ol	Max St Nu
OIE	load	9	9	10	10	24	24	29	29
	addr	5	5	10	10	21	20	30	29
RIE	load	9	9	17	10	25	25	34	30
	addr	5	5	13	10	21	22	30	29
SGD	load	34	9	40	15	49	29	57	31
	addr	32	9	38	11	44	25	56	31
DGD	load	35	23	40	25	48	38	57	42
	addr	31	18	38	21	46	37	56	40
Cmb	load	76	29	79	39	78	46	98	59
	addr	66	23	68	32	76	42	87	49

Table 2: Minimum run times, in CPU cycles, over 100000000 iterations on an Athlon64 3000+ (1.8GHz) notebook, running the benchmark compiled for 32-bit mode. The timing overhead, included in the figures above, was measured as 8 CPU cycles.

Acc Mod	Op	Internal Timing Min St Ol	Min St Nu	Max St Ol	Max St Nu	External Timing Min St Ol	Min St Nu	Max St Ol	Max St Nu
OIE	load	9	9	9	9	9	9	19	19
	addr	8	8	8	8	9	10	18	17
RIE	load	9	9	22	9	13	9	32	19
	addr	5	8	20	8	9	10	28	16
SGD	load	26	9	37	11	29	15	47	20
	addr	23	9	36	10	28	12	47	18
DGD	load	26	25	37	25	29	26	48	31
	addr	23	21	36	21	28	22	47	31
Cmb	load	47	30	62	39	52	31	72	50
	addr	42	23	59	28	49	27	68	37

Table 3: Minimum run times, in CPU cycles, over 100000000 iterations on the same Athlon64 notebook from Table 2, running the benchmark compiled for 64-bit (AMD64) mode. The timing overhead, included in the figures above, was measured as 5 CPU cycles.

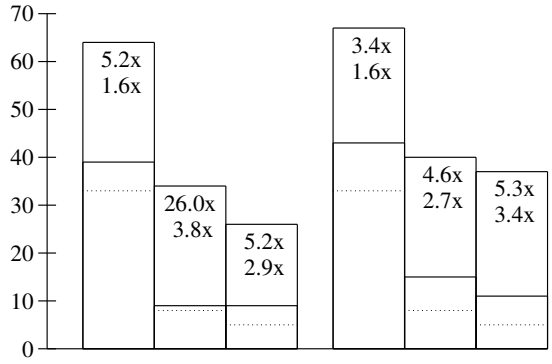


Figure 3: SGD load internal timing results.

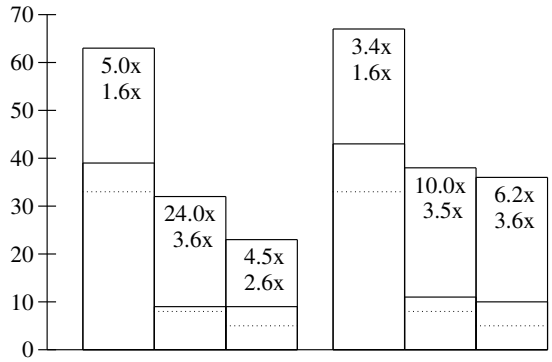


Figure 4: SGD addr internal timing results.

ecutions of the instruction that obtains this count, including the time needed to copy the contents of the first measurement elsewhere before they are overwritten by the second measurement. Careful analysis of the tables shows that the overhead is greater than or equal to the times measured for certain simple operations. Such simple instruction sequences are believed to fit in, or even help avoid additional pipeline bubbles.

OIE tests confirm the expected absence of variation, given that it is the exact same code being generated for both the old and the new TLS conventions.

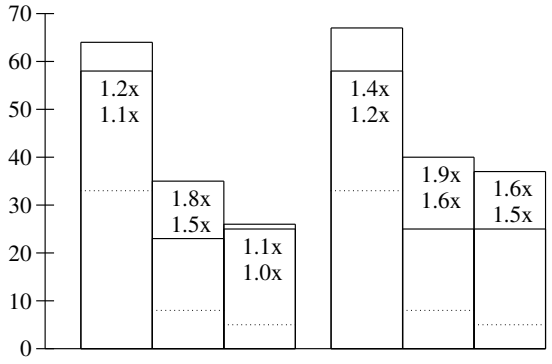


Figure 5: DGD load internal timing results.

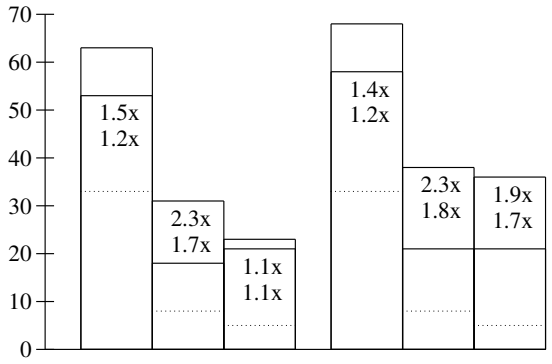


Figure 6: DGD addr internal timing results.

RIE remains nearly identical in terms of performance on IA-32 for the minimum-state tests, as expected. For the maximum-state tests, the new method begins to show improvements, as it enables the compiler to preserve more state across the TLS calls that, in these tests, end up being relaxed, but the advantage remains since the linker cannot recover the performance loss due to register spilling and reloading. The performance loss in the 64-bit minimum-state address RIE probably indicates there might be better instruction sequences we could use for relaxation.

SGD is where the new method really shines. That is no surprise, since it's exactly the situation that the new method is designed to improve, and fortunately also the most common situation in code generated for dynamic libraries that accesses thread-local variables. Absolute reductions in clock cycles are consistent between internal and external timing in 32-bit mode, where the calling conventions optimization plays a less significant role; in 64-bit mode, the absolute reductions in clock cycles are consistent if you compare results among the minimum-state tests, or among the maximum-state ones.

DGD shows that performance is improved significantly even in the case that the new method regarded as the slow case. Clearly, in 64-bit mode, most of the savings stem from the optimized calling conventions, that enable the retention of state in registers, as shown in the comparison between minimum- and maximum-state in the internal timing column, where the new model remains unchanged upon the growth in state and the old model slowed down by a significant amount. In the external timing column, the overhead from having to preserve all callee-saved registers that are used is noticeable in the maximum-state column, but not as much as in the old model. In 32-bit mode, the ability to check whether the DTV is up-to-date without setting up the GOT pointer is likely what brings most of the benefit.

Cmb essentially only confirms the results above, not offering any obvious new insights.

5 Conclusion

The proposed optimization improves performance of access to thread-local variables from dynamic libraries by a big margin for initial libraries, without any data size penalty and most often with code size reductions. For dlopened libraries, there are still performance advantages, but to a lesser, yet still significant extent, and there are data size penalties.

It should be highlighted that the performance gains from lazy relocations, by avoiding relocation processing at load time, and from code size reductions, by improving the instruction cache hit rate, have not been taken into account at all in the micro-benchmarks exposed here.

The implementation is readily available for widely-used CPU types, under Free Software licenses that enable any library to take advantage of this novel technique.

Some open questions remain to be answered in future work: whether there are relaxation sequences that could make the new relaxed code at least as fast as the old one on AMD64, and faster on IA32; whether returning an offset instead of an address from the specialized `__tls_get_addr` calls does indeed help improve performance; whether enabling the specializations to clobber one or two registers, which would enable the dynamic-case fast path to save fewer or even no registers, would cause

a measurable decrease in performance in the more common cases; how much of a performance improvement could have been obtained over the old model by using the same call sequences, and only modifying the run-time so as to compute relocations differently, and modifying `__tls_get_addr` to cope; how much benefit would be obtained by implementing DTV compression; how well the optimizations described here do on other architectures.

Acknowledgements

Alexandre Oliva thanks Glauber de Oliveira Costa, for having reviewed this paper and offered to extend this work to the ARM platform; colleagues Roland McGrath, Jakub Jelínek, Richard Henderson, and Ulrich Drepper for the early discussions on the design, for the support and reviews in designing and implementing the optimizations on various architectures; Aldy Hernandez for tolerating the delays due to the creativity in designing the FR-V TLS ABI, for drafting its early descriptions and for doing the compiler work for that implementation; Eric Bachalo, his manager at that time, for giving his approval to such creativity in a customer-funded project, approval that in the end made this work possible.

References

[1] Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005. `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[2] Kunle Olukotun and Lance Hammond. The future of microprocessors. *ACM Queue*, 3(7):26–34, September 2005.

[3] Portable Applications Standards Committee of the IEEE Computer Society and The Open Group. Portable Operating System Interface (POSIX), The Base Specifications. IEEE Std 1003.1, 2004. Issue 6, Incorporating Technical Corrigendum 1 and Technical Corrigendum 2.

[4] Ulrich Drepper. ELF Handling for Thread-Local Storage. `http://people.redhat.com/drepper/tls.pdf`, February 2003. Version 0.20.

[5] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, October 1999.

[6] Hans-J. Boehm. Fast multiprocessor memory allocation and garbage collection. Technical Report 165, HP Labs, 2000.

[7] Alexandre Oliva and Aldy Hernandez. The FR-V thread-local storage ABI. `http://people.redhat.com/aoliva/writeups/FR-V/FDPIC-TLS-ABI.txt`, December 2004. Version 0.22.

[8] Alexandre Oliva. Thread-Local Storage Descriptors for IA32 and AMD64/EM64T. `http://people.redhat.com/aoliva/writeups/TLS/RFC-TLSDESC-x86.txt`, October 2005. Version 0.9.4.

[9] Fred C. Chow. Minimizing register usage penalty at procedure calls. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 85–94. ACM Press, 1988.

[10] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[11] Kevin Buettner, Alexandre Oliva, and Richard Henderson. The FR-V FDPIC ABI. `http://people.redhat.com/aoliva/writeups/FR-V/FDPIC-ABI.txt`, April 2004. Version 1.0.

[12] Intel Itanium Processor-specific Application Binary Interface (ABI). `http://refspecs.freestandards.org/elf/IA64-SysV-psABI.pdf`, May 2001.

[13] Ian Lance Taylor. 64-bit PowerPC ELF Application Binary Interface Supplement. `http://www.linuxbase.org/spec/ELF/ppc64/PPC-elf64abi-1.7.pdf`, September 2003. 1.7 Edition.

[14] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux. `http://people.redhat.com/drepper/nptl-design.pdf`, February 2005.

[15] GOMP — An OpenMP implementation for GCC. `http://gcc.gnu.org/projects/gomp/`, November 2005.

[16] OpenMP Architecture Review Board. OpenMP Application Programming Interface. `http://www.openmp.org/drupal/mp-documents/spec25.pdf`, May 2005. Version 2.5.

GRAPHITE: Polyhedral Analyses and Optimizations for GCC

Sebastian Pop [1] Albert Cohen [2] Cédric Bastoul [2] Sylvain Girbal [2]
Georges-André Silber [1]
Nicolas Vasilache [2]

[1] *CRI, École des mines de Paris, Fontainebleau, France*
lastname@cri.ensmp.fr
[2] *Alchemy group, INRIA Futurs and LRI, Paris-Sud 11 University, Orsay, France*
firstname.lastname@inria.fr

Abstract

We present a plan to add loop nest optimizations in GCC based on polyhedral representations of loop nests. We advocate a static analysis approach based on a hierarchy of interchangeable abstractions with solvers that range from the exact solvers such as OMEGA, to faster but less precise solvers based on more coarse abstractions. The intermediate representation GRAPHITE[1] (GIMPLE Represented as Polyhedra with Interchangeable Envelopes), built on GIMPLE and the natural loops, hosts the high level loop transformations. We base this presentation on the WRaP-IT project developed in the Alchemy group at INRIA Futurs and Paris-Sud University, on the PIPS compiler developed at École des mines de Paris, and on a joint work with several members of the static analysis and polyhedral compilation community in France.

The main goal of this project is to bring more high level loop optimizations to GCC: loop fusion, tiling, strip mining, etc. Thanks to the WRaP-IT experience, we know that the polyhedral analyzes and transformations are affordable in a production compiler. A second goal of this project is to experiment with compile time reduction versus attainable precision when replacing operations on polyhedra with faster operations on more abstract domains. However, the use of a too coarse representation for computing might also result in an over approximated solution that cannot be used in subsequent computations. There exists a trade off between speed of the computation and the attainable precision that has not yet been analyzed for real world programs.

[1]This work was partially supported by ACI/APRON.

1 Introduction

Static compiler optimizations can hardly cope with the complex run-time behavior and hardware components interplay of modern processor architectures. Multiple architectural phenomena occur and interact simultaneously. The optimizer needs to combine multiple program transformations to harness the computing and

storage resources and to fight all sources of pipeline stalls or flushes. In addition, conventional processor architectures are shifting towards coarser grain on-chip parallelism, to avoid diminishing returns of further extending instruction-level parallelism. This shift rejuvenates the hard static analysis and optimization problems associated with automatic parallelization (extraction, exploitation and optimization of parallelism).

Even provided with enough static information or annotations (OpenMP directives, pointer aliasing, separate compilation assumptions), compilers have a hard time exploring the huge and unstructured search space associated with these application-to-architecture mapping and optimization challenges [16, 32, 23, 53, 1]. In a sense, the task of the compiler can hardly be called optimization anymore, in the traditional meaning of lowering the abstraction penalty of a higher-level language. Together with the run-time system (whether implemented in software or hardware), the compiler is responsible for most of the combinatorial code generation decisions to map the simplified and idealistic operational semantics of the source program to the highly complex and heterogeneous machine.

Unfortunately, optimizing compilers have traditionally been limited to systematic and tedious tasks that are either not accessible to the programmer (e.g., instruction selection, register allocation) or that the programmer in a high level language does not want to deal with (e.g., constant propagation, partial redundancy elimination, dead-code elimination, control-flow optimizations). Generating efficient code for deep parallelism and deep memory hierarchies with complex and dynamic hardware components is a completely different story: the compiler (and run-time system) now has to take the burden of much smarter tasks that only expert programmers would be able to carry.

Recent work showed that polyhedral compilation techniques are good candidates to address these challenges, and that new algorithms allow them to scale to real-size optimization problems (beyond tiny loop kernels) [34, 64]. This paper exposes our road-map towards making GCC the first general-purpose compiler to build on full-scale polyhedral compilation techniques (analysis and transformations, including affine scheduling).

In the first part of the paper we will present the steps to transform the loops and GIMPLE representations to systems of linear constraints, or polyhedra, to transform the matrix form obtained, and to eventually regenerate GIMPLE trees. This part corresponds to an adaptation to GCC of the WRaP-IT tool. Then, we discuss the integration of additional numerical domains, to support a wider range of (interprocedural) static analyses, and to improve the compile time on polyhedral compilation passes. This work will use the APRON library as a starting point, to facilitate the transparent substitution of abstract numerical domains. The APRON library is part of a joint work between different members of the static analysis community in France, and aims at providing a common interface between numerical abstract domains. Because some computations might not be tractable, or too expensive on a too precise representation, it is interesting to use more abstract representations on which computations have lower costs.

We will present experimental results to motivate the polyhedral program transformation approach, and we will survey our methods to let the code analysis and generation techniques scale to full-scale loop nests (with aggressive inlining). We will present the benefits of adopting this infrastructure: composition of transformations, and interchangeability of abstract domains.

2 State of the art

In compilers, polyhedral domains are used for different purposes:

Static analysis. Polyhedra represent conservative approximations of the properties of a program [24]. In this case, the operations on the abstract domain should preserve the safety of the computed properties, and thus the results are allowed to be over approximations.

Code transformations. Polyhedra represent the code itself [31], through the iteration domain, iteration and statement schedules, and memory access functions. The translation and the operations over the polyhedral representation have to be exact (with no loss of information), to guarantee that the code generated from the polyhedral representation after transformation will be semantically equivalent to the original program.

Several works addressed these applications in different experimental frameworks, but the underlying mathematical framework is the same.

In this section we provide an overview of the techniques used in research and industrial compilers based on polyhedral domains: first we present the polyhedral representations for loop iteration domains, then we present the array regions that approximate data accesses. We end this survey with the cost models based on the polyhedral representations.

2.1 Translation to a Polyhedral Representation

The polyhedral representations are restricted by their expressiveness to represent only sequences of loop nests with constant strides and affine bounds. It includes non-rectangular loops, non-perfectly nested loops, and conditionals with boolean expressions of affine inequalities. Loop nests fulfilling these hypotheses are amenable to a representation in the polyhedral model [54]. We call *Static Control Part* (SCoP) any maximal syntactic program segment satisfying these constraints [20]. The reader interested in techniques to extend SCoP coverage (by preliminary transformations) or in partial solutions on how to remove this scoping limitation (procedure abstractions, irregular control structures, etc.) should refer to [62, 37, 21, 72, 25, 12, 60, 11, 19, 22].

In the polyhedral model [57, 31], the iteration steps of a loop nest of depth d are represented as the integer points of a polyhedra in $\mathbb{Z}^d$. In the general case, the polyhedra are bounded by symbolic parameters: they are called parametric polyhedra, and each symbolic parameter is represented using an extra dimension. All variables that are invariant within a SCoP are called *global parameters*. For each statement within a SCoP, the representation separates four attributes, characterized by parameter matrices: the iteration domain, the schedule, the data layout and the access functions.

2.2 WRaP-IT

The WRaP-IT framework [34], developed in the Alchemy group, improves on classical polyhedral representations [31, 68, 41, 46, 2, 47] to support a large array of useful and efficient program transformations (loop fusion, tiling, array forward substitution, statement reordering, software pipelining, array padding, etc.), as well as *compositions* of these transformations. It is implemented within the Open64 and PathScale EKOPath [17] compilers. This compiler family provides key interprocedural analyses and pre-optimization phases such as inlining,

interprocedural constant propagation, loop normalization, integer comparison normalization, dead-code and `goto` elimination, as well as induction variable substitution. Thanks to these preliminary passes, our tool extracts large and representative SCoP for SPEC fp benchmarks: on average, 88% of the statements belong to a SCoP containing at least one loop. See [34] for detailed static and dynamic SCoP coverage. GCC now comes close to Open64 in terms of loop-oriented program normalizations, and the situation improves quickly; our future research will thus benefit from migrating the WRaP-IT framework to GCC.

The main technical idea behind polyhedral program representations is to clearly separate the four different types of actions performed by loop-centric transformations: modification of the iteration domain (loop bounds and strides), modification of the schedule of each individual statement, modification of the access functions (array subscripts), and modification of the data layout (array declarations). This separation makes it possible to provide a matrix representation for each kind of action, enabling the easy and independent composition of the different representation operations associated with each program transformation, and as a result, enabling the composition of transformations themselves. Current representations do not clearly separate these four types of actions; as a result, the implementation of certain compositions of program transformations can be complicated or even impossible. For instance, current implementations of loop fusion must include loop bounds and array subscript modifications even though they are only byproducts of a schedule-oriented program transformation; after applying loop fusion, target loops are often peeled, increasing code size and making further optimizations more complex. Within our representation, loop fusion is only expressed as a schedule transformation, and the modifications of the iteration domain and access functions are implicitly handled, so that the code complexity is exactly the same before and after fusion. Similarly, an iteration domain-oriented transformation like unrolling should have no impact on the schedule or data layout representations; or a data layout-oriented transformation like padding should have no impact on the schedule or iteration domain representations.

3 Loop Transformations in the Polyhedral Model

This section is a quick overview of the polyhedral framework and shows the expressiveness benefits on a practical example. A more formal presentation of the model may be found in [57, 31].

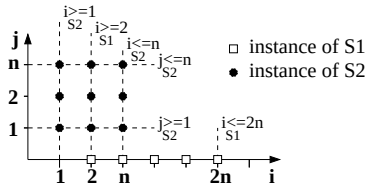
3.1 Quick Overview of the Framework

Polyhedral compilation usually distinguishes three steps: one first has to represent an input program in the formalism, then apply a transformation to this representation, and finally generate the target (syntactic) code.

Consider the polynomial multiplication kernel in Figure 1(a). It only deals with control aspects of the program, and we refer to the two computational statements (array assignments) through their names, S_1 and S_2. To bypass the limitations of syntactic representations, the polyhedral model is closer to the execution itself by considering *statement instances*. For each statement we consider the *iteration domain*, where every statement instance belongs. The domains are described using affine constraints that can be extracted from the program control. For example, the iteration domain of statement S_1, called D^{S_1}, is the set of values (i)

```
    for (i=2; i<=2*n; i++)
S1  | Z[i] = 0;
    for (i=1; i<=n; i++)
    |  for (j=1; j<=n; j++)
S2  |  | Z[i+j] += X[i] * Y[j];
```

(a) Syntactic form

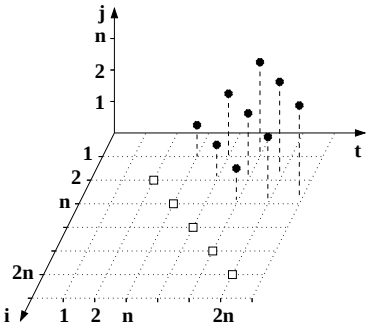


(b) Polyhedral domains ($n \geq 2$)

Figure 1: A polynomial multiplication kernel and its polyhedral domains

$$\left(\begin{array}{c|c} \mathrm{I} & \Theta \\ \hline 0 & \Lambda \end{array}\right) \cdot \left(\begin{array}{c} -\mathbf{t} \\ \hline \mathbf{i} \\ \mathbf{g} \\ 1 \end{array}\right) \begin{array}{c} = \mathbf{0} \\ \hline \geq \mathbf{0} \end{array}$$

(a) Transformation template formula



(b) Transformed polyhedra

Figure 2: Transformation template and its application

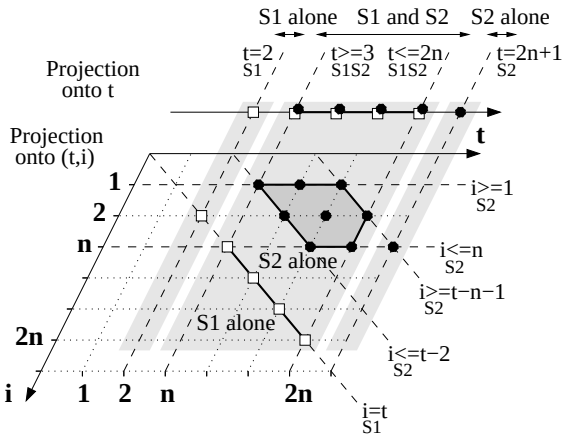
such that $2 \leq i \leq n$ as shown in Figure 1(b); a matrix representation is used to represent such constraints: in our example, D^{S_1} is characterized by

$$\begin{bmatrix} 1 & 0 & -2 \\ -1 & 2 & 0 \end{bmatrix} \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} \geq \mathbf{0}.$$

In this framework, a transformation of the execution order is characterized by *affine scheduling functions* Θ^S, for all statements S in the SCoP. Each statement has its own scheduling function which maps each run-time statement instance to a logical execution date. In our polynomial multiplication example, an optimizer may notice a locality problem and discover a good data reuse potential over array `Z`, then suggest $\Theta^{S_1}(i) = (i)$ and $\Theta^{S_2}\binom{i}{j} = (i + j + 1)$ to achieve better locality (see e.g., [14] for a method to compute such functions). The intuition behind such transformation is to execute consecutively the instances of S_2 having the same $i+j$ value (thus accessing the same array element of `Z`) and to ensure that the initialization of each element is executed by S_1 just before the first instance of S_2 referring this element. In the polyhedral model, a transformation is applied following the template formula in Figure 2(a) [13], where $\mathbf{i}$ is the iteration vector, $\mathbf{g}$ is the vector of constant parameters, and $\mathbf{t}$ is the *time-vector*, i.e. the vector of the scheduling dimensions. The nature of these vectors and the structure of the Θ and Λ matrices is detailed in [34]. Notice that in this formula, equality constraints capture schedule modifications, and inequality constraints capture iteration domain modifications. The resulting polyhedra for our example are shown in Figure 2(b), with the additional dimension t.

Once transformations have been applied in the polyhedral model, one needs to (re)generate the target code. The best syntax tree construction

scheme consists in a recursive application of domain projections and separations [59, 13]. The final code is deduced from the set of constraints describing the polyhedra attached to each node in the tree. In our example, the first step is a projection onto the first dimension t, followed by a separation into disjoint polyhedra, as shown on the top of Figure 3(a). This builds the outer loops of the target code (the loops with iterator t in Figure 3(b)). The same process is applied onto the first two dimensions (bottom of Figure 3(a)) to build the second loop level and so on. The final code is shown in Figure 3(b) (the reader may care to verify that this solution maximally exploits temporal reuse of array `Z`). Note that the separation step for two polyhedra needs three operations: $D^{S_1} - D^{S_2}$, $D^{S_2} - D^{S_1}$ and $D^{S_2} \cap D^{S_1}$, thus for n statements the worst-case complexity is 3^n.



(a) Projections and separations

```
    t=2; // Such equality is a loop running once
    | i=2;
S1  | | Z[i] = 0;
    for (t=3; t<=2*n; t++)
    | for (i=max(1,t-n-1); i<=min(t-2,n); i++)
    | | j = t-i-1;
S2  | | Z[i+j] += X[i] * Y[j]
    | i=t;
S1  | | Z[i] = 0;
    t=2*n+1;
    | i=n;
    | | j=n;
S2  | | | Z[i+j] += X[i] * Y[j];
```

(b) Target code

Figure 3: Target code generation

It is interesting to note that the target code, although obtained after only one transformation step, is quite different from the original loop nest. Indeed, multiple classical loop transformations are necessary to simulate this one-step optimization (among them, software pipelining and skewing). The intuition is that arbitrarily complex compositions of classical transformations can be captured in one single transformation step of the polyhedral model. This was best illustrated by affine scheduling [31, 41] and partitioning [46] algorithms. Yet, because black-box, model-based optimizers fail on modern processors, we propose to step back a little bit *and consider again the benefits of composing classical loop transformations, but using a polyhedral representation*. Indeed, before our recent work, polyhedral optimization frameworks have only considered the isolated application of one arbitrarily complex affine transformation. The main originality of our work is to address the *composition of program transformations on the polyhedral representation itself*, which vastly facilitates the coordination of polyhedral transformations with classical heuristics and cost models, enabling their integration into production compilers.

3.2 Code Generation from the Polyhedral Model

Regenerating syntax trees from affine schedules is one of the most time-consuming parts of the polyhedral compilation flow. The history of code generation in the polyhedral model shows a constant growth in transformation complexity, from basic schedules for a single statement to general affine transformations for wide code regions. In their seminal work, Ancourt and Irigoin limited transformations to unimodular functions (determinant 1 or -1) and the code generation process was applicable for only one

domain at once [5]. Several works succeeded in relaxing the unimodularity constraint to invertibility (the T matrix has to be invertible), enlarging the set of possible transformations [27, 45]. A further step has been achieved by Kelly et al. by considering more than one domain and multiple scheduling functions at the same time [42]. All these methods relied on the Fourier-Motzkin elimination method [61] to build the target code.

Quilleré et al. showed how to use polyhedral operations based on the Chernikova Algorithm [66] instead, to benefit from its practical efficiency to handle bigger problems [59]. Recently, a new transformation policy has been proposed to allow general non-invertible, non-uniform, non-integral affine transformations [13, 64]. Such freedom allowed to apply polyhedral techniques to much larger programs with very sophisticated transformations, and led to novel complexity, scalability and code quality challenges we discuss in this paper. In the context of GCC, it would be very interesting to try to preserve the robustness of the Quilleré algorithm, but further improve the complexity of the method, using depth-sensitive relaxations (approximations) and Fourier-Motzkin eliminations.

3.3 Optimization Experiment

We applied the WRaP-IT tool to the swim SPEC CPU2000 fp benchmark, extracting several SCoP: aggressive inlining yields one SCoP of 421 lines of code—112 instructions in the polyhedral representation—in consecutive loop nests within the `main` function. We applied more than 30 transformations to this SCoP, including multi-level loop fusion, loop shifting (pipelining), loop tiling, loop peeling, loop unrolling, loop interchange, and strip-mining [71, 4]. All these transformations are generalized to non-perfectly nested codes, and embedded in our compositional framework [34].

The resulting code is significantly larger—2267 lines—roughly one third of them being naive scalar copies to map schedule iterators to domain ones, fully eliminated by copy-propagation in the subsequent run of EKOPath or Open64. This is not surprising since most transformations in the script require domain decomposition, either explicitly (peeling) or implicitly (shifting prolog/epilog, at code generation). It takes 39s to apply the whole transformation sequence up to native code generation on a 2.08GHz AthlonXP. Transformation time is dominated by back-end compilation (22s). Polyhedral code generation takes only 4s. Exact polyhedral dependence analysis (computation and checking) is acceptable (12s). Applying the transformation sequence itself is negligible. These execution times are very encouraging, given the complex overlap of peeled polyhedra in the code generation phase, and since the full dependence graph captures the exact dependence information for the 215 array references in the SCoP at every loop depth (maximum 5 after tiling), yielding a total of 441 dependence matrices.

Compared to the *peak performance attainable by the best available compiler*, PathScale EKOPath (V2.1) with the *peak-SPEC* optimization flags, our tool achieves **32% speedup on Athlon XP and 38% speedup on Athlon 64**. Compared to the *base-SPEC* performance numbers, our optimization achieves **51% speedup on Athlon XP and 92% speedup on Athlon 64**. We are not aware of any other optimization effort—manual or automatic—that brought swim to this level of performance on *x*86 processors.[2]

[2]Notice we consider the SPEC CPU2000 version of swim, much harder to optimize through loop fusion than the SPEC 95 version.

4 Static Analysis with Polyhedra

Let us now discuss some of the static analysis opportunities and challenges offered by polyhedral methods.

4.1 Instancewise Polyhedral Dependence Analysis

Many tests have been designed for dependence checking between different statements or between different executions of the same statement. It has been shown that this problem is equivalent to detecting whether a system of equations has an integer solution inside a region of $\mathbb{Z}^n$ [9].

Most of the dependence tests try to find efficiently a reliable, approximative but conservative (they overestimate data dependences) solutions. The GCD-test [8] has been the very first practical solution, it is still present in many implementations as a first check with low computational cost. This test assumes that if the greatest common divisor of the coefficients of an equation divides the constant term, then a solution exists. A generalized GCD-test has been proposed to handle multi-dimensional array references [9]. The Banerjee test uses the intermediate value theorem to disprove a dependence: it computes the upper and lower bounds of an equation and checks if the constant part lies in that range [69]. The λ-test is an extension to this test that handles multi-dimensional array references [43]. Some other important solutions are a combination of GCD and Banerjee tests called I-test [43], the Δ-test [35] that gives an exact solution when there is at most one variable in the subscript functions, and the Power-test which uses the Fourier-Motzkin variable elimination method [61] to prove or disprove dependences [70]. Beside their approximative nature, these dependence tests suffer from many other major limitations. The most stringent one is their inability to precisely handle `if` conditionals, loops with parametric bounds, triangular loops (a loop bound depends on an outer loop counter), coupled subscripts (two different array subscripts refer the same loop counter), or parametric subscripts.

On the opposite, a few methods allow to find an exact solution to the dependence problem, but at a higher computational cost. The OMEGA-test is an extension to the Fourier-Motzkin variable elimination method to find integral solutions [55]. On one hand, once a variable is eliminated, the original system has an integer solution only if the new system has an integer solution (if this is not the case there is no solution). On the other hand, if an integer point exists in a space computed from the new system, then there exists an integer point in the original system (if this is the case, there is a solution). The PIP-test uses a parametric version of the dual-simplex method with Gomory cuts to find an integral solution [30]. These two tests not only give an exact answer, they are also able to deal with complex loop structures and (affine) array subscripts. The PIP-test is more precise than the OMEGA-test when dealing with parametric codes (when one or more integer symbolic constant are present), for instance, in the following pseudo-code:

```
for(i=0; i<=N; i++) {
  A[i] = ...;
  ... = ... A[i+100] ...;
}
```

the OMEGA-test will state that there is a dependence between the two statements while the PIP-test will precise that the dependence only exists if N is greater or equal to 100. Both tests have worst-case exponential complexities but work quite well in practice as shown by Pugh for the OMEGA-test [55]. Other costly exact

tests exist in the literature [49, 28] but are often not able to handle complex control in spite of their cost.

```
 for(i=0; i<=N; i++)
   for(i=0; i<=N; i++)
S    A[i][j] = A[j][i] + A[i][j-1];
```

We do not advocate for the use of any of these tests, but rather for the computation of *instancewise* dependence information as precisely as possible, i.e., for intensionally describing the statically unbounded set of all pairs of dependent statement iterations, called *instances*. Dependence tests are statementwise decision problems associated with the existence of a pair of dependent instances, while instancewise dependence analysis provides additional information that can enable finer program transformations, like affine scheduling [44, 31, 46, 36]. The intensional characterization of instancewise dependences can take the form of multiple *dependence abstractions*, depending on the precision of the analysis and on the requirements of the user. The simplest and least precise one is called *dependence levels*, it specifies for a given loop nest which loop carry the dependence. It has been introduced in the Allen and Kennedy parallelization algorithm [3]. The *direction vectors* is a more precise abstraction where the i-th element approximates the value of all the i-th elements of the *distance vectors* (which shows the difference of the loop counters of two dependent instances). It has been introduced by Lamport [44] and formalized by Wolfe [71] and is clearly the most widely used representation. A more precise abstraction is the *dependence polyhedron* [40] which is able to determine exactly the set of statement instances in dependence relation. The choice of a given dependence abstraction is crucial for further study: choosing an imprecise one can result in blacking out interesting transformations. For instance, let us consider the following example:

there are three dependences in this loop nest (a read-after-write dependence from $\langle S,i,j\rangle$ to $\langle S,i,j+1\rangle$, another read-after-write dependence from $\langle S,i,j\rangle$ to $\langle S,j,i\rangle$ and a write-after-read from $\langle S,j,i\rangle$ to $\langle S,i,j\rangle$). Dependence levels are 2, 1 and 1: each loop carries at least one dependence and no parallelism can be found. Direction vectors are $(0,1)$, $(+,-)$, $(+,-)$: the second coefficients 1 and $-$ hamper any parallelism detection. Using dependence polyhedra, parallelism may be found: the Feautrier algorithm suggests the affine schedule $\theta(i,j) = 2i+j-3$ (all instances with the same schedule may be run in parallel), see [67]. We propose to compute one of the most precise representation of dependences: the dependence polyhedra.

We exercise this implementation on 6 full SPEC CPU2000 fp benchmarks. In the most challenging examples, the biggest SCoP almost contains the whole program after inlining. On a 2.4GHz Pentium 4, the full instancewise dependence analysis takes up to 37.512 seconds, for the largest SCoP in applu. This is an extreme case with huge iteration spaces (more than 13 dimensions on average, and up to 19 dimensions). This may sound quite costly, but it still shows that the analysis is compatible with the typical execution time of aggressive optimizers (typically more than ten seconds for Open64 with interprocedural optimization and aggressive inlining and loop-nest optimizations). In all other cases, it takes **less than 5 seconds**, despite thousands of operations on polyhedra with close to 10 dimensions on average. These are very compelling results since we compute very large dependence graphs, taking *all pairs of references* into account, without k-limiting heuristic on their syntactic or nesting distance

as it is the case in classical optimization frameworks. Also, a typical loop optimizer will perform on-demand computations on part of the dependence graph only.

4.2 Array Regions

In the context of interprocedural analysis of data dependences, array regions techniques have been proposed to extend the data dependence analysis. This representation is able to accurately describe sets of reads or writes that occur during the execution of a procedure. Approximations of accessed regions is not useful in general, as the precision degradation harm to the extraction of precise use-def chains, or dependence tests. As a practical implementation, the PIPS compiler [39, 38] uses the notion of transformers, or transfer functions for defining polyhedral relations between memory stores. Transformers are computed interprocedurally bottom-up accumulating the effects of procedures on memory accesses [26, 25], while preconditions are computed top-down accumulating a safe description of the state of the program when entering a procedure. Once this information has been gathered, the dependence test can be refined using the interprocedural information. A more advanced dependence test can be implemented by gathering more precise information: the in and out regions. The regions that contain both reads and writes potentially prevent the parallelization of a loop. When the data is written once and then read, the region can be selected to be privatized [63]. Because the duplication of runtime data can be harmful to the execution speed, the decision to privatize the data is deferred to an analyzer that can determine the benefit of the transformation.

4.3 Cost Models

Most of the loop nest transformations require a profitability analysis: often, several transformation schemes are available and following the specificities of the target architecture some strategies are preferable. The metrics developed for the polyhedral model are generally based on counting the number of integer points in polyhedra [58, 18, 65]. As polyhedra represent iteration spaces or data accesses, the evaluation of the number of integer points corresponds to the evaluation of the number of iterations of a loop nest, or the size or frequency of the accessed data. From this measure it is possible to infer other useful informations such as the memory bandwidth, cache reuse for the execution of a loop [48], cache misses [33], etc. Most of these methods are exponential, but some recent works [65] implemented and experimented with promising polynomial time algorithms for counting integer points in polyhedra.

5 Road Map

We describe the components that compose GRAPHITE, the priorities and dependences between the modules, and discuss the proposed plan for the integration into GCC. An overview of the modules is depicted in Figure 4: the development of the modules contains five stages. First the translation from GIMPLE to the polyhedral representation, then the translation back to GIMPLE, the development of cost models and the selection of the transform schedule. The interprocedural refinement of the data dependence information based on the array regions is optional, but it is necessary for gathering more precise informations that potentially could enable more transformations, or more precise transform decisions. Finally, the least critical component is the integration of the numerical domains common interface, based on which it will be possible to change the complexity of the algorithms used in the polyhedral analyses.

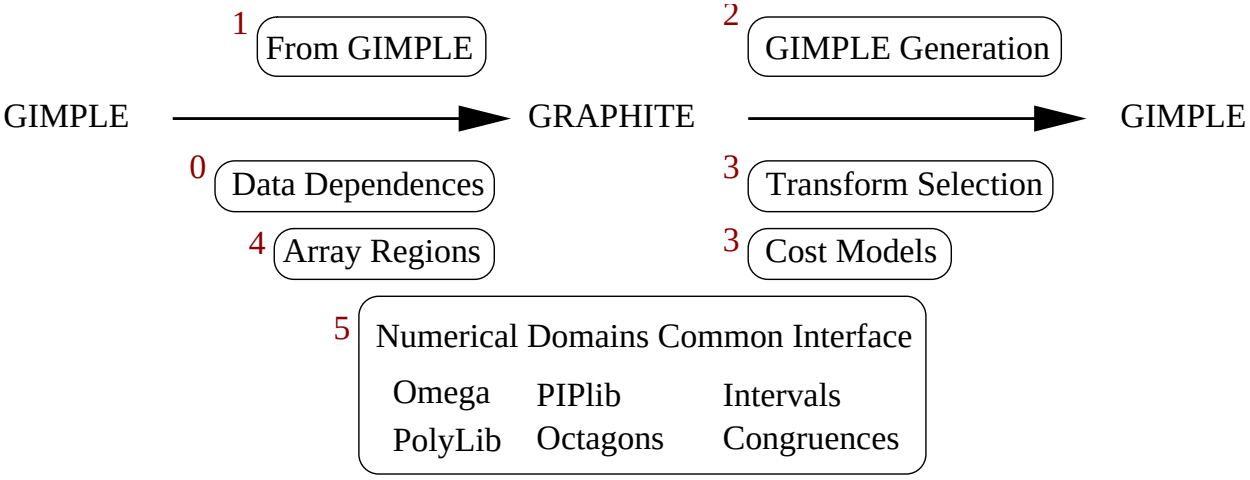


Figure 4: Overview of the modules composing GRAPHITE. The numbers indicate the order in which each module will be integrated.

5.1 Polyhedral Data Dependences

The code for statementwise dependence testing has been integrated in GCC and directly uses the OMEGA data structures for representing polyhedra.

Instancewise dependence analysis will be imported from the WRaP-IT framework, and extended to non-affine constructs and `while` loops [72, 10].

The main challenge here is memory usage, and the associated adaptation of dependence abstraction accuracy depending on some form of distance in the SCoP.

5.2 Translation to and from GRAPHITE

The next step for the integration of the GRAPHITE project is the translation of the code to the polyhedral representation and back to GIMPLE. This translation will be developed as an extension of the existing LAMBDA kernel [45, 15]. The main limitation of LAMBDA is that it works only on a single loop nest and on distance vectors, whereas more exact dependence informations like the polyhedral representation of the dependences among several sibling loop nests will be needed for GRAPHITE.

To support the code generation engine of GRAPHITE, we will only need a reduced set of operations on polyhedra [13, 64]—projection, difference, and intersection—but these operations are very expensive when naively implemented on top of OMEGA. For this reason we will use a more efficient algorithm that is part of the PIP library [29].

After this basic infrastructure has been implemented, it is possible to transform the code by selecting the transformations either by hand, or let an expert system select the transformation sequence based on some metric.

5.3 Optimization Heuristics

Cost models and optimization selection could be implemented using different techniques. Purely static methods are based on the evaluation of statically computable properties, and

are frequent in the classic heuristics for loop transformations. New heuristic techniques include cost models based on performance measurements either on abstract interpreters, simulators, or real hardware. The integration of this information in the compiler can be based on machine-learning techniques [1] operating directly on the polyhedral representation. Hybrid techniques include a part of static analysis in the classification or compression of the data gathered by the dynamic measurement for enabling the generalization of the decisions for patterns contained in programs that were not part of the training benchmark suite.

The analyzers for the profitability of a transformation sequence are critical to GRAPHITE and have to be implemented just after the translators in and out of the polyhedral form.

5.4 Integration of Array Regions

The computation of transformers and preconditions as in PIPS, will be based on the generic propagation engine [52], that has to be extended to the interprocedural mode. In intraprocedural mode, the array regions informations is not very useful, because the data dependence analysis is able to produce the same accurate results.

The extraction of more precise memory accesses in interprocedural mode can be deferred to a later stage of improvements. More improvements in the precision of the data dependence analysis can be done in parallel with the implementation of GRAPHITE as they will also benefit to other optimization passes.

5.5 Numerical Domains Common Interface

Up to now, we considered parameterized polyhedra (with integral points) as the foundation for program abstractions, representations and optimizations. In the search for faster compilation techniques or more flexible representations (beyond static control nests), we are interested in several related *numerical domains*.

A common interface for numerical domains computations has been designed in the APRON project [6] that gathers several members of the static analysis community in France: École des mines de Paris, École normale supérieure, École polytechnique, Vérimag, and IRISA. The goal of this interface is to ease the use of different numerical domains libraries with minimal code changes. Several existing libraries that implement intervals, octagons, polyhedra, linear and polynomial equalities, polynomial inequalities, etc., have been considered during the design, and a reduced number of common operations have been retained for the common interface: these operations are those that occur among all the domains, i.e. construction, meet, join, projection, etc., for which every numerical domain library is providing an implementation.

One of the main goals to the integration of this common interface in GCC is to ease the integration of new developments from the static analysis community. This interface is just a contract between the user and the implementors of the numerical domains, as the interface does not include the code of the underlying libraries: it is just a guarantee that a new numerical domain library will provide the basic operations. For this reason we will have to either consider the inclusion of the numerical domains libraries in the core of GCC, or add a new dependence on some library developed aside. In both cases there is an overhead to the inclusion. We consider the integration of the APRON common interface only as a long term project, as we can use the OMEGA library for precise operations, and implement on the side the missing specialized algorithms provided by other libraries.

In the following, we separately describe some

of the libraries that contain the main functionalities needed for the GRAPHITE project.

5.5.1 OMEGA Library

The OMEGA Library [56] has been developed for solving and reducing Presburger arithmetic formulas. OMEGA is known to be expressive, but it is doubly exponential in the worst case (and often exponential in practice). This library has been already integrated to GCC, and will also be part of the APRON common interface, but it cannot (alone) face the complexity of polyhedral code generation and dependence analysis. Some specialized algorithms that are faster in practice are used: as for example the algorithm from the PIP library.

5.5.2 PIP Library

The PIP library [29] contains a specialized algorithm to compute affine objective function or lexicographical minima (or maxima) in convex polyhedra. The main algorithms of this library can be contributed to GCC as a refinement for the operations that use OMEGA.

5.5.3 Octagons

A library that provides a domain for octagons has been implemented by Antoine Miné [50, 51]. Its use in GRAPHITE would be just experimental, yet has the potential to be a local multi-criteria optimum in terms of accuracy, expressiveness (as a program representation vehicle) and compilation speed. We wish to conduct active research in this area, yet it is not on the critical path.

5.5.4 Specific Algorithms for Polyhedra

There are some libraries that implement specialized algorithms that we will consider for the reduced computation cost: we will consider the integration of the Barvinok library [65] to count in polynomial time the number of points in integer polyhedra, but also some missing parts of the PolyLib.

5.6 Maintenance of Components

The objective is to minimize the effort needed to implement and to maintain the code: smallest number of lines of code, fast algorithms specialized to compilation, rewrite some existing code, clean up, etc, as for the integration of OMEGA. It will also be interesting to benefit from the existence of active communities that develop some of the abstract numerical domains, and create dependences on outer libraries when their license is compatible.

More concretely, all existing code for the WRaP-IT project is licensed under the (L)GPL. The code is mostly implemented in C, except for parts of WRaP-IT (C++ and domain-specific transformation language). As all this code is specific to the internal representations of the compiler, it will be integrated to GCC. This will be the most costly part, in number of lines of code, to be integrated in GCC.

The analysis of the profitability will be based on libraries developed aside, and will contain fewer lines of code. The APRON library will be licensed under LGPL, and the libraries that will work within this framework are either in the public domain, as the OMEGA library, or under GPL, as the Parma PolyLib [7], the PolyLib, and Polka. We propose to keep all these libraries out of the core of GCC, for taking profit of their active communities. When one of the libraries is not maintained anymore, as in the

case of OMEGA, there are several ways to handle the changes: the first solution is to host the changes on savannah as a new project for the library, the other solution is to integrate the code to an active library, or as a last resort, to integrate the code to GCC. For instance the PIP library is a short, a few thousand lines of code, specialized algorithm that could be integrated in the PolyLib.

5.7 Other Loop Optimization Modules

The heavy infrastructure of the loop-nest optimizer can be implemented independently of GRAPHITE, such as privatization, array renaming, flattening of arrays, automatic parallelization, etc. However some of these transformations could use the cost models and validity analyses developed for GRAPHITE.

6 Conclusion

GRAPHITE is the first production compiler project to implement a polyhedral loop-nest optimizer. It has a strong potential for existing architectures, and even more, as a framework to improve the productivity of compiler construction for future and emerging processors.

From a research perspective, GRAPHITE will raise several unsolved problems.

- Polyhedral frameworks only offer a limited support for interprocedural or irregular control flow. Solving this problem would lead to a dramatic extension of the potential of the polyhedral model.
- Is there a good tradeoff in expressiveness and algorithmic complexity? Are the octagons precise enough to represent enough common cases?
- What is the future of cost models and heuristics, based on polyhedral description of target machines, static and feedback-directed performance estimation?

References

[1] F. Agakov, E. Bonilla, J.Cavazos, B.Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *4th Annual International Symposium on Code Generation and Optimization (CGO)*, Mar. 2006.

[2] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *ACM Supercomputing'00*, May 2000.

[3] R. J. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, october 1987.

[4] R. J. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[5] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loop. In *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'91)*, pages 39–50, June 1991.

[6] Apron: Numerical program analysis. `http://www.cri.ensmp.fr/apron/`.

[7] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the parma polyhedra library. In M. V. Hermenegildo and

G. Puebla, editors, *Static Analysis: Proceedings of the 9th International Symposium, LNCS 2477*, pages 213–229, 2002.

[8] U. Banerjee. Data dependence in ordinary programs. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, November 1976.

[9] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, 1988.

[10] D. Barthou. *Array Dataflow Analysis in Presence of Non-affine Constraints*. PhD thesis, Université de Versailles, France, Feb. 1998. `http://www.prism.uvsq.fr/~bad/these.html`.

[11] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. *Int. J. of Parallel Programming*, 28(3):213–243, June 2000.

[12] D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *J. of Parallel and Distributed Computing*, 40:210–226, 1997.

[13] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Parallel Architectures and Compilation Techniques (PACT'04)*, Antibes, France, Sept. 2004.

[14] C. Bastoul and P. Feautrier. Improving data locality by chunking. In *CC'12 International Conference on Compiler Construction, LNCS 2622*, pages 320–335, Warsaw, april 2003.

[15] D. Berlin, D. Edelsohn, and S. Pop. High-level loop optimizations for GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 37–54, 2004. `http://www.gccsummit.org/2004`.

[16] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proc. Workshop on Profile and Feedback Directed Compilation*, 1998.

[17] F. Chow. Maximizing application performance through interprocedural optimization with the pathscale eko compiler suite. `http://www.pathscale.com/whitepapers.html`, Aug. 2004.

[18] P. Clauss and V. Loechner. Parametric Analysis of Polyhedral Iteration Spaces. In *IEEE Int. Conf. on Application Specific Array Processors, ASAP'96*. IEEE Computer Society, August 1996.

[19] A. Cohen. *Program Analysis and Transformation: from the Polytope Model to Formal Languages*. PhD thesis, Université de Versailles, France, Dec. 1999.

[20] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. In *Euro-Par'04*, number 3149 in LNCS, pages 292–303, Pisa, Italy, Aug. 2004. Springer-Verlag.

[21] J.-F. Collard. Automatic parallelization of while-loops using speculative execution. *Int. J. of Parallel Programming*, 23(2):191–219, Apr. 1995.

[22] J.-F. Collard. *Reasoning About Program Transformations*. Springer-Verlag, 2002.

[23] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. of Supercomputing*, 2002.

[24] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among

variables of a program. In *5thACM Symp. on Principles of Programming Languages*, pages 84–96, Jan. 1978.

[25] B. Creusillet. *Array Region Analyses and Applications*. PhD thesis, École Nationale Supérieure des Mines de Paris (ENSMP), Paris, France, Dec. 1996.

[26] B. Creusillet and F. Irigoin. Interprocedural array region analyses. *Int. J. of Parallel Programming*, 24(6):513–546, Dec. 1996.

[27] A. Darte and Y. Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20(5):679–710, 1994.

[28] C. Eisenbeis and J.-C. Sogno. A general algorithm for data dependence analysis. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 292–302, Washington, D. C., United States, 1992. ACM Press.

[29] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, Sept. 1988.

[30] P. Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, Feb. 1991.

[31] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, Dec. 1992. See also Part I, one dimensional time, 21(5):315–348.

[32] G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *11thWorkshop on Languages and Compilers for Parallel Computing*, LNCS, Washington DC, July 2002. Springer-Verlag.

[33] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[34] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. of Parallel Programming*, 2006. 57 pages. Accepted for publication.

[35] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 15–29, New York, june 1991.

[36] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis. Facultät für Mathematik und Informatik, Universität Passau, 2004.

[37] M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of `while` loops. In S. Haridi, K. Ali, and P. Magnusson, editors, *EuroPar'95*, volume 966 of *LNCS*, pages 315–326. Springer-Verlag, 1995.

[38] F. Irigoin, P. Jouvelot, and R. Triolet. Overview of the PIPS project. In P. Feautrier and F. Irigoin, editors, *2ndIntl. Workshop on Compilers for Parallel Computers*, pages 199–212, Paris, Dec. 1990.

[39] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *ACM Int. Conf. on*

Supercomputing (ICS'91), Cologne, Germany, June 1991.

[40] F. Irigoin and R. Triolet. Computing dependence direction vectors and dependence cones with linear systems. Technical Report ENSMP-CAI-87-E94, Ecole des Mines de Paris, Fontainebleau (France), 1987.

[41] W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, University of Maryland, 1996.

[42] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers'95 Symposium on the frontiers of massively parallel computation*, McLean, 1995.

[43] X. Kong, D. Klappholz, and K. Psarris. The i test: A new test for subscript data dependence. In *ICPP'90 International Conference on Parallel Processing*, pages 204–211, St. Charles, august 1990.

[44] L. Lamport. The parallel execution of do loops. *Communications of ACM*, 17(2):83–93, 1974.

[45] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, Apr. 1994.

[46] A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *24thACM Symp. on Principles of Programming Languages*, pages 201–214, Paris, France, jan 1997.

[47] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 102–112, 2001.

[48] V. Loechner, B. Meister, and P. Clauss. Precise data locality optimization of nested loops. *J. of Supercomputing*, 21(1):37–76, Jan. 2002.

[49] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 1–14, New York, NY, USA, 1991.

[50] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[51] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 2006. (to appear) `http://www.di.ens.fr/~mine/publi/article-mine-HOSC06.pdf`.

[52] D. Novillo. A propagation engine for gcc. In *Proceedings of the 2005 GCC Developers Summit*, pages 175–184, 2005. `http://www.gccsummit.org/2005`.

[53] D. Parello, O. Temam, A. Cohen, and J.-M. Verdun. Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *ACM Supercomputing'04*, Pittsburgh, Pennsylvania, Nov. 2004. 15 pages.

[54] G.-R. Perrin and A. Darte, editors. *The Data Parallel Programming Model*. Number 1132 in LNCS. Springer-Verlag, 1996.

[55] W. Pugh. The omega test: a fast and practical integer programming algorithm

for dependence analysis. In *Proceedings of the third ACM/IEEE conference on Supercomputing*, pages 4–13, Albuquerque, Aug. 1991.

[56] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[57] W. Pugh. Uniform techniques for loop optimization. In *ACM Int. Conf. on Supercomputing (ICS'91)*, pages 341–352, Cologne, Germany, June 1991.

[58] W. Pugh. Counting solutions to presburger formulas: How and why. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–134, 1994.

[59] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl. J. of Parallel Programming*, 28(5):469–498, Oct. 2000.

[60] L. Rauchwerger and D. Padua. The LRPD test: Speculative run–time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems, Special Issue on Compilers and Languages for Parallel and Distributed Computers*, 10(2):160–180, 1999.

[61] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, Chichester, UK, 1986.

[62] R. Triolet, P. Feautrier, and P. Jouvelot. Automatic parallelization of fortran programs in the presence of procedure calls. In *Proc. of the 1st European Symp. on Programming (ESOP'86)*, number 213 in LNCS, pages 210–222. Springer-Verlag, Mar. 1986.

[63] P. Tu and D. Padua. Automatic array privatization. In *6th Workshop on Languages and Compilers for Parallel Computing*, number 768 in LNCS, pages 500–521, Portland, Oregon, Aug. 1993.

[64] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, LNCS, pages 185–201, Vienna, Austria, Mar. 2006. Springer-Verlag.

[65] S. Verdoolaege, K. Woods, M. Bruynooghe, and R. Cools. Computation and manipulation of enumerators of integer projections of parametric polytopes. Technical Report CW 392, Katholieke Universiteit Leuven, Dept. of Computer Science, 2005. `http://www.kotnet.org/~skimo/barvinok/`.

[66] H. L. Verge. A note on Chernikova's algorithm. Technical Report 635, IRISA, 1992.

[67] F. Vivien. *Détection de parallélisme dans les boucles imbriquées*. PhD thesis, ENS - Lyon, 1997.

[68] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, Aug. 1992. Published as CSL-TR-92-538.

[69] M. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, 1987.

[70] M. Wolfe and C. W. Tseng. The power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, 1992.

[71] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[72] D. G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, University of Maryland, 1995.

Treegion Instruction Scheduling in GCC

Michael C. Rosier and Thomas M. Conte
Center for Embedded System Research
Department of Electrical and Computer Engineering
North Carolina State University
{mcrosier,conte}@ncsu.edu

Abstract

Instruction scheduling is a critical compilation phase for extracting significant amounts of parallelism within a program. The first step of instruction scheduling is region formation; the size and characteristics of the region play an important role in determine the amount of available ILP. In this work the status of the implementation of an architecture-independent, aggressive global instruction scheduler based on Treegions is presented. A Treegion is a a tree-shaped subgraph of the control-flow-graph (CFG). Unlike other region formation algorithms, such as Traces or Superblocks, Treegions take into account multiple execution paths, producing more opportunities for parallelism. Unlike Hyperblocks, Treegions do not require predicates. Treegion formation can use tail duplication. To limit the possible negative side effects of code expansion, our Treegion scheduler uses the Instantaneous Code Size Efficiency (ICSE) heuristic of Zhou for conservative tail duplication. Experimental results show that Treegion formation dramatically increases the average region size as compared to the current region formation code. The implementation currently resides on the sched-treegion-branch.

1 Introduction

High performance microprocessors use complex hardware techniques (*e.g.*, out-of-order execution, branch prediction, prefetching) to exploit parallelism within a program. Alternatively, instruction scheduling is a compile-time technique for extracting instruction-level parallelism (ILP). For wide-issue statically scheduled processors (*e.g.,* EPIC, VLIW), instruction scheduling plays an exceedingly important role in improving performance.

During global instruction scheduling the compiler divides a program's control flow graph (CFG) into multiple regions and then schedules each region separately. The scope of a region may be limited to a basic block (*i.e.*, basic block scheduling) or encompass the entire CFG. Past work has predominantly focused on *linear* regions, *i.e.*, regions containing a single control path, which often limits speculation, resulting in an under utilization of processor resources. These types of global instruction scheduling techniques include Trace scheduling [1], Superblocks [2], and Hyperblocks [3]. Trace scheduling forms linear regions, called traces, of basic blocks that execute sequentially. Loop-unrolling is commonly used as a trace enlarging optimization. Similar to Trace scheduling, Superblocks form single-entry, (possibly)

multiple-exit linear regions. After Superblock formation side entrances are removed via tail duplication. Finally, Hyperblocks extend upon Superblocks by using hardware predication to reduce the need for tail duplication. These techniques suffer from a number of pitfalls. First, formation is based on profile information. Often when scheduling for the more probable path the less likely path suffers. Variation in input sets or lack of profile information can result in a significant performance penalty. Furthermore, the linearity of these regions limit the opportunity for speculation.

A Treegion is a non-linear, single-entry, multiple-exit region of code containing basic blocks that constitute a tree-shaped subgraph of the control-flow-graph (CFG). Building large regions is a critical aspect of instruction scheduling that enables the compiler to extract parallelism. Unlike other region formation algorithms, such as Traces and Superblocks, Treegions include multiple paths of execution, producing larger regions and more opportunities for speculation. In addition, Treegions do not require special architectural features for region formation.

GCC currently supports both linear and non-linear regions. Linear regions are supported in the form of Superblocks (`tracer.c`) and Extended Basic Blocks (EBB) (`sched-ebb.c`). Meanwhile, support for non-linear regions (`sched-rgn.c`) are limited to loop-free procedures and reducible inner loops. Treegions have the advantage that unlike Superblocks and EBB, their formation includes multiple paths of execution and do not require profile information. Generally, Treegions can realize significantly larger regions than other region formation techniques.

The remainder of this paper is organized as follows. Section 2 describes the current GCC global instruction scheduler. Section 3 describes natural treegion formation, or treegion formation without tail duplication. Section 4 describes an efficient technique for the tail duplication of treegions. Sections 5 and 6 discuss Treegion scheduling and experimental results, respectively. Finally, section 7 gives a brief conclusion.

2 GCC Instruction Scheduling

The GCC instruction scheduler is a list-based instruction scheduler derived from work originally developed at IBM Haifa Labs. The generic parts of the scheduler are found in `haifa-sched.c`. The goal of list-scheduling is to minimizing the length of the critical path while maximizing the opportunity for parallelism. The steps to list scheduling are as follows:

1. Build the data dependence graph.
2. Calculate priorities for each instruction.
3. Iteratively schedule ready instructions.

The scheduler is invoked before and after register allocation. Treegion scheduling extends upon the interblock scheduling pass, found in `sched-rgn.c`, performed prior to register allocation. Instructions may be speculatively scheduled during the first pass with much greater ease than during the second pass. After register allocation each pseudo-register has been assigned a physical register, introducing anti- and output-dependencies. These dependencies greatly restrict scheduling.

Region formation is the first step of interblock scheduling. In this work, treegions are the chosen region type. Treegion formation is a two step process involving natural treegion formation and tail duplication, which are discussed in sections 3 and 4, respectively.

Prior to scheduling, dependencies between instructions are found for each basic block within the region. Such dependencies include those between registers (*i.e.,* true-, anti-, and output-dependencies), memory dependencies, dependencies to maintain function call ordering, and the dependence between a conditional branch and the setting of the condition code. Routines for building the data dependence graph are found in `sched-dep.c`.

Next, instruction priorities are calculated. The priority of an instruction dictates the order in which it may reside on the *ready list*, or the list of instructions whos dependencies have been resolved and are available for scheduling. Priorities are calculated in reverse order beginning with a basic block's tail instruction and ending with the head instruction. The priority of an instruction is found by summing the latency of the instruction and the maximum priority of any dependent successor instruction. This has the effect of exposing the longest dependency chain, giving those instruction along the critical path highest priority.

Finally, after finding dependencies and calculating priorities, `schedule_block()` is called for each basic block within the region to perform list-scheduling. During the scheduling process instructions are added to the ready list when their dependencies are resolved. Dependent instructions that become ready, but do not reside in the current block, may be added to the ready list if the current block dominates the block in which the potentially speculative instruction resides. The flow probability of a speculative instruction is an important factor to consider when performing interblock motion. Over speculation may delay the critical path or increase contention for resources, while under speculation may result in missed opportunities for increasing parallelism.

The ordering of the ready list is an important factor to consider when list-scheduling. If multiple instructions share the same priority, attributes of these instructions, such as register pressure, affect later scheduling decisions. Choosing between these instructions plays a critical role in finding the optimal schedule. The algorithm for sorting instruction in the ready list is as follows:

1. select the instruction with the highest priority, ties broken by
2. select the instruction which least contributes to register pressure, ties broken by
3. prefer in-block upon interblock motion, ties broken by
4. prefer useful upon speculative motion, ties broken by
5. choose the instruction with the highest flow probability, ties broken by
6. choose the instruction which is least dependent upon the previously scheduled instruction, ties broken by
7. choose the instruction which has the most instructions dependent upon it, or finally
8. choose the instruction with the lowest UID.

Sorting instructions based on this algorithm maximizes the opportunity for parallelism while minimizing the length of the critical path.

3 Treegion Formation

This section describes the two step process of treegion formation. First, natural treegions based on the original CFG are formed. Then, the ICSE heuristic is applied to perform tail duplication.

3.1 Natural Treegion Formation

Natural treegion formation begins at the entry block of a procedure, which forms the root of a new treegion. Starting at this root, the CFG is traversed and successor basic blocks are absorbed into the treegion if they are not a *merge point* (*i.e.*, have multiple predecessor edges). Eventually all successor blocks that do not contain merge points are consumed by the treegion and only leaf nodes remain. These leaf nodes, referred to as *saplings*, are then added to a *saplings list*. Saplings form the roots of new treegions. For each sapling the same process is applied until all basic blocks in the CFG have been consumed.

Figure 1 shows pseudo-code for finding natural treegions. Initially the saplings list includes only the successor to the `ENTRY_BLOCK` of the current procedure. This basic block is then removed from the saplings list to form the root of a new treegion. Next, all successors of the root node are added to the successor edge list. Each edge in the edge list is then traversed in breadth first order to absorb successor blocks into the newly formed treegion. Backedges are not traversed to prevent the forming of cyclic regions. Traversal also ends at the `EXIT_BLOCK`. If the current basic block has multiple predecessor edges (*i.e.*, `EDGE_COUNT(edge->preds) > 1`) the node is added to the saplings list and its successor basic blocks are not considered for inclusion in the current treegion. Finally, if the current node is absorbed into the treegion all its successor edges are added to the successor edge list.

```
find_treegions (void)
{
 add ENTRY_BLOCK->succs to saplings;
 while(more saplings)
 {
  node = first set bit (saplings);
  treegion += node;
  edge_list += node->succs;

  while(more edges)
  {
   curr_edges = edge_list[];
   curr_node = curr_edges[]->dest;

   while(more succ in curr_edges[])
   {
    /* Dont traverse backedges */
    if(edge->flags & BACK_EDGE)
     continue;

    /* Skip Exit Block */
    if(curr_node == EXIT_BLOCK)
     continue;

    /* Add merge to saplings */
    if(EDGE_COUNT(edge preds) > 1)
    {
     SET_BIT (saplings, curr_node);
     continue;
    }

    /* Add node to treegion */
    treegion += curr_node;

    /* Add succs to edge list */
    edge_list += curr_node->succs;
   }
  }
 }
}
```

Figure 1: Pseudo-code for natural treegion formation

Figure 2 shows an example CFG after treegion formation. The size and number of treegions is based on the layout of the CFG, not profile information. From figure 2 it can be seen that for any block in a treegion all predecessor blocks

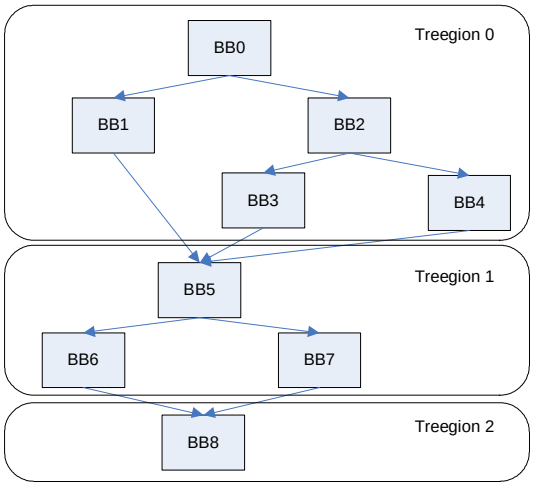


Figure 2: CFG after treegion formation

dominate it. In section 5.2 further optimizations based on dominator parallelism are discussed. It is also important to note that speculatively scheduled instructions are never duplicated because treegions do not contain merge points. For our chosen benchmark suite the average natural treegion contains 2.65 blocks and 20.89 instructions. For these regions, on average 3.65 instructions are speculatively scheduled.

4 Tail Duplication

Tail duplication is performed in order to increase region size providing more opportunity for speculation. However, overly aggressive duplication has the potential to negatively impact the performance of the instruction cache and TLB. This section begins by presenting the tail duplication implementation, with treegions being the unit of duplication. Then an efficient technique for deciding upon when to apply tail duplication is presented. This metric, referred to as the Instantaneous Code Size Efficiency (ICSE), is defined as the change in IPC relative to the change in code size after tail duplication.

For each edge between a pair of treegions the ICSE is calculated to determine if the duplication of the child treegion will be beneficial.

4.1 Tail Duplication Example

The tail duplication process begins by calculating the ICSE of each candidate, discussed in subsection 4.2. Each control edge between a parent and child treegion is a potential candidate with the child treegion being the target for duplication. After calculating all ICSEs, the best candidate is selected for duplication if it is above the ICSE threshold. If no more candidates are available for duplication then the scheduling process may begin.

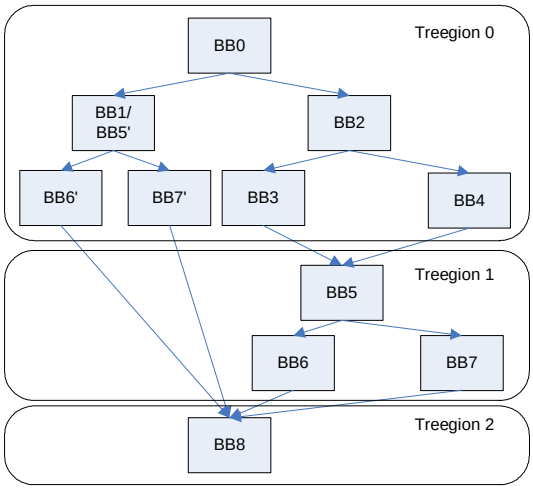


Figure 3: Duplication of candidate edge between BB1 and BB5

Continuing with the example in Figure 2, Figure 3 depicts the result of selecting the candidate edge between basic blocks 1 and 5. Blocks 5, 6, and 7 in the child treegion, treegion 1, are duplicated. These duplicated blocks, denoted with tick marks, are then absorbed into treegion 0. After calling `cleanup_cfg()` basic blocks `BB5'` and `BB1` are merged into a single block. Tail duplication continues until either no more candidates exist or no more can-

didates are above the ICSE threshold. Under code size or compile time constraints, treegion size may also be limited by the number of basic blocks and/or the number of instructions contained with in the treegion. Compilation flags for constraining tail duplication and region formation are discussed in subsection 4.4.

4.2 Instantaneous Code Size Efficiency

In previous work Zhou *et al.* [4] have shown that for a minimal code size increase (~2%) a significant speedup can be obtained. Furthermore, duplication beyond that of the initial code size increase produces only small additional gains in performance. Due to these facts the ICSE equation was developed and is as follows:

$$Efficiency = \frac{IPC_{after\_td} - IPC_{before\_td}}{code\_size_{after\_td} - code\_size_{before\_td}} \quad (1)$$

In equation 1, IPC_{before_td} and IPC_{after_td} refer to the instruction-per-cycle (IPC) ratio of a treegion before and after the application of tail duplication, respectively. $code\_size_{after\_td} - code\_size_{before\_td}$ refers to the change in code size due to tail duplication. Equation 1 requires the IPC of a region to be known at compile time. Since this information is not available, a heuristic is used to estimate the execution time of a treegion, defined as follows:

$$Exec\_Time = \sum_{path_i} [Max(ddb_{path_i}, rb_{path_i}) * freq_{path_i}] \quad (2)$$

The estimated execution time of a multi-path treegion is defined as the sum of the expected execution time of each path through a treegion biased by the execution frequeny of each path. The execution frequency of each path, $freq_{path_i}$, is determined through profiling. If profile information is not available, GCC uses a number of heuristics to approximate the execution frequencies. The expected execution time of any path is the maximum of the data dependence bound, ddb_{path_i}, and the resource bound, rb_{path_i}.

The data dependence and resource bounds are found using similar techniques as those used during modulo scheduling [5] to find the minimum initiation interval (MII). For a given treegion, the data dependence bound is calculated as the height of the longest true-dependency chain in the DDG. The resource bound is computed as the number of instructions in the treegion divided by the issue width of the target machine.

4.3 Tail Duplication Implementation

Figure 4 shows a partial call graph for the main tail duplication function, `td_treegions()`. The `td_init_candidates()` function is first called to calculate the ICSE for all possible tail duplication candidates. Prior to calling `td_add_candidate()`, candidates that exceed user defined parameters (*e.g.*, maximum number of basic blocks per region) are eliminated to restrict the formation of excessively large regions. This prevents compile time from becoming exceedingly long.

Next, `td_classify_candidate()` is called to classify the candidate into one of four possible types. The classification is based on two factors: (1) the number of predecessor edges entering the child treegion and (2) the number of parent treegions the child possesses. These two factors strongly influence efficiency. For example, assume there exists two edges between a parent treegion *A* and a child treegion *B*. No additional predecessor edges are

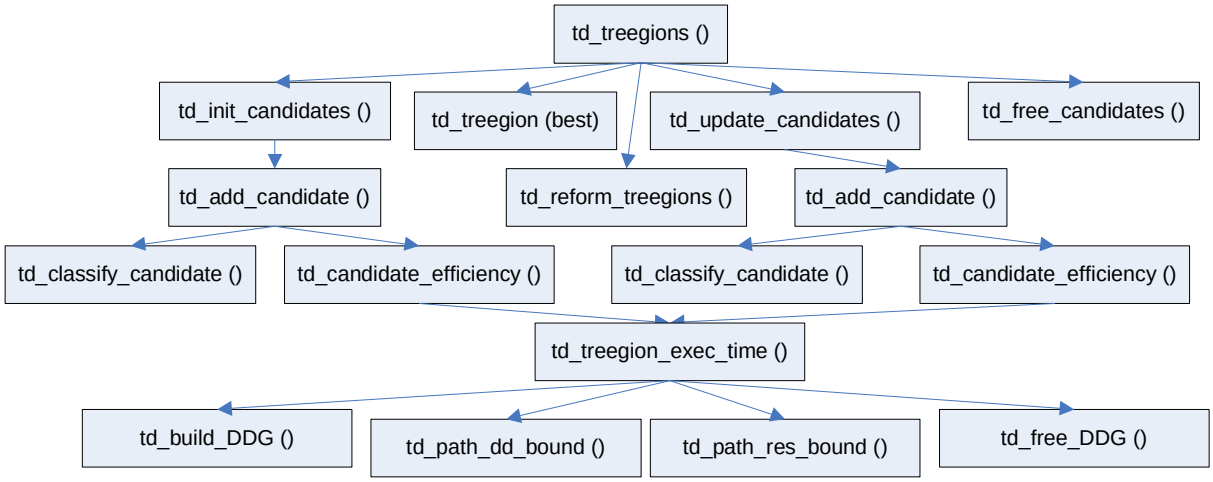


Figure 4: Partial call graph for tail duplication code

entering treegion *B*. This implies treegion *A* is the lone parent of treegion *B*. After duplicating the child treegion, denoted *B'*, both treegion *B* and treegion *B'* can be merged into the parent treegion *A*. This type of duplication produces much larger regions relative to a small code size increase. Alternatively, if three edges are shared between the parent and child only the duplicated treegion *B'* can be merged into the parent treegion *A*.

After classifying the candidate, `td_candidate_efficiency()` is called to calculated ICSE. Based on the type of candidate the estimated execution time, as defined in Equation 2, is calculated by `td_treegion_exec_time()`. The estimated execution time is used to approximate the change in IPC before and after tail duplication. To find the resource and data dependence bounds of a treegion the DDG must first be built. This is done using the routines found in `sched-dep.c`. The treegion is then traversed in depth-first order. For each unique path through the treegion the `td_path_res_bound()` and `td_path_dd_bound()` functions are called to find the maximum bound.

Once all ICSEs have been calculated the best candidate is selected for duplication. After duplication, `td_reform_treegions()` is called to incrementally update the data structure of each effected treegion. `td_update_candidates` is then called to recalculate the ICSE for each effected treegion. This incremental updating process is critical for minimizing compile time. After all possible candidates have been duplicated, `td_free_candidates()` is called to free all tail duplication related data structures. Finally, `cleanup_cfg()` is called to optimize the CFG and merge basic blocks. Scheduling then begins after calling `find_treegions()` again due to the fact the calling of `cleanup_cfg()` invalidates all region related data structures.

4.4 Compilation Parameters

Compile time is an important consideration for a production level compiler. Various compilation parameters can be used to limit compile time as well as fine tune the performance of the application being compiled. These parameters are as follows:

1. *max-sched-region-blocks* - limit the size of

the region based on the number of basic blocks.

2. *max-sched-region-insns* - limit the size of the region based on the number of instructions.

3. *treegion-max-code-growth* - limits tail duplication based on a maximum amount of code growth.

4. *treegion-icse-threshold* - sets the ICSE threshold. Prior work [4] has shown the optimal range to be between 0.268 and 0.577. A higher threshold results in less duplication.

5. *min-spec-prob* - the minimum probability of reaching a source block for interblock speculative scheduling.

5 Treegion Scheduling

Due to the acyclic nature of treegions, the Haifa scheduler does not require any modifications to accomodate treegions. However, in this section various modifications are proposed to enhance the performance of the scheduler.

5.1 Tree Traversal Scheduling

The goal of Tree Traversal Scheduling (TTS) [6] is to speedup every execution path through the treegion. This is accomplished by prioritizing speculative instructions from different paths which compete for limited resources. Profile information is used to prioritize the scheduling of basic blocks within a treegion.

The algorithm for tree traversal scheduling is as follows:

1. For a treegion, sort the basic blocks according to a depth-first traversal order with the child block selected with the highest execution frequency.

2. Begin list scheduling blocks at the root basic block.

3. During the scheduling of a basic block, consider speculation for instructions dominated by this basic block.

4. Repeat step 3 until all basic blocks in the treegion have been scheduled.

The primary strength of Tree Traversal Scheduling is that the frequently executing path is given highest priority, while the less frequently executing paths are not severly penalized.

5.2 Operation Combining (Future Work)

The application of tail duplication enables the removal of merge point between treegions, producing larger regions. However, despite the benefits, tail duplication has the potential to decrease the performance of the instruction cache and TLB due to the creation of many redundant instructions. In some instances the instruction scheduler can take advantage of *dominator parallelism* to remove redundant operations at schedule time.

Dominator parallelism [7] presents itself when an instruction is speculatively scheduled into a predecessor block that dominates blocks containing redundant copies of the scheduled instruction. In these instances, a form of schedule-time partial redundancy elimination (PRE), also referred to as operation combining, may be applied to remove all but the speculatively scheduled copy of the instruction. The

single remaining instruction performs the operation for all paths. If the instruction is redundant in every control path below the target block the instruction can be made non-speculative.

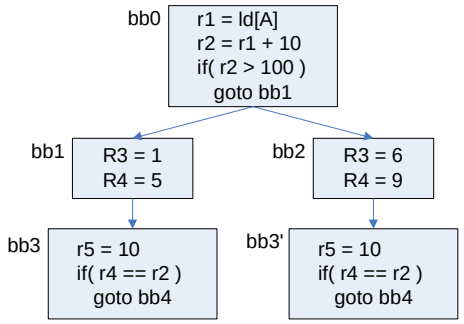


Figure 5: Example of operation combining within a treegion

Figure 5 depicts an example treegion after the duplication of basic block `bb3`, denoted `bb3'`. During the scheduling of `bb0`, the instruction `r5 = 10` can be speculatively moved from both blocks `bb3` and `bb3'` into basic block `bb0`. Assuming `r5 = 10` has already been hoisted from `bb3`, the redundant copy from `bb3'` may be safely eliminated. The scheduler can easily detect this optimization due to the characteristic that any basic block within a treegion dominates all its successor blocks. Any instruction speculated upward is always moved into a dominator. Therefore, if an instruction is speculated into a block where a redundant copy of the instruction has already been scheduled, one copy can be removed.

6 Experimental Results

Experiments were performed to evaluate the performance of the Treegion instruction scheduler. All tests were conducted on an Itanium 2 processor. The benchmark suite consisted of a subset of benchmarks from the SPEC2K suite including: *gzip*, *mcf*, *crafty*, *parser*, *gap*, *bzip2*, *twolf*, *wupwise*, *swim*, *mgrid*, *applu*, *equake*, *ammp*, *sixtrack*, and *apsi*. Profile information was generated using the `-fprofile-arcs` flag and all benchmarks were compiled using the `-O3` and `-fbranch-probabilities` flags. Flags set to control speculation include `-fsched-interblock`, `-fsched-spec`, and `-fsched-spec-load`.

Table 1 presents various region related statistics for the original region formation code, natural treegion formation, treegion formation with tail duplication bounded by an ICSE threshold of 0.577, and treegion formation bounded by a maximum of 100 instructions per region, respectively. The current region formation code produces and average region size of 1.10 basic blocks, containing 8.66 instructions of which 0.09 were speculatively scheduled. Due to the limited scope of the region the opportunity for speculation is limited. Natural treegions, *i.e.,* treegions without tail duplication, produce an average region size of 2.65 basic blocks, containing 20.89 instructions of which 3.65 were speculatively scheduled. Even without the application of tail duplication natural treegions provide greater opportunity for parallelism. Limiting duplication to a ICSE threshold of 0.577 produces only slightly larger regions beyond that of natural treegions. Finally, applying unlimited tail duplication while limiting region size to a maximum of 100 instructions produces an average regions size of 5.70 basic block, containing 35.95 instructions of which 6.00 were speculatively scheduled.

Table 2 shows the speedups for the various region formation techniques. Speedups are relative to basic block scheduling. The execution time of each SPEC benchmark was found by averaging five runs using the *ref* input set. The speedup results vary across benchmarks. The original region code produces speedup for *parser*, *twolf*, and *ammp* while *gap* and

	Region	Natural Treegion	Treegion (k = 0.577)	Treegion (100 insns)
# Basic Blocks	1.10	2.65	2.79	5.70
Instructions	8.66	20.89	21.70	35.95
Interblock	0.09	3.65	3.81	6.00

Table 1: Region statistics

	Region	Natural Treegion	Treegion (k = 0.577)	Treegion (100 insns)
gzip	1.00	0.96	0.96	1.03
mcf	1.00	1.00	1.00	1.00
crafty	1.00	0.99	1.00	1.00
parser	1.01	1.01	1.01	1.01
gap	0.99	1.01	1.00	1.00
bzip2	0.99	1.06	1.06	1.06
twolf	1.03	1.01	1.01	1.03
wupwise	1.00	0.99	1.02	1.01
swim	1.00	1.02	1.04	1.01
mgrid	1.00	0.99	0.99	1.00
applu	1.00	1.00	1.00	1.00
equake	1.00	1.00	1.01	1.00
ammp	1.01	1.01	1.00	1.00
sixtrack	1.00	0.98	0.98	0.98
apsi	1.00	1.01	1.01	1.01
average	1.00	1.00	1.01	1.01

Table 2: Speedup results

bzip slowdown. Natural treegions produce a speedup for seven of the fifteen benchmarks, slowdowns for five of the benchmarks, while three benchmarks remain uneffected. The most significant speedup (6%) is for *bzip2*. For *wupwise*, *swim*, and *equake* the best performance gain is realized using the ICSE threshold. Applying unlimited tail duplication while limiting region size to 100 instructions produce a speedup for seven of the fifteen benchmarks, slowdown for only *sixtrack*, while seven benchmarks remain uneffected. On average the original region formation code and natural treegion formation provide no speedup, while treegion formation with tail duplication bounded by ICSE and treegion formation bounded by instruction count produce and average speedup of 1%.

7 Conclusions

This paper presents the status of the implementation of an architecture-independent, aggressive global instruction scheduler based on Treegions. Natural Treegion formation and tail duplication have been completed and are currently maintained on the sched-tree-branch. The ICSE heuristic has also be implemented as

a means of judiciously applying tail duplication. To ensure compile time does not become exceedingly long fine tuning of this code is an ongoing process. Also, while tail duplication has the benefit of increasing region size, it does introduce redundant instructions. Finally, operation combining is presented as future work for eliminating redundant instructions as schedule time.

Our results show that treegion formation dramatically increases the average region size as compared to the current region formation code. This in turn results in a significant increase in the number of speculatively scheduled instructions. Our results show performance benefits for a few benchmarks (i.e., *parser*, *bzip2*, *wupwise*, *twolf*, *swim*, and *apsi*) while others show little improvement because of architectural features such as memory latencies that hide scheduling improvements. Techniques such as software prefetching should be able to alleviate such issues resulting in future performance gains from Treegion scheduling.

8 Acknowledgments

Thanks go to Diego Novillo and Gerald Pfeifer for their assistance during the opening of the Treegion scheduling branch. Thanks also to TINKER members Balaji Iyer, Paul Bryan, Jesse Beu, and Saurabh Sharma, for their helpful insight.

References

[1] J. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," in *IEEE Transactions on Computers*, pp. 478–490, 1981.

[2] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *Journal of Supercomputing*, 1993.

[3] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective Compiler Support for Predicated Execution using the Hyperblock," in *25th Annual International Symposium on Microarchitecture*, 1992.

[4] H. Zhou and T. M. Conte, "Code Size Efficiency in Global Scheduling for ILP Processors," in *Proceedings of the 6th Annual Workshop on the Interaction between Compilers and Computer Architectures (INTERACT-6) held in conjunction with the 8th International Symposium on High Performance Computer Architecture (HPCA-8)*, (Cambridge, MA), February 2002.

[5] M. Hagog and A. Zaks, "Swing Modulo Scheduling for GCC," in *The 2004 GCC Developers' Summit*, (Ottawa, Canada), June 2004.

[6] H. Zhou, M. D. Jennings, and T. M. Conte, "Tree Traversal Scheduling: A Global Scheduling Technique for VLIW/EPIC Processors," in *Proceedings of the 14th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, (Cumberland Falls, KY), August 2001.

[7] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 1986.

Improving Software Floating Point Support

Nathan Sidwell
CodeSourcery Inc
nathan@codesourcery.com

Joseph Myers
CodeSourcery Inc
joseph@codesourcery.com

Abstract

GCC's runtime library contains a set of software floating point routines, to be used when the required operation is not available in hardware. These routines have not been significantly optimized, and software floating point performs more poorly than it could. We discuss various pitfalls in their implementation. The GNU C library, `glibc`, also contains software floating point routines, and those have been optimized reasonably well. We show performance numbers obtained from portions of the EEMBC benchmark running on two PowerPC systems comparing the routines from the two libraries. We discuss the incorporation of the `glibc` routines into GCC's runtime library, and show how to convert other backends to use the new `glibc` routines.

1 Benchmarks

Our initial goal was to improve the performance of a subset of EEMBC[1] benchmarks running on PowerPC 405 and 440 hardware without using floating point instructions. The EEMBC benchmarks consist of a few sets of tests targeted at particular application areas—automotive, office, networking, consumer, etc. We used a subset of the automotive, networking and consumer sections. The automotive suite is particularly floating point intensive. After obtaining baseline benchmark numbers, we profiled the suite and examined each test's profile. Nearly all of the automotive tests spent some time in floating point routines. Five of the 16 automotive benchmarks spent significant time in a few floating point routines. Table 1 tabulates the time spent by those benchmarks in the floating point routines. It tabulates every routine accounting for more than 2% of the total processor usage. As can be seen, the benchmarks that use floating point spent considerable time in the floating point library. The final column is the geometric mean of the of the fraction of execution time for those benchmarks that showed usage. Mathematically, that is not a robust calculation because of the arbitrary 2% cutoff. However, it does give a guideline as to which routines are important.

2 Floating Point Libraries

GCC contains an implementation of software floating point in `fp-bit.c` and associated files. These implement the regular IEEE 754[2] operations of addition, subtraction, multiplication, division, comparison and conversions. Through the use of macros, `fp-bit.c` is used to generate `float`, `double`, and `long double` routines. In this paper, `fpbit` refers to the combined `float` and `double` routines

Testcase Routine	basefp01	matrix01	a2time01	tblook01	iirflt01	Geometric Mean
__floatsidf			63.1%	18.1%	55.6%	39.9%
__muldf3	29.0%	32.42%	6.4%	13.3%		16.8%
__divdf3	12.7%					12.7%
__pack_d	16.3%	19.70%	7.4%	16.8%	4.7%	11.3%
__unpack_d	12.4%	15.74%	6.6%	9.6%	3.5%	8.5%
_fpadd_parts	22.1%	20.52%		3.9%		7.7%
__pack_f				5.8%		5.8%
__subdf3		4.45%				4.5%
__extendsfdf2				2.9%		2.9%
__adddf3	2.4%					2.4%
__divsf3				2.2%		2.2%
Total	94.9%	92.83%	83.5%	72.6%	63.8%	

Table 1: Benchmark Profiles

of these files. In addition, `libgcc2.c` contains some conversion routines which are used in certain circumstances.

Our initial plan involved optimizing `fpbit` itself. There are a number of improvements that can be made, and we estimated they would probably give a factor of 2 speedup on some of the routines. However, it came to our attention that an alternative library had already been proposed. Torbjorn Granlund submitted `ieeelib`[3] some time ago, but it had never been integrated. `ieeelib` implements many of the ideas we had for `fpbit`. We experimented by using it for the benchmarks and found that it gave a speedup of around 25% on EEMBC. Integrating `ieeelib` would be a better way forwards than improving `fpbit` itself.

Following this, we realised that `glibc`[4] also contained software floating point routines. Again we experimented with a version of GCC containing those routines and found it gave an improvement of around 20%. Clearly `ieeelib` and `glibc` were both candidates for integration. There were a number of advantages of each library:

- `ieeelib` has a smaller footprint than `glibc`.
- `glibc` contains support for different rounding modes, including runtime selection of the rounding mode. (The benchmarks we performed hardwired the rounding mode, so the comparison was comparing like for like features.)
- `glibc` has support for floating point exceptions, even integrating these into the hardware, when that is feasible. (Again, we made the above measurements with this disabled.)
- Using `glibc` routines would reduce the number of different software floating point implementations in GNU software.

This last point was very attractive. Reducing the number of implementations of software floating point would reduce maintenance. As we discovered, by uncovering some bugs both latent and otherwise, writing correct floating

point code is tricky. Therefore, having a common implementation would improve software quality, because if a bug was found in either `glibc` or GCC, the patch could be applied to both.

The size difference between `ieeelib` and `glibc` seemed disadvantageous to `glibc`. Table 2 shows the sizes of `fpbit`, `ieeelib` and `glibc` routines. As can be seen, the first two are in the same ballpark, whereas the `glibc` routines are much larger.

Analysis showed there to be two causes of this. Firstly `glibc` has separate addition and subtraction routines, and secondly its multiplication and division routines are larger. Both of these turn out to have the same cause, namely correct support for NaNs, rounding modes and exceptions. Even though we had disabled as many additional features as possible, their effects were still present. We thought that it would be possible to improve the `glibc` code size somewhat, but were not sure how far the tendrils of the optional features could be removed. In the worst case, we felt that on a modern system an additional 6–7K bytes is not as significant as it used to be.

Additional advantages of `glibc` are its control of rounding mode and support for exceptions. Indeed, it could provide dynamic control of the rounding mode, which is desirable in some contexts. Although we were not immediately concerned with this, we were sure that others would be.

We decided that merging the `glibc` routines would be a technically better solution, and chose to pursue it.

However, there was a license issue; `glibc` is licensed under the LGPL[5], whereas the compiler's floating point emulation routines need to be licensed with runtime exception. That is, although the implementation of the routines can be licensed under the LGPL, merely linking them into a program as part of GCC's runtime support should not bring that program under the requirements of the LGPL (of course, this would not invalidate any other reason why the (L)GPL might apply). As the FSF[6] is the copyright holder of both `glibc` and GCC, only they could make the decision to allow the runtime exception for the `glibc` routines. We presented the technical arguments to the FSF, and persuaded Richard Stallman to allow a change of license. The FSF approved the use of LGPL plus runtime exception for the copies in *both* `glibc` and GCC. This means that the source files can be identical in both places, rather than having to add the runtime exception license wording to only the GCC copies.

3 Unpacking IEEE Numbers

The primary failing of `fpbit` is in its packing and unpacking of floating point numbers. Nearly all its deficiencies are artifacts of how this is done.

All `fpbit` routines commence by fully unpacking the floating point number's mantissa, exponent and sign into separate fields of a structure. In addition they determine the number's category as one of zero, denormal, signalling NaN, quiet NaN, infinity or regular number. The unpacked exponent is unbiased and the unpacked mantissa has the implicit 1 bit inserted. Denormals are scaled to be consistent with the regular representation of $-1^S \times M \times 2^E$. Apart from clearly taking time, this unpacking has an immediate deficiency. Firstly it means the floating point routines use structures, and pass them by address, thereby forcing these parameters to be passed in memory with all the slowdown that entails.[1] A second deficiency

[1] GCC's structure splitting optimization is inapplica-

Library	`fpbit`	`ieeelib`	`glibc`
Text size (bytes)	10688	9284	16940

Table 2: Library Sizes on PowerPC 440

is more subtle. The complete categorization makes the floating point routines begin with several separate checks for the rare categories. For instance the code of `_fpmul_parts` (the core of the multiplication routine) begins with:

```
if (isnan (a)) ...
if (isnan (b)) ...
if (isinf (a)) ...
if (isinf (b)) ...
if (iszero (a)) ...
if (iszero (b)) ...
```

This is actually more checks than is immediately apparent, because `isnan` checks for both quiet and signalling NaN categories. Naturally these checks have to be done, but the IEEE encoding uses only two special exponent values (zero and all ones) to encode all of the non-normal numbers. Thus it would be possible to have an early check for the two special encodings, and then determine which specific non-normal encoding has occurred out of the mainline of the routine. In fact, in the case of `_fpmul_parts`, this separation of all the distinct special cases is not necessary because *all* the separate `if` bodies are identical, or nearly so! This is a classic case of optimizing for the rare condition.[2] Before we started investigating `ieeelib` and `glibc` we made a 0.4% improvement by adding `__builtin_expect` calls to the various `isnan` and `isinf` macros so the compiler can optimize the expected code path.

ble here, as the packing and unpacking routines are not inlined.

[2]Implementers of networking stacks have discovered that early and complete unpacking of packet headers is a pessimization for the same reasons.

4 Improvements to `glibc`

We made a number of improvements to `glibc` in order to bring its performance, where it was deficient, up to that of `ieeelib`.

4.1 Unpacking

Some `glibc` routines do partial unpacking to obtain an exponent, sign and mantissa, thereby not having the pessimization described in Section 3. The exponent encodes the special values of interest. Most do further classification, but using macros and separate local variables instead of functions and structures, and using `switch` statements to reduce the number of checks. Depending on the operation being performed, `ieeelib` is even more specialized; it might do a minimal unpacking. It also never bothers adding in the implicit one bit in its expanded form. This turns out to be significant for float conversion operations and addition and subtraction where `ieeelib` was much faster than `glibc`. We patched `glibc` to perform minimal unpacking in these cases and improved its performance to be similar to that of `ieeelib`.

4.2 Addition and Subtraction

As mentioned, `glibc` has separate addition and subtraction routines. Naively it would seem that they could be trivially combined. Unfortunately this turns out to be difficult to achieve, because of the need to generate the correct NaN value in certain circumstances. A

set of lower level macros, which provide target specific features, are used to construct glibc's routines. In the case of addition and subtraction, the macros of interest are _FP_ADD_INTERNAL and _FP_CHOOSENAN. The subtraction routine inverts the sign of the subtrahend *unless* it is a NaN. Then it calls _FP_ADD_INTERNAL. Addition simply invokes _FP_ADD_INTERNAL. This would suggest a simple merging scenario, but unfortunately:

- There is an excess of state to simply call an underlying routine efficiently.
- _FP_ADD_INTERNAL takes an operation parameter so that target specific code can return a different NaN for each operation, in the case of an exception.

Removing the excess state could be achieved by not bothering to detect a NaN subtrahend, and simply inverting its sign in all cases. This would change the behaviour of 'F − NaN'; rather than returning 'NaN', it would return '−NaN'. For GCC's purposes, the operation parameter is unimportant, but removing it would probably break our requirement that the GCC copy be readily updateable from the glibc sources. It also appears that glibc itself only uses this operation parameter for the x86 and x86_64 targets, where the software floating-point code is not actually used. However, the same glibc code is used in the Linux kernel math emulation, where consistency with hardware choice of NaN is required. It is unfortunate that this single target family causes such difficulty. It is possible that the distinction is unnecessary in even these two cases, in which case the operation parameter could be removed, and much simplification achieved. We decided to leave this as an open issue, keeping the addition and subtraction routines separate.

4.3 Bit Shifting

One significant change we made to fpbit before we started porting glibc was to use __builtin_clz to find the most significant set bit in integer to floating point conversion routines. This yielded a 7% performance improvement on EEMBC. We made the same changes to glibc where we replaced hand coded asm inserts with __builtin_clz, leaving it to the compiler to determine the most efficient code sequence.[3]

4.4 Other Patches

As part of preparing the glibc code for integration into GCC, support was added for the new floating point functions that had been inserted into fpbit since 1999. Functions were changed to use typedefs, such as SFtype instead of float. Additionally many changes were made to reduce the number of compiler warnings generated by the code. Some of these resulted from the heavy use of macros in the glibc code where unreachable code caused unwanted warnings.

4.5 Bug Fixing

In addition to improving glibc's performance, we uncovered a number of implementation bugs. This bolstered our thesis that software floating point can be tricky, and using the same implementation in both glibc and GCC

[3] In general glibc contains a large number of assembly inserts for 'optimized' code sequences. These might have produced better code than GCC in the past, but we now find the compiler producing better sequences. Worse, we have discovered that an assembly insert might be subtly wrong in that it does not describe the side effects or constraints correctly, leading to incorrect code generation with GCC's better optimizers.

would improve both. We found these bugs both through code inspection and the use of testsuites. Firstly, the GCC testsuite found some problems with the `glibc` routines. Secondly, we used the `ucbtest`[7] testsuite, which is designed for checking awkward IEEE cases. The bugs found and fixed in `glibc` include the following:

- Undefined behavior involving signed integer overflow.
- Undefined behavior involving shifting integers by the width of their type.
- Conversion of `float` to `long long` could left shift by a negative amount.
- Conversion of `long long` to `float` used a macro on `long long` values that only worked correctly on values of size `_FP_W_TYPE_SIZE` (typically `sizeof long`).
- An off-by-one-error in integer to floating point conversion when the integer value had exactly one more bit than the number of floating point mantissa and guard bits. For example, converting 3×2^{26} to `float` yielded 2^{28}.
- Incorrect exceptions were set in various cases.

We also found and fixed bugs outside of `glibc`:

- In EEMBC—reliance on undefined behavior of out-of-range floating point to unsigned integer conversions.
- In `fpbit`—a latent bug in a previously unused function causing incorrect rounding.
- In `libgcc2`—conversions of `TImode` (128-bit) integers to floating-point values had fundamental bugs.

5 Results

The EEMBC benchmarks report a number of values for each test. The value we used to measure improvement was the number of iterations per second. Because EEMBC reports iteration times as an integral number of microseconds, precision is lost with that more obvious measure of speed. As all the different tests have not been weighted against each other, we used the geometric mean in order to give each test equal weighting.[4] As stated earlier, we restricted our measurements to the automotive, networking and consumer subsections of the EEMBC suite. For the 405 benchmarks we used `-mcpu=405 -O2` and for the 440 benchmarks we used the `-mcpu=440 -O2` optimization flags. In addition to the floating point changes, we improved `strlen` and 16 bit multiplication by adding support for additional instructions. These particular benchmarks do not appear to make use of those features, and we believe the entire performance improvement shown here is due to the software float changes. Table 3 enumerates the before and after iteration counts and the speedup achieved. *Note, these are not official EEMBC benchmark results, and may be used as a speedup guide only.* As can be seen, the most improved test case's performance increased by nearly 360%.

6 Using the `glibc` Routines

We have imported the `glibc` routines into GCC. The primary source for these routines remains `glibc`, and any fixes to GCC's copy

[4] Using an arithmetic mean would unfairly bias the work to improving the speed of the longer benchmarks. There is no evidence that the longer iteration times are anything other than an artifact of the particular test being performed.

Benchmark	Tests	405			440 softfp		
		Before	After	Speedup	Before	After	Speedup
basefp01	1	3226.5	8762.2	2.72	13205.7	35550.5	2.69
matrix01	1	20.1	44.5	2.21	78.2	183.8	2.35
a2time01	1	26264.0	115293.7	4.39	97561.0	448129.1	4.59
tblook01	1	11009.2	23158.3	2.10	40566.3	97924.0	2.41
iirflt01	1	19656.4	60975.6	3.10	73432.2	234521.6	3.19
Automotive	16	13361.2	18445.2	1.38	52301.0	73497.2	1.41
Consumer	5	29.7	29.8	1.00	106.7	108.7	1.02
Network	6	802.9	806.3	1.00	2390.7	2386.9	1.00
Combined	27	2308.1	2797.2	1.21	8366.4	10266.2	1.23

Table 3: Benchmark Numbers

needs to be sent upstream to glibc. Fortunately the sources are identical in both GCC and glibc, because of the identical license change in both places.

The integration of the glibc routines into GCC has been designed to make it easy to start using these routines for new targets. Whereas fpbit uses special case code in mklibgcc.in, glibc uses the existing target makefile fragment mechanism. GNU Make features are used in t-softfp to select the functions required on a given target. In addition to defining the variables used by t-softfp, a file called sfp-machine.h must be provided for each target. Initial versions of this file for many targets are already located in the appropriate sysdeps/ARCH/soft-fp directory of glibc.

A target may specify the floating point and integer modes for which functions are to be compiled. The conversions between floating point modes to support may also be specified. This allows for targets with some hard-float and some soft-float modes. For those, glibc code will be used for conversions between the hard-float and soft-float modes. In such a case, the sfp-machine.h file may define how to raise exceptions and determine the rounding mode for the soft-float modes in a manner consistent with the exception flags and rounding modes provided by the hardware. A case where this might be useful in future is to support the optional __float128 type in the x86_64 ABI.

7 Future Work

The glibc routines offer the possibility of further improvements. As has been already mentioned, the dynamic control of rounding mode is possible, along with integrating the exception mechanism with that provided by the system's glibc fenv.h interface. The routines could be extended to support the float128 type present in the x86_64 ABI.

Further investigation of the issues involved in merging the addition and subtraction routines could be done, thereby reducing the code footprint.

There currently remains some overlap between the operations provided in the glibc routines and those provided by libgcc2. The glibc routines replace the libgcc2 routines. For a pure soft-float target, this is exactly what is desired, but for a target with hardware floating point, but supporting a variant soft-float ABI,

the `glibc` routines would be used in both sets of multilibs. The `libgcc2` routines will potentially handle rounding and exceptions consistent with the hardware floating point. This can be solved by implementing the above mentioned rounding and exception control to the `glibc` routines, and expunging the `libgcc2` routines from GCC. This would continue the reduction in the number of different floating point routines.

Further details of the `glibc` routines and suggested further work are available on the GCC Wiki at `http://gcc.gnu.org/wiki/Software%20floating%20point`.

8 Acknowledgements

This work was sponsored by the PowerPC Licensing Team at IBM. We are grateful for the opportunity afforded to speed up this part of GCC's support across all architectures.

References

[1] The Embedded Microprocessor Benchmark Consortium, `http://www.eembc.org`.

[2] Standard for Binary Floating Point Arithmetic, ANSI/IEEE Standard 754–1985.

[3] New IEEE P854 emulation library, Torbjorn Granlund (`tege@swox.com`), `http://gcc.gnu.org/ml/gcc/1999-07n/msg00553.html`

[4] The GNU C Library, `http://www.gnu.org/software/libc`

[5] GNU Lesser General Public License, `http://www.gnu.org/copyleft/lesser.html`

[6] The Free Software Foundation, `http:www.fsf.org`

[7] Testing difficult cases of IEEE 754 floating point arithmetic, David G. Hough (`dgh@validgh.com`) et al., `http://www.netlib.org/fp/ucbtest.tgz`

Low-Level Performance Analysis

Identifying opportunities for improving compiler code generation.

Steven Munroe
IBM Corporation
munroesj@us.ibm.com

Peter Steinmetz
IBM Corporation
steinmtz@us.ibm.com

Abstract

As the clock speeds of modern processors approach the limits of today's manufacturing technologies, current and future performance improvements will rely more on instruction parallelism and compiler exploitation. The designs of these processors include complex trade-offs between maximal clock speed and circuit complexity; assumptions about instruction usage and timing are included in these trade-offs and result in processor stalls if not met. These design points must be considered by compilers to ensure optimal machine performance.

This paper will present methods that compiler writers/performance analysts can use to identify non-optimal code generation sequences, including *hazard-prone* code streams. These strategies are applied to specific examples on a PowerPC POWER5™ processor to illustrate their effectiveness.

Introduction

The highly competitive market in today's leading edge technology sector forces vendors to continually find ways to improve their products. While many factors can play into a customer's decision to purchase one product over another, a key item that drives many decisions is the product's performance.

Numerous industry standard benchmarks exist which allow customers to compare the performance of products from multiple vendors [1]. Thus, it becomes a key marketing focus for technology companies to present optimal benchmark results for their products.

The underlying hardware plays a large role in the benchmarking results for a machine. As modern processors become more complex and approach the physical limitations of the technologies upon which they are built, chip designers are forced to make assumptions and include complex trade-offs during the processor design cycle. Code streams which diverge from these assumptions may contain hazards that result in poor performance. As a result, highly optimized software becomes increasingly important.

Thus, optimizing compilers become a key component to the overall performance of the product. Benchmarks, and applications alike, depend on the compiler to generate code which takes advantage of the hardware characteristics of the machine.

This paper will present methods that a compiler writer or performance analyst can use to identify opportunities for improving the perfor-

mance of an application. It covers well known techniques, but goes a step further by demonstrating how more subtle problems can be identified.

1 Traditional Techniques

1.1 Profiling

Typically, the greatest opportunities for improving the performance of an application lie within the most heavily used, or "hot," sections of code. Profiling tools [2, 4] can be used to identify these hot sections within an application. Once identified, there are various methods of recognizing opportunities for improving them.

For example, an analyst looking to improve the performance of gzip [5] could use gprof [2] to generate a report similar to the following:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self
 time   seconds   seconds .. name
 55.62      6.63     6.63 .. deflate
 15.60      8.49     1.86 .. send_bits
 12.33      9.96     1.47 .. ct_tally
 11.83     11.37     1.41 .. compress_blo
  2.77     11.70     0.33 .. updcrc
  1.43     11.87     0.17 .. build_tree
  0.25     11.90     0.03 .. send_tree
  0.17     11.92     0.02 .. bi_reverse
  0.00     11.92     0.00 .. flush_outbuf
  0.00     11.92     0.00 .. file_read
  0.00     11.92     0.00 .. flush_block
```

Over 55% of the application's run time is spent within a function named "deflate" making it an ideal candidate for additional focus.

Using the same tool, one can drill down deeper and locate the source instructions which are most heavily executed. Using gprof with the `-l` option yields a line-by-line analysis looking something like this:

```
Each sample counts as 0.01 seconds.
  %
 time  .. name
  8.81 .. deflate (deflate.c:477)
  5.96 .. deflate (deflate.c:679)
  3.61 .. compress_block (trees.c:1052)
  3.61 .. deflate (deflate.c:548)
  3.52 .. send_bits (bits.c:141)
  3.44 .. compress_block (trees.c:1052)
  3.10 .. send_bits (bits.c:122)
  3.10 .. deflate (deflate.c:544)
  3.10 .. deflate (deflate.c:437)
  3.02 .. deflate (deflate.c:758)
  2.94 .. deflate (deflate.c:741)
  2.77 .. deflate (deflate.c:738)
  2.77 .. updcrc (util.c:73)
  2.60 .. deflate (deflate.c:686)
  2.52 .. ct_tally (trees.c:987)
  2.43 .. ct_tally (trees.c:965)
  2.35 .. deflate (deflate.c:732)
  1.85 .. deflate (deflate.c:543)
  1.68 .. deflate (deflate.c:675)
  1.68 .. deflate (deflate.c:679)
```

The analyst can now focus his or her attention on very specific sections of the code. The source code and associated assembler code can be examined for weaknesses such as missed loop unrolling or code inlining opportunities.

1.2 Compiler Benchmarking

On machines where more than one compiler is available, generated code sequences can be cross referenced and compared. This aids the analyst in identifying the strengths and weaknesses of each compiler and often leads to opportunities for improvement.

Optimizations performed by default in one compiler, may be disabled by default in another. Comparing the generated code helps the analyst to identify these cases. Tuning the compiler invocation to include optimizations which are beneficial to the code of interest, and disable those that might be harmful can lead to improved code performance.

Again, using gzip as an example, the application is compiled using IBM's VisualAge C compiler using optimization level three (-O3).

Its performance is compared against the same code compiled with gcc-4.1 [7] at optimization level three (-O3). It's observed that the gcc compiled code runs slower than the code compiled with the VisualAge compiler. Using the methods described above, one can determine that a key loop was unrolled by the VisualAge compiler, but not by gcc. By simply adding the -funroll-loops option to the gcc compiler invocation, the key loop is now unrolled and the performance of the compared applications is roughly equal.

Many compiler optimizations, especially those which are machine independent, are driven by various assumptions or heuristics. For example, the number of times a loop is unrolled will depend on the size of the loop, or the number of iterations it is expected to perform. These heuristics are set based on averages which result in good code generation for most cases. Here again, comparing the output of multiple compilers can identify cases where heuristics are set differently. The analyst or compiler writer can then adjust the heuristics to more effective levels for their application.

1.3 Hand Tuning

An experienced performance analyst may reach a point where adjusting compiler options and heuristics results in code with remaining opportunities for improvement. Reasons may include a missing optimization phase in the subject compiler; alternatively, the application may contain a specific "corner case" not previously considered by the optimization's developer.

Missed opportunities are of interest to those looking to improve code generated by a compiler. However, proposed improvements are more readily accepted if empirical evidence is included to support the request.

In cases such as this, the analyst may be required to manually manipulate the code generated by the compiler in order to demonstrate that proposed changes have a measurable effect on the performance of the application.

2 Modern Processor Design

The tools and strategies covered thus far are well known in the industry and do a reasonable job of improving the performance of an application. However, as the clock speeds of modern processors approach the limits of today's technologies, compiler writers need to go a step further to ensure that high degrees of instruction parallelism exist within generated code.

Understanding and identifying the low-level details of the processor design is required in order to fully exploit its capabilities. These details are usually complex and can be challenging for compilers to deal with.

Modern processor designs involve complex trade-offs between the maximum clock speed and circuit complexity. These trade-offs involve assumptions about average instruction usage and timing which can cause delays (execution hazards) if not met.

To support a higher clock rate, execution pipelines are often designed to contain multiple, simple stages. This can mean that even basic instructions require multiple cycles to execute.

The potential for high throughput (i.e. more than one instruction per cycle) exists when multiple independent instructions can be dispatched in parallel. In practice, however, code sequences generally contain instructions that depend on the results computed by previous instructions. If an instruction requires a value

from a prior instruction, the hardware may have to delay the second instruction to insure that its input values are available when needed. This delay or latency should be familiar to most as a pipeline stall or bubble.

Latencies are small, well defined constants for most instruction combinations. It is important for a compiler to know these latencies in order to schedule (rearrange) instructions into an order that minimizes pipeline stalls. Streams of independent code sequences can be overlapped such that instructions from one stream fill the bubbles from the other.

As processor designers push the limits of technology, a higher degree of variance in these latencies becomes prevalent. For example sign extension for sub-word loads, setting the condition code for arithmetic results, or updating the address for auto increment can add one or more cycles to the nominal latency. This complicates the compiler's back end as there is now a complex matrix of instruction latencies to deal with. Furthermore, it complicates the task of identifying places where these latencies are causing performance problems. These types of problems are difficult to locate using the traditional methods described earlier.

Chip circuit densities appear to be increasing as fast or faster than clock frequencies. The trend is to use this circuit density for instruction parallelism with multiple pipelines and the ability to dispatch multiple instructions per cycle. The micro-architecture (the type and number of pipelines) determines which instructions and how many can be dispatched in any cycle. This obviously impacts the compiler which now needs to model parallel executions in the scheduler.

2.1 The POWER5™ Processor

In order to illustrate methods a performance analyst or compiler writer might use to identify more advanced performance issues, we now take a closer look at IBM's PowerPC POWER5™ processor. Tools used to identify execution hazards are also discussed as well as strategies for avoiding these hazards.

IBM's POWER5 processor can dispatch up to five instructions per cycle into six separate issue queues. The microprocessor core contains eight pipelines which select instructions in an out-of-order fashion from the issue queues. The pipelines are dedicated into functional categories; load/store x 2, fixed point x 2, floating-point x 2, branch, and condition register/logical. The core tracks the dispatch groups through execution such that it can complete instructions in an in-order fashion.

The rules for forming dispatch groups are too complex to describe here but it is fair to say that sustaining multiple instructions per cycle is a non-trivial exercise. For example: some instructions are "cracked" by the core into two simpler instructions, each of which may be issued to a different pipeline. Cracked instructions take two slots in a dispatch group. Other instructions have additional restrictions and must be dispatched singly. Still others must be dispatched from specific slots (first or last) within a dispatch group. The instruction fetch mechanism interacts with the underlying first level instruction cache which further restricts dispatch group formation. Finally, dependent instructions are allowed to dispatch within the same group.

This puts additional pressure on the compiler. Dependent instructions need to be separated by independent instructions to force them into dispatch positions where execution latencies will be minimized. This is often difficult, however,

as filling in a two cycle bubble between dependent instructions could require up to fourteen instructions on the POWER5.

These dispatch restrictions, along with varying instruction latencies, create numerous non-obvious performance issues.

2.2 OProfile, ITrace, and Sim_ppc

This complexity makes it difficult to get an accurate picture of the performance of any code sequence. Visual examination is challenging for even small sequences as upstream code dependencies can delay data sources and impact timing for the target sequence in unexpected ways.

Micro benchmarks can measure the performance of a sequence for comparison to others. Benchmarks can tell which sequence executes in the fewest number of machine cycles, but not why. In this environment, a shorter (fewer instructions) "optimized" sequence may measure slower than the "non-optimized" sequence.

More advanced tools are required to identify these low-level hazards. Examples include cycle accurate timing simulators and profiling based on hardware performance counters.

Sim_ppc [3] is an example of a "cycle accurate timing model" for PowerPC™. A cycle accurate timer is a simulator that precisely models the micro-architecture and memory hierarchy for a specific processor design. As a software simulator, it can log details useful for visualization tools and accumulate statistics for detail reports.

Sim_ppc is a "trace driven timer" and does not execute the target code directly. Instead another tool like Performance Inspector's ITrace [6] (Instruction Trace) is used to generate trace data for input to sim_ppc.

Hardware event counters can be programmed to count specific events including hazards specific to the processor's micro-architecture. The counters can also be programmed to interrupt after a specific number of events. A profiling tool like OProfile [4] can use these interrupts to build histograms of the code associated with the events.

Each platform and processor design has unique micro-architectural features and event counters. OProfile supports hardware event counters; see the OProfile documentation for the complete list of supported processors and processor specific events.

Tools with similar function and capability are available on other hardware platforms as well. [9]

3 Drilling Deeper

We now illustrate how the advanced tools previously described can be used to identify non-obvious opportunities for code improvement.

We cover a scenario which illustrates the importance of hot branch target alignment on POWER5. We follow that with a discussion on branch prediction. And finally, we take an in depth look at several instances where performance was affected by "load-hit-store" hazards. Each of these cases is non-trivial to recognize by simply examining code, yet can have a dramatic impact on its performance.

3.1 Branch Target Alignment

On POWER5, all branch instructions terminate a dispatch group. The instructions at the branch target will then form the beginning of a new group. POWER5 always fetches an oct-word

(32-byte) aligned block of eight instructions. When the branch target is near the end of the block (and the branch is taken) the next dispatch group is limited to the instructions remaining in the fetched block. If there are not enough instructions remaining to fill an entire dispatch group, the group is prematurely terminated.

This means that the alignment of code following a taken branch matters and impacts down stream dispatch group formation. In general, instruction schedulers will schedule code assuming a new dispatch group starts at a block boundary or branch target. If the code is scheduled with this assumption, but the instructions fetched for the branch target don't include all instructions in the first dispatch group, the schedule for the entire block may be non-optimal.

This scenario was observed in glibc functions like memcmp when compiled with gcc-3.x. Memcmp shows up as a hot function in certain large benchmarks but inconsistent results were obtained when running with micro benchmarks.

Using Sim_ppc it was observed that memcmp was very sensitive to the starting alignment of the function. Gcc-3.x only enforced word alignment for functions so memcmp could start at any word within the block. This in turn impacted the alignments of internal branch targets and how instructions were assigned to dispatch groups.

For memcmp, the dispatch groups determined how many instructions where executed in parallel for the inner loop. This was especially true for rotate/shift/compare instructions required for the unaligned case. Where the alignment allowed more independent instructions to be dispatched and execute in parallel, memcmp ran 30% faster on POWER5.

For gcc-4.x the default function alignment was changed to quad-word (16-byte). Nops were also inserted to align hot branch targets to quad-word within the function (top of loops would be a prime example). This also impacts dispatch group formation for down stream code and can impact the performance of the outer loop by increasing its path length. However this creates smaller ranges with more predictable dispatch group formation which in turn simplifies the scheduling problem. See Figure 1.

The performance of memcmp using gcc-4.x gives results that are midway between the best and the worst case for gcc-3.x. However, these results are consistent and don't suddenly change after adding another "if" or "printf" to the test case. This change also increases the program/library size slightly which also increases the icache footprint, but the overall impact was positive as verified by SPEC results.

3.2 Branch Prediction

For most programs with if-then-else sequences, there is a limited opportunity to move instructions into the latency bubble between setting the condition code and the branch instruction. Most processors implement branch prediction circuitry to allow the processor to proceed past the branch before the condition code is ready.

This improves performance significantly as long as the prediction is correct. However, a misprediction requires that any instructions speculatively executed are aborted (a pipeline flush) and the instructions refetched from the correct path. This takes a minimum of twelve cycles on POWER5.

The POWER5 processor implements a branch prediction scheme that will scan an icache line of up to eight fetched instructions, predicting up to two of them per cycle (assuming the first

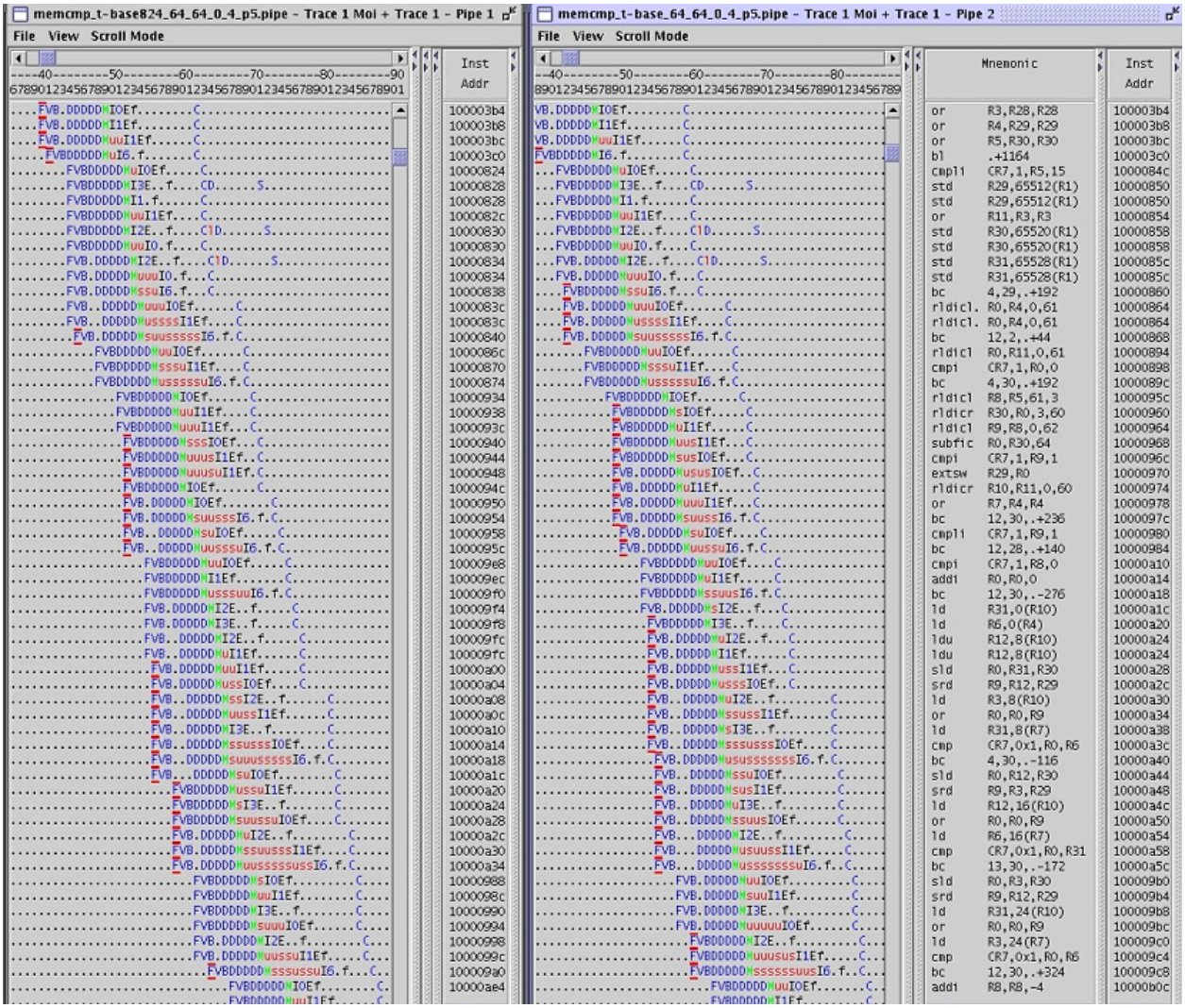


Figure 1: memcmp() compiled with gcc-3.x with different alignments. This scrollpipe view shows the same memcmp object code executing at different starting addresses side by side. On the left, memcmp() starts at offset 0x824 while on the right memcmp() starts at offset 0x84c. The version on the right runs 30% faster because the dispatch groups and fixed point issue queue slots line up to allow more instructions to execute in parallel. Note the instruction sequence on the left starting at offset 0x934, three rotate instructions are dispatched in a single group, the first rotate is sent to Fixed Point Pipe 0 ("I0") while the second and third rotates are sent to Fixed Point Pipe 1 ("I1"). This is unfortunate because the second and third instructions are independent and could execute in parallel if dispatched to different Fixed Point Pipes. Looking at the scrollpipe on the right we see that the same sequence starts at offset 0x95c (which is the end of an oct-word following a branch taken) so the first rotate is dispatched separately and the second and third are dispatched together with the following subtract immediate and compare immediate instructions. This sends the rotate pair to different Fixed Point Pipes ("I0", "I1") to execute in parallel, similarly with the independent subtract/compare pair that follows. There is a two cycle delay delivering the rotate result R30 as input to the subtract immediate. The POWER5 fills the pipe bubble (in "I1") with another (out-of-order) rotate from offset 0x974. These are small differences individually but over the length of memcmp they add up.

is predicted to fall through). Branch history tables are maintained and updated with the behavior of branches as they execute. These tables are then used on subsequent predictions of a branch.

On the first visit to a branch, there is no data in the branch history tables, likewise for branch intensive code, the finite size of the branch history tables may mean that the data for a branch has been flushed. This can result in the hardware incorrectly predicting the outcome of the branch and fetching instructions from the wrong path. When the branch ultimately executes, the resulting flush and refetch occurs.

Here again, sim_ppc can be used to recognize cases where branch misprediction is occurring. Techniques such as static branch prediction, where the compiler flags a branch as predominantly taken or not taken, can be employed to alleviate pressure from the branch history tables. POWER5 supplies the means for a compiler to statically predict a branch through the use of two bits on conditional branch instructions. When static branch prediction is employed, the branch does not occupy a position in the branch history tables. Compilers can use profile data to make educated decisions on which branches should be statically predicted, and which should be left for the hardware to predict. See Figure 2 for an example.

As hardware branch prediction becomes increasingly sophisticated, the need for static branch prediction decreases. Yet, there remain opportunities where a programmer's knowledge of the expected behavior of a piece of code can provide performance improvements through strategic use of static branch prediction.

3.3 Load-Hit-Store

On POWER5, instructions within the same dispatch group can execute out-of-order. The group of instructions is tracked by the core until all operations have finished at which point the entire group is completed in an in-order fashion.

For store instructions, cache updates must be non-speculative, and therefore cannot occur until the instruction reaches the completion stage.

As mentioned earlier, dependent instructions can be dispatched together on POWER5. This leads to an interesting problem when a store instruction is followed by a load from the same memory location within the same dispatch group. The store cannot update the cache until all other instructions are finished, but the load cannot finish until the store has updated the cache! This results in the entire dispatch group being flushed and re-dispatched with each instruction in a separate group.

A similar scenario can occur even if the load is in a separate dispatch group. If the load requires the data before the store has completed, the load is rejected causing a pipeline stall.

Figure 3 is a screen shot which illustrates a load reject caused by a load-hit-store scenario. An analyst inspecting this trace may notice the "j" indicating the reject, followed by the re-issue of the load. If this sequence happens to be in a hot section of code, then it may be worth investigating if compiler changes can be made to use other instructions to add more distance between the store and subsequent load. Alternatively, it may be possible to avoid the situation by avoiding the load altogether and simply reusing the value which is already in the register used on the store.

For the case where the store and load end up within the same dispatch group because there

Figure 2: Branch prediction missed. The branch in the cross hairs (Iop Id 1009680) was mispredicted as branch taken. This branch is dependent on the previous compare logical (cmpl) instruction which was dependent on the previous add immediate shifted (addis). All were dispatched in the same cyle, but due to data dependencies, the condition code was set 21 cycles later. this example also shows the impact of code alignment. Note that the load double (ld) (Iop Id 1009685) is dispatched by itself because it is the last word in the oct-word fetched following the branch.

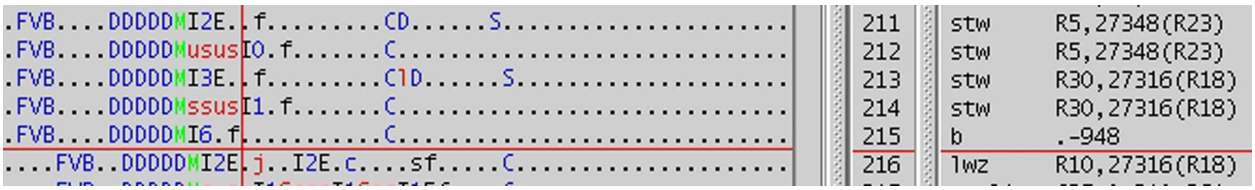


Figure 3: This illustrates a case where a Load-Hit-Store hazard is encountered between code that could be far apart in the code stream. The load word zero (lwz) at the cross hairs is the first instruction at the target of the unconditional branch. It loads from the same address stored to just prior to the branch. Since the store has not yet completed, the load is rejected. This illustrates an opportunity where the compiler might be able to eliminate the load and reuse the value in R30, or schedule additional instructions prior to the load.

are no independent instructions available to separate them, "nop" instructions can be inserted by the compiler to force the instructions into separate groups. This averts the problem of the entire dispatch group being flushed to reissue the instructions in separate groups.

You may think that the load-hit-store scenario would be rare, but it is quite common in the GLIBC [8] math library. For example the generic IEEE754 implementation of the finite function (`s_finite.c`):

```
typedef union
{
  double value;
  struct
  {
    u_int32_t msw;
    u_int32_t lsw;
  } parts;
} ieee_double_shape_type;

int
__finite (double x)
{
  int32_t hx;

  do/* GET_HIGH_WORD (hx, x) */
    {
      ieee_double_shape_type u;
      u.value = (x);
      (hx) = u.parts.msw;
    }
  while (0);
  return (int)
     (((hx & 0x7fffffff) -
             0x7ff00000) >> 31);
}
```

The finite function uses a union to transfer the double float value to an integer so it can use fixed point/logical operations to extract the exponent and check for infinity/NAN pattern. This generates the following powerpc64 assembler:

```
.__finite:
stfd 1,-16(1)
ld 3,-16(1)
rldicl 3,3,32,33
addis 3,3,0x8010
srwi 3,3,31
extsw 3,3
blr
```

The double parameter in FPR1 is stored (as a temporary) on the stack then immediately loaded into GPR3. This creates the classic load-hit-store scenario since the store float double/load double are the first two instructions of the function and normal (quadword) function alignment will guarantee that both are dispatched in the same cycle.

The POWER5 processor will try to dispatch the first four instructions of `__finite()` before it detects the load-hit-store condition. In this case the processor must flush all instructions, from store float double forward, then refetch the instruction stream starting from the store. This allows the processor to redispatch those same four instructions individually (each in a different cycle) and guarantees that the store enters its execution phase before the load. It also causes a delay of at least fourteen cycles on POWER5. See Figure 4 for an example.

The `_finite()` function is small with no real opportunity for scheduling. The only real option here is for the compiler to insert nops between the store and the load as described earlier. The gcc compiler inserts these nops by default when the -mtune=power5 option is specified. See Figure 5.

```
__finite:
stfd 1,-16(1)
nop
nop
nop
ld 0,-16(1)
rldicl 3,0,32,33
addis 3,3,0x8010
srwi 3,3,31
extsw 3,3
blr
```

The use of this union and coding style is pervasive in the GLIBC Math library. IBM has provided alternative (powerpc specific) implementations for some of the simpler math functions (`ceil`, `floor`, `rint`, `round`, `trunc`).

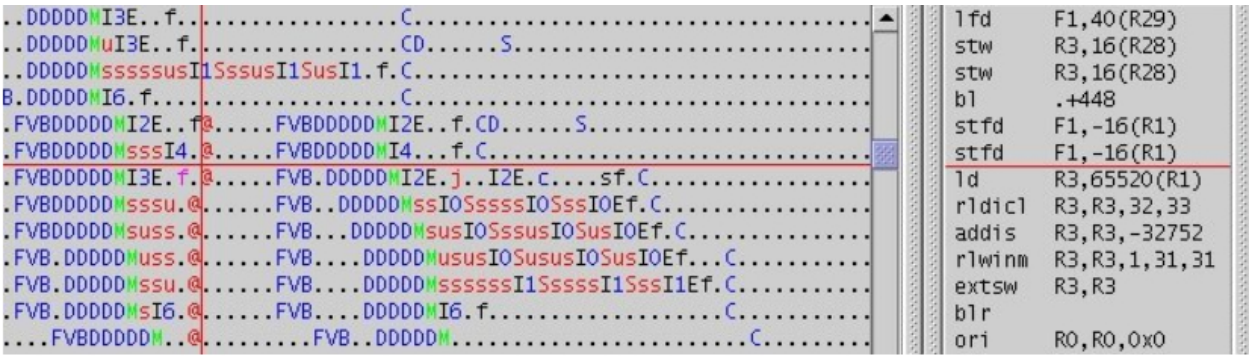


Figure 4: `__finite()` compiled by gcc-4.1 without -mtune=power5, shows Load-Hit-Store that requires a pipeline flush and refetch. The '@' at the cross hairs indicates the flush, followed by [re]fetch (FV), branch prediction (B), decode (D), and dispatch (M) cycles. Note that even after the flush/refetch the store does not complete in time, so the load incurs a load-reject (j) and retries the execute (I2E) before it can catch the store-forward (c) and finally the store-forward-finish (s). The completion of the store-forward allows the next instruction (rotate left double immediate then clear left (rldicl)) to enter its execution state successfully after two ISU-rejects (S) due to a source operand not being available.

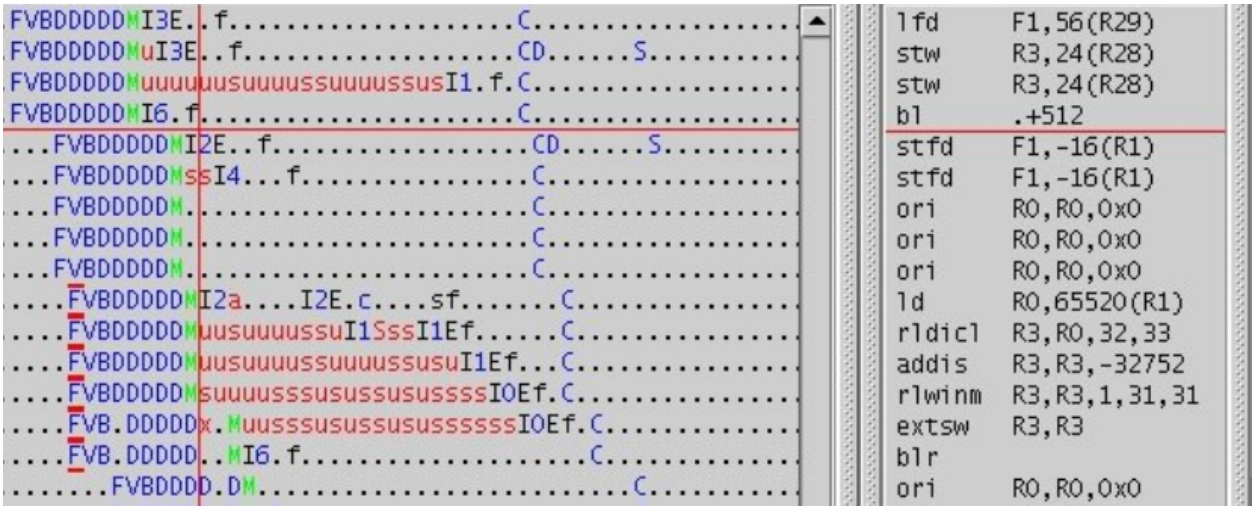


Figure 5: `__finite()` with -mtune=power5, shows that inserting nops (ori r0,r0,r0) between the store and the dependent load forces the load into a separate dispatch group. This eliminates the pipeline flush and reduces the cycles to completion from 27 to 13.

The more complex functions provide better opportunities for scheduling to separate the store from the load. Unfortunately the current version of gcc shows a preference to schedule loads as early a possible which defeats attempts to separate loads from stores.

There are other cases where program structure creates the load-hit-store scenario. One case was found in the Multiple Precision Arithmetic functions (`libc/math/mpa.c`). While investigating the poor performance of the `log()`, `exp()`, and `pow()` functions using OProfile, it was found that the internal functions `_inv()` and `__mul()` from `mpa.c` were the main contributors:

```
CPU: ppc64 POWER5, speed 1656.4
MHz (estimated) Counted CYCLES
events (Processor cycles) with a
unit mask of 0x00 (No unit mask)
count 10000
```

samples	%	symbol name
2398	52.1304	`__inv`
1314	28.5652	`__mul`
312	6.7826	`__dvd`
264	5.7391	`sub_magnitudes`
57	1.2391	`norm`
56	1.2174	`__add`
35	0.7609	`__dbl_mp`
35	0.7609	`__sub`
31	0.6739	`_wordcopy_fwd_aligned`
28	0.6087	`__mpexp`
24	0.5217	`memset`
16	0.3478	`__cpy`
11	0.2391	`__ieee754_pow`
5	0.1087	`__halfulp`
5	0.1087	`__mplog`
2	0.0435	`__exp1`
2	0.0435	`memcpy`
1	0.0217	`__ieee754_sqrt`
1	0.0217	`__slowpow`
1	0.0217	`_dl_init_paths`
1	0.0217	`_int_malloc`
1	0.0217	`strlen`

By inspection, we see that `__inv()` is calling `__cpy()`, `__mul()`, `__sub()` in the inner loop and calls `__mul()` twice each iteration.

```
for (i = 0; i < np1[p]; i++)
  {
    __cpy (y, &w, p);
    __mul (x, &w, y, p);
    __sub (&mptwo, y, &z, p);
    __mul (&w, &z, y, p);
  }
return;
```

Since `__mul()` shows up much higher in the histogram than either `_cpy()` or `__sub()`, we should look at that first.

At first glance there is nothing unusual about `__mul()`. Just a nested loop doing floating-point multiplies and adds.

```
k2 = (p<3) ? p+p : p+3;
z->d[k2]=zero.d;
for (k=k2; k>1; ) {
  if (k > p) {i1=k-p; i2=p+1; }
  else {i1=1; i2=k; }
  for (i=i1,j=i2-1; i<i2; i++,j--)
    z->d[k] += x->d[i]*y->d[j];

  u = (z->d[k] + cutter.d)-cutter.d;
  if (u > z->d[k]) u -= radix.d;
  z->d[k] -= u;
  z->d[--k] = u*radixi.d;
}
```

And in fact the code generated for the inner loop uses fused multiply and add instructions as appropriate.

```
#  for (i=i1,j=i2-1; i<i2; i++,j--)
#     z->d[k] += x->d[i]*y->d[j];
.p2align 4,,15
.L347:
lfd 13,0(11)
lfd 0,0(10)
addi 11,11,8
addi 10,10,-8
fmadd 12,13,0,12
stfd 12,0(8)
bdnz .L347
```

It is unfortunate that the store float double for `z->d[k]` is part of this inner loop as "k" is constant for this loop. In this case as x, y, and z are pointer parameters and the compiler does not know that they don't overlap in storage, this

store is required for correctness. At least gcc hoisted the initial load for `z->d[k]` out of the inner loop, which would have been an obvious load-hit-store scenario.

So we still don't have a clear picture of what the problem is (other than too much computation). Here we need to look for non-obvious hazards. The previous OProfile histogram was the basic timer/cycle count based sampling. To dig deeper we can use OProfile to sample based on hardware specific events that represent various execution hazards. When sampling for load-hit-store events we see that `__mul()` shows up with higher frequency than before (default cycle-based sampling).

```
CPU: ppc64 POWER5, speed 1656.4
MHz Counted     PM_LSU0_REJECT_SRQ_
LHS_G13 events (LSU0 reject due to
load hit store) with count 1000
```

samples	%	symbol name
4170	50.9905	`__inv`
3467	42.3942	`__mul`
429	5.2458	`__dvd`
53	0.6481	`sub_magnitudes`
16	0.1956	`__add`
12	0.1467	`norm`
11	0.1345	`__dbl_mp`
8	0.0978	`__ieee754_pow`
4	0.0489	`__sub`
2	0.0245	`__cpy`
2	0.0245	`__exp1`
1	0.0122	`__halfulp`
1	0.0122	`__mpexp`
1	0.0122	`__mplog`
1	0.0122	`pow`

It was found that the final store of the outer loop `z->d[--k] = u*radixi.d;` was hit by the initial load of `z->d[k]` in the next iteration. This would not normally be a problem as there are nine instructions and three branches between the store at the bottom of the outer loop and the corresponding load near the top. However the correct but not strictly necessary stores to `z->d[k]` and dependent calculations delayed the completion of the final store sufficiently to interfere with the next load. See Figure 6.

In this case we know that the array z does not overlap with arrays x and y. So a simple recoding of `__mul()` eliminates the redundant stores and the load-hit-store.

```
k2 = (p<3) ? p+p : p+3;
z->d[k2]=zero.d;
z_k = z->d[k2];
for (k=k2; k>1; ) {
  if (k > p) {i1=k-p; i2=p+1; }
  else {i1=1; i2=k; }
  for (i=i1,j=i2-1; i<i2; i++,j--)
    z_k += x->d[i]*y->d[j];

  u = (z_k + cutter.d)-cutter.d;
  if (u > z_k) u -= radix.d;
  z->d[k] -= u;
  z_k = u*radixi.d;
  --k;
}
z->d[k] = z_k;
```

See Figure 7 for a view of the updated scrollpipe.

4 Additional Benefits

The focus of this paper has been on methods to improve code generation sequences; the tools presented here are equally useful when hunting down performance regressions between two versions of the same compiler. A side by side analysis of scroll pipe data from sim_ppc can show areas where one code stream is behaving differently than a different code stream. Subtle variances in code generation or scheduling can expose unexpected hazards that can be observed using this method.

The examples presented here deal primarily with hazards occurring within the processor's CPU. The same techniques, however, can be used to identify problems triggered by components such as the memory subsystem.

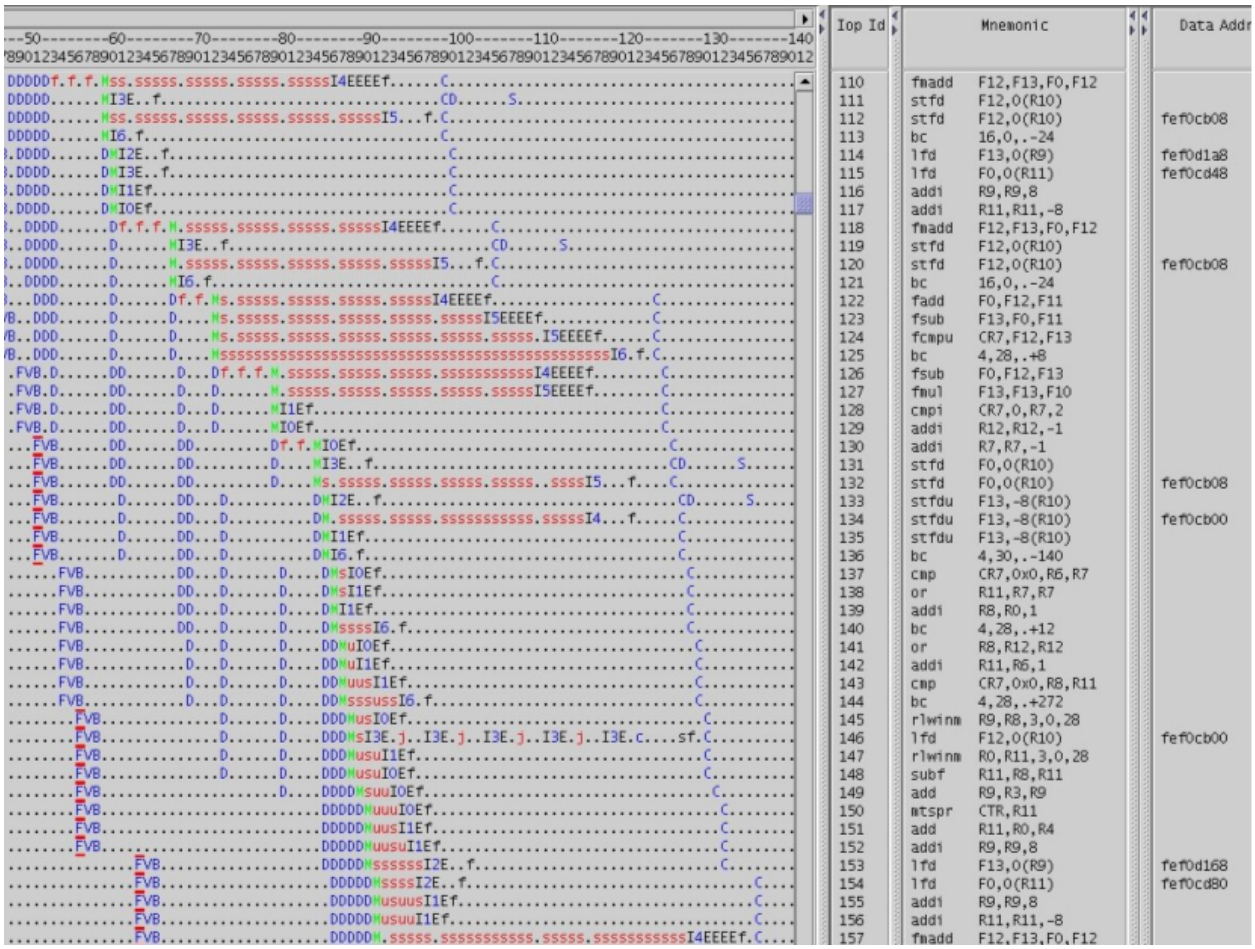


Figure 6: `__mul()` called from ieee754/slowpow. Current implementation compiled with gcc-4.1. Notice that the inner loop (Iop Ids 114-121) is dispatched in two groups but completion requires six cycles per iteration. This is paced by the floating-point multiply and add (fmadd) who's result is required input on the next iteration, plus one cycle for the dependent (unnecessary) store floating-point double (stfd). Note that we have filled the FPU (12 entry) issue queue, and dispatch has to be delayed until a previous FPU instruction completes (the repeated "f's" preceding the "M" dispatch on Iop Ids 110, 118, 122, ...). Also note the "Load Refects" ("j's") for the load floating-point double (lfd) on IopID 146 which hit the store floating-point double with update (stfdu) at Iop Id 134. The multiple load rejects are caused by the dependent computations which delay the stfdu by 30 cycles.

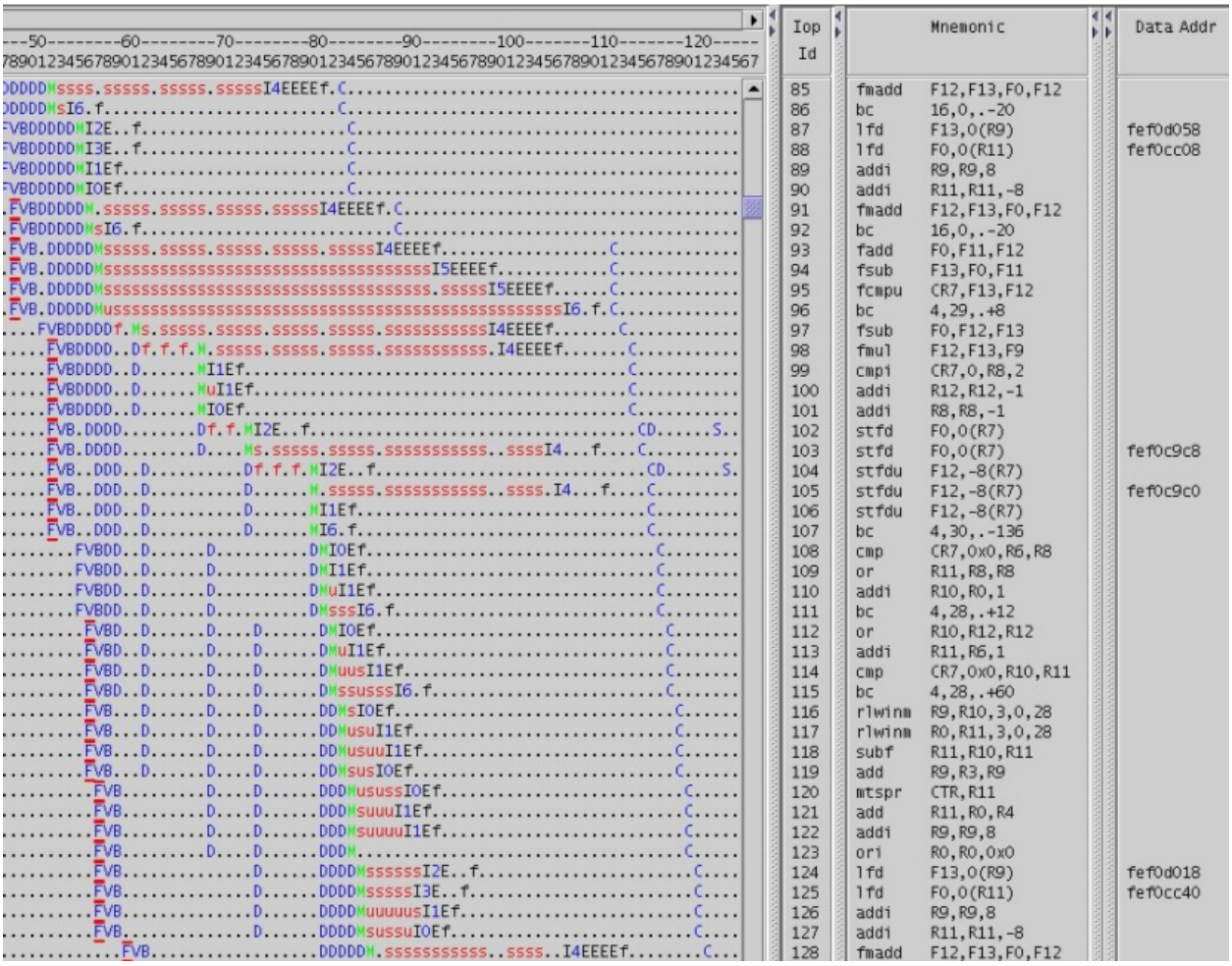


Figure 7: `__mul()` called from ieee754/slowpow. With proposed changes, compiled with gcc-4.1. Note that the inner loop is still paced by the floating-point multiply and add (fmadd), but we don't see any dispatch delays until later in the sequence because eliminating the (unnecessary) store floating-point double (stfd) from the inner loop frees slots in the FPU issue queue. We have also eliminated the load floating-point double (lfd) (`z->d[k]`) from the outer loop and eliminated the load-hit-store as well.

In addition, sim_ppc, or equivalent simulation tools can be invaluable to compiler developers during early hardware development. As compilers are tuned to generate code for new processors, these tools can be used to measure their effects well before actual hardware is available for testing.

5 Conclusions

The design of modern processors is creating an increasingly complex environment for compilers; generating optimal code while avoiding subtle hazards is becoming more and more difficult. Compilers have to model instruction latencies and dispatch group formation while considering all the special cases involved. It is very difficult to tell from just looking at the code how it will perform.

In addition, representing the large variation of rules can become problematic to a compiler. The instruction scheduler in gcc is based on a deterministic finite automaton (DFA). The increasing complexity of machine descriptions can result in prohibitively large DFAs. Building them can become impossible due to the memory requirements posed by the massive number of states that must be represented. While creative factoring of the DFA can be used in most cases, these limitations can force a compiler to pick and choose between those rules that are deemed most important to represent.

This paper has covered several tools and strategies for identifying the most critical of these rules. Methods to identify and correct hazard prone code streams were presented. Several detailed examples were shown to illustrate how their use can be effective in identifying opportunities for improving the code generated by a compiler and in turn the performance of the application and the machine in general.

6 Acknowledgements

The authors would like to thank Ryan Arnold for his investigation into, and assistance with, several of the examples used in this paper. We would also like to thank Maynard Johnson for support and assistance with ITrace and OProfile.

References

[1] Information on SPEC benchmarks can be found at `http://www.spec.org`

[2] Information on gprof can be found at `http://www.gnu.org/software/binutils`

[3] Information on sim_ppc can be found at `http://www.alphaworks.ibm.com/tech/simppc`

[4] Information on oprofile can be found at `http://oprofile.sourceforge.net/news`

[5] Information on gzip can be found at `http://www.gzip.org`

[6] Information on itrace can be found at `http://perfinsp.sourceforge.net/itrace_ppc.html`

[7] Information on gcc can be found at `http://www.gcc.org`

[8] Information on glibc can be found at `http://www.gnu.org/software/libc`

[9] Information on VTUNE™ can be found at `http://www.intel.com/cd/software/products/asmo-na/eng/vtune/vlin/240665.html`

Switch Statement Case Reordering FDO

Edmar Wienskoski
Freescale
edmar@freescale.com

Abstract

When gcc parses a switch statement, it uses some criteria to decide between generating a jump table or a binary search tree. The jump table has a fixed significant cost, and for large switch statements the sequence of compare-branches from the root to the leaves can also be costly.

The optimization described here collects a histogram of a switch statement condition expression and uses it to balance the binary search tree. We also implement a default statement promotion: Single values in the histogram that maps to default in the switch statement, are candidates to become a new node in the binary search tree, which gives a chance to that particular value to have its path on the tree optimized.

1 Motivation

In looking at the perl benchmark in Spec2k for optimization opportunities, it was noticed that the basic blocks more often executed where part of one big switch statement inside the main interpreter loop.

From the switch condition expression evaluation to those basic blocks, there were many instruction cycles. A feedback directed optimization that moves the most often executed case statements out of the switch statement could improve the performance of this benchmark considerably.

To test the effectiveness of pursuing this optimization: I changed the original source code of function `regmatch` on file `regexec.c` such that the two case statements more often executed were hoisted outside the switch statement. Figure 1 illustrates the idea, where em x and *y* represents the two case statements in question. The result was a surprising 17% improvement.[1]

```
                          if (x){
  switch (c){               :
    case a:               } else
      :                   if (y){
    case x:                 :
      :             =>    } else
    case y:               switch (c){
      :                     case a:
    case b:                   :
      :                     case b:
                              :
```

Figure 1: Hoisting case statements out of the switch

Gcc infrastructure has developed quite fast re-

[1]This was obtained with the Motorola research compiler and a G3 Linux PowerPC machine. The same experiment with gcc development branch *tree-profiling-branch* on a G5 Linux PowerPC machine yields 12.6% improvement.

cently: SSA based optimizations, auto vectorization, and feedback directed optimizations among them. All that activity motivated us to apply the above idea into gcc.

2 The implementation

Currently, gcc has two strategies to generate code for a switch statement: a jump table, or a binary search tree.

The jump table offers uniform access time for all case statements, but that time is usually much slower than a single compare-and-branch sequence. On the other hand, the binary search tree requires several compare-and-branch sequences to reach the leaves of the tree. As the number of case statements increases, the length of the compare-branches sequences from the root of the binary search tree to their leaves also increases.

Considering that in a binary search tree 50% of all nodes are leaves,[2] the chance of one of the most executed case statements ending up either in a leaf or in a node next to a leaf is quite high. That of course, would be the worst possible performance outcome. Thus, whenever the number of case statement is above some threshold, gcc will give preference to generate the jump table instead of the binary search tree in order to avoid risking that performance penalty.

But in fact, the gcc infrastructure supports a weight[3] attribute to each node of binary search tree. That effectively allows of the implementation of balanced binary search tree algorithm. Unfortunately, this feature is not used effectively to avoid the performance penalty described in the previous paragraph. Currently, gcc attributes weights of 1 to all nodes that represents one single value of the switch statement condition expression, and weight of 2 to nodes that represents a range of values.[4]

Considering the infrastructure available in gcc to balance a binary search tree, it is a better idea to tweak the weight of the binary search tree nodes, instead of implementing all the possible basic block manipulations and condition expression generations to achieve the code transformation depicted in Figure 1.[5] The result of this strategy could be as effective as the proposed solution, as Figure 2 illustrates.

Another consideration in the implementation of this optimization is how gcc collects feedback information.

Gcc can instrument basic blocks to measure basic block frequency, but can also instrument the target application to compute a histogram of run time values of any expression in the application. That includes the computation of histograms of switch statement condition expressions.

The basic block counters could had be used to weigh the binary search tree, but the histogram gives us one extra optimization opportunity: To promote values that map to the default statement into individual nodes in the binary search tree.

In general, all run time values of the condition expression that maps to the default statement, causes the execution of a sequence of compare-branch instructions that goes from the root of

[2] Except for some odd shaped trees, the observation is quite accurate for balanced or not trees as long as each internal node has 2 siblings.

[3] Gcc sources name this attribute as `cost`. We will now refer to weight or cost interchangeably.

[4] Incidentally, the switch statement in function `regmatch` on file `regexec.c` of Spec2k perl benchmark consists of single values only.

[5] It would also be much easier to debug and prove correctness of the optimization.

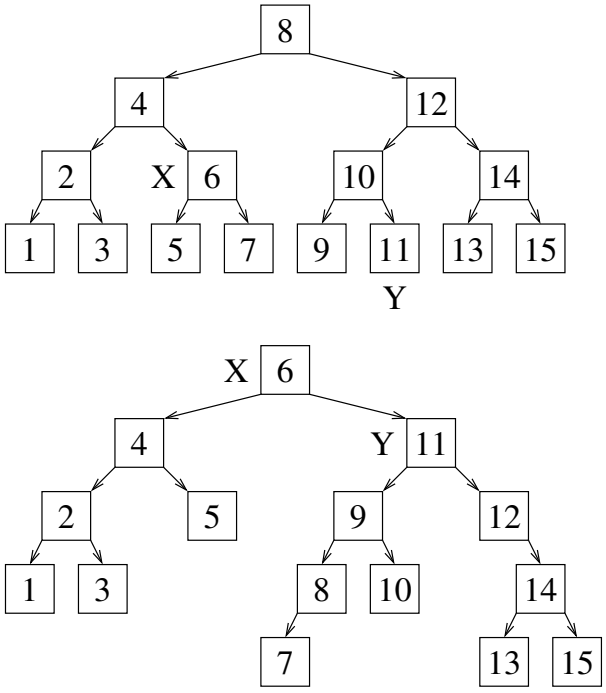


Figure 2: A proper cost function can cause the tree balancing algorithm to move the nodes *X* and *Y* closer to the root

the binary search tree to one of its leaves, before it can be identified as one default value. That means the execution of the default statement will always incur in the highest performance penalty. But what if the default statement is among the ones with highest execution frequency?

Default value promotion is the answer. The idea is to identify one particular value in the histogram that maps to the default statement, and whose execution frequency is high, (This can be easily done by inspection of the histogram and some heuristic to define "high execution frequency"), and then create a new node in the binary search tree for that value. This is semantically equivalent to the code transformation illustrated in Figure 3.

After balancing the tree, the path between the root and the newly created node can be optimized, leaving the general performance penalty of the default statement to the less frequent occurring values.

```
switch (c){                    switch (c){
  case a:                        case a:
    :                              :
  default:          =>           case x:
    :                            default:
}                                  :
                               }
```

Figure 3: The value *x* on the histogram of condition expression *c* satisfies the "high execution frequency" criteria.

Examples of heuristics that could be used are: A fixed threshold (e.g.: 10% of execution frequency); and a relative parameter (e.g.: Highest five values in the histogram).

One last implementation detail concerns the case when the case statement labels are too sparse, and some case labels are below the histogram range and / or some are above it. If that happens, the execution frequency of values below and / or above the histogram range are equally distributed among the corresponding labels.

3 Results

The optimization described in this paper was implemented and tested on a snapshot of the *tree-profile-branch* development branch from May 24, 2005 [TPB].

All of Spec2k was validated on a G5 running Linux PowerPC [YDL, YHPC]. Except for perl, all other benchmarks had no variation in performance and are omitted from this discussion.

	Individual parts							Σ	%
Original gcc	21.71	1.23	18.76	63.60	36.16	32.22	59.93	233.61	—
Case stmt hoisted	21.43	1.24	18.72	53.80	31.82	27.71	52.63	207.35	12.6
Balanced - train	18.99	1.19	17.94	56.20	32.77	28.71	54.32	210.12	11.2
Balanced - validation	19.29	1.23	17.95	51.51	30.42	26.44	50.30	197.04	18.5
Balanced & hoisted	18.74	1.20	17.84	52.21	31.03	26.96	51.57	198.35	17.7

Table 1: Spec2k perl benchmark execution times. Individual parts and total time shown in seconds.

On table 1 we show all Spec2k perl results. The first line of data, was obtained with the original compiler and is used as base reference for relative comparison. The same level of optimization was used across all executions, including other profile feedback optimizations. The second line, was obtained with the benchmark manually optimized as depicted in Figure 1. The third and fourth lines, were obtained with the switch statement case reordering enabled. On the third, the benchmark was trained with the *train* data set. On the next one the *validation* data set was used instead. Finally, the last line has both manually hoisted case statements and the switch statement case reordering.

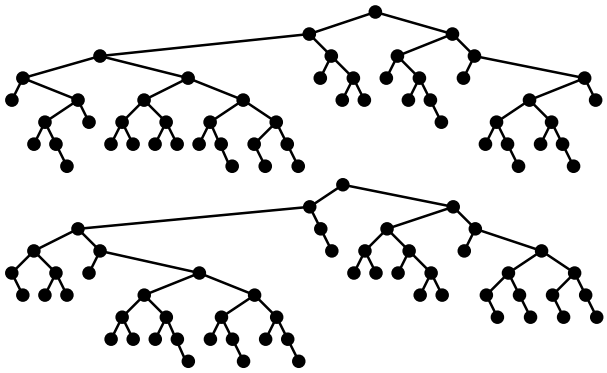


Figure 5: Binary search tree of the switch statement condition expression of function `regmove` on file `regexec.c` obtained with the train data set and validation data set respectively

The reader should note that, in order to manually apply the case statement hoisting as described in Section 1 of this paper, one must have the knowledge of which two case statements were more often executed in the validation execution of the benchmark. But that knowledge is not available when the Spec test harness is used. Because the data set used to train the application may not represent adequately the data set used for validation, a direct comparison of the second and third lines would not be fair. That is the reason for having the fourth line on Table 1. By training the application with the same data set used for validation, we ensure that both the switch statement case reordering fdo and the manually applied case hoisting have the same knowledge base. We don't claim any official Spec results, per Spec rules.

For illustration, Figure 4 shows the histogram of the switch statement condition expression of function `regmove` on file `regexec.c`. The left columns are the values of the histogram obtained using the train data set, the right columns are the values of the histogram obtained using the validation data set. Figure 5 shows the actual binary search tree after balancing it. The one in the top was obtained with the training data set, the one on the bottom was obtained with the validation data set.

On Section 2 it was argued that rebalancing the binary search tree would be enough to obtain the same results as hoisting case statements out of the switch. The fifth line on Table 1 confirms that: in the presence of switch statement

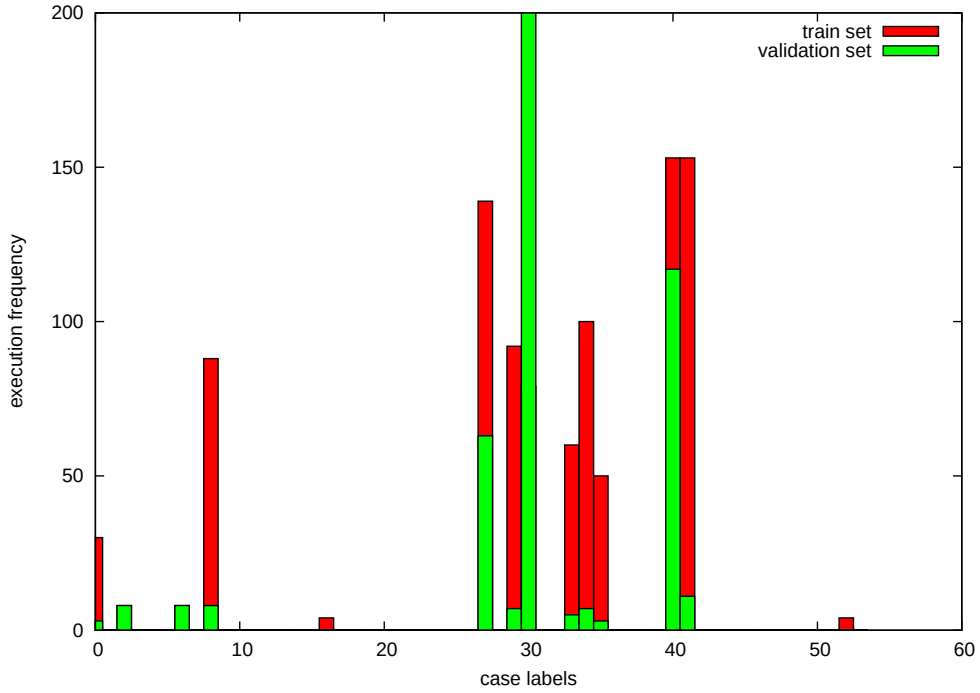


Figure 4: Histograms of the switch statement condition expression of function `regmove` on file `regexec.c` obtained with the train data set (left columns) and validation data set (right columns).

case reordering, hoisting the two most often executed case statements is ineffective.

	Time	%
train data set	202.42	—
validation data set	204.58	-1.0
Case stmt hoisted	193.52	4.6

Table 2: Spec2k perl benchmark execution time in seconds, using xlc 7.0 compiler.

To contrast, on Table 2 we have a set of Spec2k perl results for the IBM compiler [XLC] running in the same machine.[6]

On table 3 we show all Spec95 m88ksim results. Those results were obtained in the same G5 Linux PowerPC machine. The first line of data was obtained with the original compiler. The second and third lines, were obtained with the switch statement case reordering enabled and training with the train and validation data set respectively.

This benchmark has a total of 66 switch statements, 56 of them were never executed. Among the other 10, the two largest switch statements have 38 and 16 case labels respectively. Figure 6 shows the binary search tree for those two switch statements after balancing them. Note the heavy effect of the balancing in the symmetry of the trees.

As discussed in Section 2, gcc makes a tradeoff between generating a jump table and a binary search tree. As the memory latency of the target architecture decreases, one can expect the importance of this tradeoff to be less significant or non-existing altogether. This trend is visible on Table 4, it summarizes all perl and m88ksim results for two other PowerPC parts that has re-

[6]The benchmark was compiled with flags -O5, and -qpdf1 / -qpdf2.

	Time	%
Original gcc	30.25	—
Balanced - train	27.25	11.0
Balanced - validation	26.83	12.7

Table 3: Spec95 m88ksim benchmark execution time in seconds.

	7450				8548			
	perl		m88ksim		perl		m88ksim	
	Time	%	Time	%	Time	%	Time	%
Original gcc	454.8	—	46.0	—	501.0	—	60.0	—
Balanced - train	443.2	2.6	46.0	0.0	505.3	-0.8	59.8	0.3
Balanced - validation	428.7	6.1	46.0	0.0	469.7	6.8	59.9	0.2

Table 4: Summary of results for two other PowerPC parts: the 7450 and the 8548.

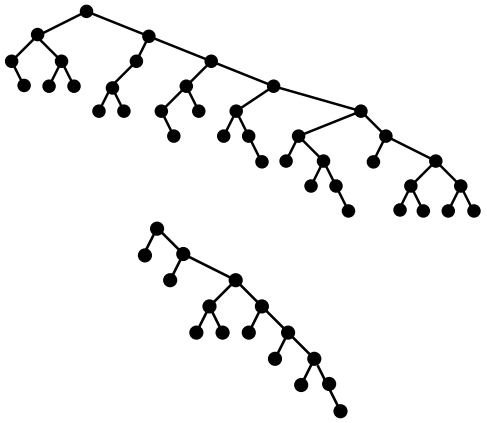


Figure 6: Binary search trees of two large switch statement on Spec95 m88ksim benchmark.

spectively smaller memory latencies: the 7450, and the 8548.

4 Conclusion and Future work

We showed how feedback directed optimization can be used to improve large switch statements performance, for both explicit case statements and default statements.

Futhermore, that this optimization can be implemented in gcc by leveraging existing infrastructure, namely the capacity to generate balanced binary search trees for switch statements, and the capacity to create run time value histograms of the switch statement condition expressions of the target application.

In conclusion, for such small code change in gcc, the gain in performance of certain applications is quite significant.

Minor improvements to the current implementation are planned:

- At the present, all the histograms have the same fixed range. We could use this range as a minimum range, and have the compiler to scan the case labels and enlarge the range if necessary. We could also provide command line parameters to override the minimum range values, and establish maximum range values.
- The default value promotion heuristic used in this implementation was a fixed 10% threshold. There should be a set of param-

eters to change this to some other value, or to some other heuristic.

Another idea, also related to switch statement optimization, is to explore the application of feedback directed information on the register allocator. In the perl benchmark, there is one long case statement that causes a large amount of register pressure, but is seldom executed. That could cause spill to be inserted elsewhere, perhaps into the main path of execution. The switch statement feedback information could be used to guide the register allocator to spill registers inside the seldom executed path instead.

5 Acknowledgments

I would like to thank Kate Stewart for her unlimited support during all the stages of this project. Without her, this work would still be forgotten in some cabinet.

I also want to thank Kristi Morton for her helpful comments and encouragement, Jan Hubicka for his candid feedback on the first gcc patch, and Freescale for making this work possible.

References

[TPB] FSF, `http://gcc.gnu.org/projects/tree-profiling.html`

[YDL] Terra Soft Solutions, *Yellow Dog Linux Version 4.0 for PowerPC*, 2005.

[YHPC] Terra Soft Solutions, *Y-HPC User's Manual*, January 13, 2005.

[XLC] Absoft, *XL C/C++ IBM Compiler for Y-HPC Linux 7.0*, Reference XLC4CSS70, 2005.

Changes to RTL Dataflow Analysis

Danny Berlin
Google
dannyb@google.com

Kenneth Zadeck
Natural Bridge, Inc.
zadeck@naturalbridge.com

Abstract

Significant revisions have been made to DF, the alternative RTL dataflow analysis module for the dataflow branch, to make it suitable for everyday use as a provider of both backend liveness information, as well as various other dataflow facts (use-def and def-use chains, death notes and register information). These changes include:

- Enhancement of the RTL scanning so that it now encapsulates all of the hard register special cases that were scattered throughout the backend.
- Revision of the interfaces to better support the solution of abstract dataflow problems.
- Replacement of the use-def and def-use chain algorithms that speed up their computation by up to three orders of magnitude.
- Removal of the non working incremental dataflow interface.

Additionally several phases of the compiler, such as the global register allocator, DCE, and DSE have been modified to use the new DF rather than FLOW. This change has provided for better generated code as well as faster compilation.

All of these will be discussed in the full paper.

1 Motivation

The back end of GCC uses `flow.c` to perform the dataflow analysis.

- **The flow analysis engine is archaic.** The generally accepted method for dataflow analysis is to scan a basic block once building a summary of the instructions that occur within that block. Global analysis then uses that summary as its input. `Flow.c` rescans each block at each step of the iteration. This is quite expensive.
- **The iteration technology is primitive.** Significant improvements to iterative dataflow were first developed by Hecht [5] in 1975. The technology in flow is inferior to this.
- **Unrelated problems have been added to the iteration.** Many of the problems, like finding auto inc instructions derive no benefit from being done inside the main iterative loop aside from being able to reuse some intermediate structure.
- **The current algorithm may not terminate.** This has been *solved* by only allowing a certain number of iterations. However, when this limit is reached, flow is currently left with incorrect answers.
- **Flow does not get the *best* solution when used incrementally.** There are many

possible correct solutions to the dataflow equations. However, only one solution is *minimal*. This is the desired solution.

The back end of GCC uses FLOW to perform the dataflow analysis.

2 Underlying Technology

Dataflow analysis is defined over a graph of basic blocks, the *control flow graph*, (cfg). Dataflow problems can be characterized in several ways:

direction Dataflow problems are either *forward*, information flows in the direction of the cfg edges, *backward* information flows against the edges in the cfg or *bidirectional* information can flow in both directions. The *logical predecessors* of a block basic block *b* are the cfg predecessors if the problem is forwards or bidirectional and the cfg successors if the problem is backwards. The *logical successors* are defined in a corresponding way.

In each basic block, two sets are defined, an *in* and and *out* set. For forwards and bidirectional problems the *in* set is at the top of the block and the *out* set is at the bottom of the block. This is reversed for backwards problems.

domain The set of items to be analyzed: commonly used domains are the set of registers, the set uses, or the set of definitions.

confluence The operation to be executed at the merge point in the cfg. Confluence operators are either simple or complex. The simple operations are generally *set union* where the *or* operator is used for bit vectors, or *set intersection* where the operator used at merge points is set intersection.

However complex operations may also be used. In constant propagation, the confluence operator is equality over values. While DF is capable of solving dataflow problems with complex merge operations, the technology used in DF is generally not the appropriate for this kind of problem.

A dataflow problem is a set of simultaneous equations.

> The *in set* of each basic block *b* is defined to be the confluence operator applied to the *out set* of each logical predecessor of *b*.
>
> The *out set* of each basic block *b* is defined to be the transfer function applied to the *in set* of *b*.

There are many possible correct solutions to these equations. The goal of the dataflow is to find the *best* solution to the above system of simultaneous equations. For a set union (intersection) problem we want the *smallest solution (largest solution)*, i.e. the solution with the fewest (most) bits that is *correct*.

2.1 Solving Dataflow Equations

The dataflow equations can be solved in a variety of ways. Most of the techniques fall into two categories:

elimination algorithms The earliest elimination techniques tried was Gaussian elimination. However it was realized very quickly that the structure of the control flow graph could be taken advantage of yield faster solutions. There were many techniques developed [1, 4, 7].

Elimination algorithms are built on the idea of divide and conquer. It is easy to compute the solution to the data flow equations for control flow graphs of certain forms. The idea is to parse the control flow graph into a recursive tree that contains only these forms. Then the dataflow solution can be obtained by walking the tree in a particular order.

While elimination algorithms are generally fast, they do not work if the control flow graph cannot be parsed into these simple forms. In particular, any graph that contains a multiple entry loop must use other techniques.

iterative algorithms The earliest iterative algorithms were simple worklist iteration, described below and implemented in FLOW. This has the advantage over the elimination techniques that it works for all control flow graphs. The disadvantage is that it is slow.

Matthew Hecht [5] observed that if you impose a certain structure onto the worklist, the iteration will tend to converge very quickly. Forward problems process the blocks in reverse postorder and backwards problems process the blocks in postorder. Bidirectional problems can be solved by alternating passes of postorder and reverse postorder. For programs that have only single entry loops, the number of passes is never greater than one plus the maximum loop nesting.

Atkinson and Griswold [2] made a small modification where an additional depth first search is added in certain conditions. They showed that this modification improved Hechts algorithm for many common cases. This is the technique implemented in DF.

There are three steps to solving a dataflow problem:

- The first step in solving a dataflow problem is typically building the *transfer functions* for the basic blocks. The transfer function for block *b* describes how the instruction within *b* can be used to compute the *out set* of *b* from the *in set* of *b*.

  While it is certainly correct rescan *b* each time we wish to compute *b*'s *out set*, this is too expensive since the *out set* may have to be computed many times before the equations converge.

  For most simple dataflow problems it is possible to summarize the action of a basic block into something that is much simpler than rescanning the block. This is almost always true for problem that represent their results as bit vectors. For these problems, the summary for a block is typically represented as two bit vectors, *kill* and *gen* where the kill bitvector knocks bits out of the vector and the gen bitvector adds other bits back.

- The second step is to initialize the *in* and *out* sets for each block. For set union problems the sets can be initialized to the empty set and for set intersection problems the sets can be initialized to the universal set. However it has been observed that convergence is faster if the *out* set is initialized to the *gen* for set union problems or *kill* for set intersection problems.

- The third step is to actually solve the equations. There are a wide variety of techniques that have been proposed over the years. Almost all compilers use some form of fixed point iteration using a worklist.

  The technique implemented in FLOW is based on the simple worklist iteration

```
worklist <- all blocks
until (worklist is empty) {
  take b off the worklist
  b->in = empty set
  foreach logical preds p of b
    b->in |= p->out
  temp = trans_function(b, b->in)
  if (temp != b->out) {
    b->out = temp
    foreach logical succ p of b
      add p to worklist
  }
}
```

above. However, no transfer functions are ever computed. Instead the instructions in a block are rescanned each time the block is processed by the iteration.

3 Underlying Technology

3.1 Predefined Dataflow Problems

Unlike the analysis in FLOW which only solves live variables, DF is capable of solving any forwards or backwards dataflow problem[1] Several of these problems have been defined in `df-problems.c` and are usable in any pass in the backend that maintains a control flow graph. There are nine predefined dataflow problems that are packaged in a way that is easy to use:

Scan (SCAN) DF works on an abstraction of the RTL. The scanning phase builds that abstraction. There are several options control the scanning. The most common option controls if hard registers are to be considered in addition to pseudo registers.

For each instruction, the sets of registers defined and of registers used are created as well as the the set of multiword references. For each basic block *b* there is also the set of artificial uses and defs that occur at the bottom and top of *b*. Artificial registers are implicit uses or definitions of registers that cannot be attached to explicit instructions. For instance, before instruction selection, the stack and frame pointers are considered live everywhere. Exception handling also give rise to artificial uses and definitions.

Technically scanning is not a dataflow problem in that there are not equations to solve. What this problem does is compute datastructure that make the computation of the transfer function for the other problems efficient.

Live Registers (LR) The live registers problem answers the question "what registers still contain values that are used later in the program?"

At the end of computation, there is a bitvector at the bottom of each basic block, *b*, that contains the set of registers whose value may be used on some path reachable from *b* to the exit of the program. Live variables is a backwards, set union, problem where each slot in the bit vector represents one register. *Gen* is the set of registers that are used in the block. *Kill* is the set of assignments to whole registers. The bitvector at the top of *b* is the union of the bitvectors at the bottom of the preds of *b*.

Uninitialized Registers (UR) The uninitialized registers problem answers the question "what registers are used before they are defined?"

At the end of computation, there is a bitvector at the top of each basic block, *b*, that contains the set of registers which

[1] With a small amount of work, bidirectional dataflow problems could be accommodated.

may provide values on some path from the beginning of the function to *b*. Uninitialized registers is a forwards, set union, problem where each slot in the bit vector represents one register. *Gen* is the set of registers that are set in the block. *Kill* is the set of clobbers to whole registers. The bitvector at the bottom of *b* is simple the union of the bitvectors at the bottom of the preds of *b*.

UR with Early Clobber (UREC) This problem is a specialization of the uninitialized registers problem that takes into account "early clobber" instructions. This processing is over conservative and should be incorporated directly into the interference graph building. When this happens, this problem will go away.

Reaching Uses (RU) The reaching-uses problem answers the uses analogue of the Reaching definitions problem, "which uses of a register may reach this definition site?" If a use site reaches some point in the program, it means not every path in the program redefines that register.

This is a very expensive problem to compute because there are a large number of uses in a large function and each use requires one slot in all of the bitvectors. Thus, the use of this problem should be discouraged. Currently it is only used in modulo scheduling. Getting rid of the use of this problem would most likely speed up that phase[2].

Reaching Definitions (RD) The reaching definitions problem answers the question "which definitions of a register may reach this point in the program?" If a definition site reaches some point in the program, it means not every path to that point in the program kills the definition.

Chain Building (CHAIN) Def-Use and Use-Def chains provide explicit chains formed from either the reaching uses, or reaching definitions problems. In particular, given a use of a register, the use-def chains will provide links to all definitions of that register that may reach that use. Given a def of a register, the def-use chains will provide links to all uses of the register the definition reaches.

Building chains is also not technically a dataflow problem because the construction of either type of chain is based on the solution of the reaching definitions problem.

Register Information (RI) Register information is a collection of information about registers used by passes such as the register allocator. This includes the number of references made to the register, how many calls the register lives across, and other miscellaneous information used in register allocation heuristics. This is the same information about how it builds this information so the information that was computed in FLOW with the `PROP_REG_INFO` parameter however the information produced here is more precise that that computed in FLOW.

`REG_DEAD` **and** `REG_UNUSED` **Notes** `REG_DEAD` and `REG_UNUSED` notes are simple information that was previously provided by FLOW. `REG_DEAD` notes represent kills of registers. Anytime a register dies in an instruction, a `REG_DEAD` note is generated. `REG_UNUSED` notes represent the last use of a register. If no further uses of the register occur in the program, a `REG_UNUSED` note is generated. This is

[2] In the original version of DF, use-def chains were built using this problem. Since modulo scheduling used use-def chains, this problem was available for free. The current implementation of DF builds both use-def and def-use chains from reaching definitions so using this problem is expensive.

the same information about how it builds this information so the information that was computed in FLOW with the `PROP_DEATH_NOTES` parameter however the information produced here is more precise that that computed in FLOW.[3]

3.2 Other Features of FLOW

FLOW has also become a catch basin for a wide variety of transformation that have nothing to do with dataflow analysis except that they utilize some datastructure that was private to FLOW. These include:

- Cleanup of conditional assignment statements with a basic block.
- Combining memory references and increment/decrement instructions into pre and post increment instructions.
- Discovery of functions that change the stack pointer.
- Computation of register use statistics.

There is little synergy between these problems and the rest of dataflow analysis. Thus, we have decided to either make these separate passes or make it a separate dataflow problem.

3.3 Dead Code Elimination

Dead code elimination (DCE) and dead store elimination (DSE) are handled in `dce.c`. There are two dead code elimination algorithms and one dead store elimination algorithm.

[3] `LOG_LINKS`, which are now only used by combine have been integrated into `combine`. The use of this datastructure is discouraged but `combine` is difficult to rewrite.

3.3.1 The First Dead Code Elimination Algorithm

The first DCE was developed by Richard Sandiford of CodeSourcery. This algorithm is based on the optimistic dead code elimination in [3] but differs in two important ways: it uses use-def chains rather than SSA form and it currently does not utilize the control dependence graph to remove dead branches. The latter difference will be fixed when time permits.

The overall form of the algorithm is to

1. build use-def chains.
2. mark instructions that can never be dead as live. Everything else is assumed dead.
3. iteratively mark any instructions as live if it is used by something live.
4. delete everything marked live.

3.3.2 Dead Store Elimination

The dead store elimination was also developed by David Sandiford. It deals with two forms of dead stores: stores in the exit block that store to into the stack frame and stores, whose value is stored over before the value can be read.

To find the latter, a dataflow problem is solved where each symbolic address is modeled with position in the bit vector. The flow equations track which stores may reach other instructions.

3.3.3 The Second Dead Code Elimination Algorithm

The second dead code elimination algorithm was implemented by Kenneth Zadeck and is

similar in principle to the existing dead code elimination in FLOW; i.e. it is based on live variable analysis and processes the instructions on a block by block basis.

This algorithm is inferior to the first dead code elimination in that it cannot remove code that only depends on itself (dead induction variables) and cannot be modified to remove control dependent dead code.[4] However, this algorithm is much faster than the first because live variables is much less expensive to compute than use-def chains. Also, this algorithm is generally called as an almost free, side effect of building live variables, which are used for many other passes of the compiler.

The main difference between the algorithm in DF and the one in FLOW is that the basic blocks are processed in postorder. The initial value for liveness that is used at the bottom of the block is either value computed by DF if none of the successors has changed or the union of liveness at the top of the successors if any of them has changed.

As each block is processed (from last to first instruction) the liveness is kept up to date. When the top of the block is reached, this computed liveness is compared with the value at the top of the block computed by DF. If the values differ, the locally computed value replaces the value computed by DF and this block is marked as changed.

The only time that it is necessary to actually go back and re-solve the dataflow equations is when the live variable bitvector changes at the top of a block that is the destination of a cfg back edge. This is quite rare, particularly since DCE is called at the beginning of many passes of the backend. In the DCE in FLOW the equations are resolved if any instruction is deleted.

[4] This difference will only be important when the first dead code algorithm is enhanced with control dependence information.

3.3.4 Status

Currently the first DCE algorithm not called directly but is used as a part of the dead store elimination. When it is enhanced with the control dependence graph, it may be useful to call this as a separate pass at higher optimization levels.

It may also be possible to enhance the first DCE so that it can be called as a side effect of building use-def and def-use chains. The performance issues are not with the dead code part of the algorithm but with the building of the chains. However, in passes such as the modulo scheduler which builds both use-def and def-use chains, it may be possible to integrate the first DCE algorithm and fix up the chains.

4 Abstractions and API

The abstractions used by the dataflow analysis are meant to be both usable, and efficient, at the same time.

There are several important structures provided by the dataflow engine, the main ones being the dataflow reference structure and the insn info structure.

The dataflow reference structure is the heart of dataflow information. It is generated by the dataflow scanner, and represents a def or a use of a register (IE a reference to a register). The information it provides consists of:

- `REG`, the register this reference is referencing.
- `BB`, the basic block in which the instruction occurs.
- `INSN`, a pointer to the instruction containing the reference.

- LOC, a pointer to the place in the instruction containing the reference.
- CHAIN, a pointer to the chain of uses of this reference if it is at def, or a chain of defs of this reference if it is a use.
- ID, a unique id for this reference.
- TYPE, whether the reference is a use or a def, and if it is a use, whether it is a regular use, or part of a memory addressing operation.
- FLAGS, various informational flags about the reference, such as whether it is a clobber, whether the reference is artificial, whether it occurs in a note, and other useful pieces of information.

The dataflow insn info structure is the second most used dataflow structure. It provides information about each instruction in the program consisting of:

- USES, the list of register uses in this instruction.
- DEFS, the list of register definitions in this instruction.
- MWREGS, the list of multiword register uses and defs in this instruction (this is separated out for use by REG_DEAD and REG_UNUSED note generation).

In addition to these structures, there are some small and easily understood structures used by various simpler problems, as well as various tables that contain pointers to the structures defined above . An example of one of these table is the the register-use and register-def table, which is a table of all the def/use structures for a given register.

4.1 Using predefined problems

Using the predefined problems is very simple. An example:

```
struct df *df
  = df_init (DF_HARD_REGS);
df_lr_add_problem (df, 0);
df_analyze (df);

bbinset
  = DF_LR_BB_INFO(df, bb)->in;
bboutset
  = DF_LR_BB_INFO(df, bb)->out;
df_finish (df);
```

The call to df_init initializes the dataflow instance, and is passed flags that tell the DF instance about the details of the info you need. The current flags include DF_SUBREGS, which includes information about subregs, DF_HARD_REGS, which includes information about machine registers, and DF_EQUIV_NOTES, which provides information about references that occur in REG_EQUIV notes.

Once initialized, adding problems to the dataflow instance only requires calling the right df_add_problem function. There is one for each predefined problem. To add def-use or use-def chains, df_chain_add_problem should be called with either DF_UD_CHAIN, DF_DU_CHAIN, or both of these flags or'ed together.

After adding all the problems you want to the DF instance, calling df_analyze will cause the dataflow engine to perform all the dataflow and generate the info you have requested.

Once that is done, the info will be stored in various structures, depending on what you asked for.

Problems that generate *IN* and *OUT* sets usually have macros to access these sets, such as `DF_LIVE_IN` and `DF_LIVE_OUT`.

Problems that generate register info, reg-defs, reg-uses, or instruction info, generally have macros to access the appropriate tables, like `DF_INSNS_GET`.

The definitions of all of these macros and tables can be found in the file `df.h`.

4.2 Defining Your Own Problem

DF can solve many dataflow problems in addition to the ones defined in Section 3.1. The full harness that DF uses is somewhat complex and is there to make it very easy to use the canned problems. Only a small amount of the structure is necessary to understand if you wish to define your own problem.

The function `df_simple_iterative_dataflow` has been defined to allow simple dataflow problems defined over some part of the control flow graph to be solved. There are eight parameters to this function:

`dir` Either `DF_FORWARD` or `DF_BACKWARD`. We do not currently do bidirectional, but could add if there was a need.

`init_fun` This function of type *df_init_function* initializes the *in* and *out* sets before starting to solve the equations.

`con_fun_0` This function of type *df_confluence_function_0* is the convolution function if the block has no logical predecessors. If the value of this parameter is `NULL`, the *in set* remains at the value it was initialized to. This is useful in obscure cases to deal with no return blocks in backwards problems. This case can never happen in a forwards problem because such a block would not appear reachable.

`con_fun_n` This function of type *df_confluence_function_n* is the convolution function if the block has one or more logical predecessors.

`trans_fun` This function of type *df_transfer_function* is the transfer function through the block.

`blocks` A bitmap that defines which blocks are to be processed.

`postorder` An array of int that contains the basic blocks in `blocks` in postorder.

`n_blocks` The number of blocks in postorder.

5 Incremental Dataflow

Some of the api of FLOW and the original implementation of DF is based on the assumption that reasonable algorithms exist for performing dataflow analysis incrementally.

The first algorithms for incremental dataflow were the phd dissertations of Ryder [6] and one of the authors of this paper, Zadeck [8]. This was followed other for a period of about 15 years. In that time the community failed to develop any algorithms that were clearly superior to well engineered optimizations that did not rely on being able to update dataflow results.

The problem can be best illustrated with in the live variables problem (but correspondingly similar problems exist for all dataflow problems). There are a large number of correct solutions for the dataflow equations, there is only a

single minimal solution. For the live variables problem, as is true for any set union bit vector problem, the minimal correct solution is the one with the smallest total number of one bits in the solution bit vectors.

`update_life_info_in_dirty_blocks` will generally find a correct solution.[5] However, for changes where uses are deleted, defs are added and/or edges are moved, this function will generally not find the minimal solution: i.e., it finds solutions that contain extra one bits in some vectors.

For illustration purposes, let us take the case where there are several uses of a particular pseudo register *r* and we wish to remove some of these uses.

Finding the minimal solution is trivial if the program has no loops. The problems in finding a solution arise at join points (live variables is a backwards problem: the join points are the conditional branches in the control flow graph.) Since the solution at the bottom of join point *b* is the union of the solutions at the top of the successor of *b*, the question that must be asked when one of the difficult changes is made, is what caused one bits in those successor blocks. If any of the bits in the successors are derived from the uses of *r* that remain, then the bit for *r* at the bottom of *b* is set. However, if all of the bits were derived from instances that were deleted, the bit for *r* at the bottom of *b* must be cleared.

There are two ways to approach designing an algorithm to solve this problem: one can either clear all of the bits in all of the blocks associated with *r* or one can build auxiliary datastructures to hold the information for where the bits come from. The first approach means that one is solving the offline algorithm whenever one makes the change.

[5] Those cases where it fails to find a correct solution are bugs that are generally easy to fix.

The second approach is the one that was pursued by the incremental dataflow community in various forms. For the live variables problem, the information necessary to track changes is equivalent to the information provided by the more expensive reaching uses problem. However, to keep the information in the reaching uses up to date when structural changes are made to the flow graph requires even more expensive path information. While many interesting datastructures were investigated, the overhead involved in keeping these up to date was rarely worth the trouble. This result, coupled with the fact that most optimizations can be implemented without the need for incremental updates basically killed the area.

For these reasons, we decided that it was time to face the fact that incremental dataflow was not going to happen and have reimplemented DF without a non working incremental API. There are a few passes such as if conversion that have required extensive revision, but for most passes the changes have been simple and mechanical.

6 Demand Driven Backend Passes

In addition to improving the quality of the dataflow analysis of the backend, it has been our goal to improve the modularity of the backend. It is highly desirable to be able to reorder the passes in a compiler. We have addressed a number of these issues in the dataflow rewrite.

There are two ways to achieve modularity in a compiler: (1) implement a set of invariants that must be true at the end of each pass or (2) make sure that each pass that needs some invariant to be true applies the steps to make it that way.

One of the reasons that the backend is difficult to deal with is that many of the dependencies

between the phases do not fall into either of these categories. The components of FLOW are at the core of many of these problems and as we have replaced flow with DF, we have done so in a manner that is consistent with one of the above categories.

- At the beginning of any part of the back-end that has a correct control flow graph, any of the DF problems can be computed. Furthermore, it is expected that each pass, compute exactly the dataflow problems that it needs for itself and at the end of that pass, the flow information is discarded.[6]

- Dead code elimination is only performed if the phase where the analysis is being done, requires the program to be clear of dead code. Any phase that my produce dead code leaves that code.

  There are, at higher optimization levels, passes near the end that do a comprehensive job of cleaning up the dead code.

- Like dce, CFG Cleanup now is only performed if the phase before phases that would be inhibited by having an untidy control flow graph. The long term plan with this is to change this from a phase that eight different modes to one that has two modes: a lightweight one that is part of other passes, and a heavy weight one that is powerful and not particularly inexpensive.

While this cleanup removes many of the inter-pass dependencies, some remain, in particular there are still many issues with the way register information is built and maintained throughout the pass stream.

[6]The only exception to this are `REG_DEAD` and `REG_UNUSED` notes that are left until the next pass that uses them cleans them up.

7 Status and Results

The work so far has concentrated on been replacing the use of `flow.c` with dataflow provided by `df.c`. Currently, the code for most of our work is on the `dataflow-branch` in the GCC SVN repository. On this branch, `flow.c` is not used at all for global liveness calculation, or dead code elimination. All passes, except combine, use dataflow instances to do their work. This work should begin soon.

8 Acknowledgments

We would like to than everyone who has helped with the dataflow branch. This includes, Steven Bosscher, David Edelsohn, Jan Hubicka, Richard Sandiford, Ian Lance Taylor.

References

[1] F. E. Allen. Control flow analysis. *SIGPLAN Notices*, 5, July 1970.

[2] D. C. Atkinson and W. G. Griswold. Implementation techniques for efficient data-flow analysis of large programs. *Proc. of the 2001 International Conf. on Software Maintenance*, November 2001.

[3] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] S. L. Graham and M. N. Wegman. A fast and usually linear algorithm for global flow analysis. *J. ACM*, 23(1):172–202, January 1976.

[5] M. S. Hecht and J. D. Ullman. A simple algorithm for global data flow analysis problems. *SIAM J. Computing*, 4(4):519–532, December 1975.

[6] B. G. Ryder. Incremental data flow analysis. *Conf. Rec. Tenth ACM Symp. on Principles of Programming Languages*, pages 167–176, January 1983.

[7] R. E. Tarjan. Testing flow graph reducibility. *J. Computer and System Sciences*, 9:355–365, December 1974.

[8] F. K. Zadeck. Incremental data flow analysis in a structure program editor. *Proc. SIGPLAN'84 Symp. on Compiler Construction*, pages 132–143, June 1984. Published as *SIGPLAN Notices* Vol. 19, No. 6.