High-Level Loop Optimizations for GCC

Daniel Berlin
IBM T.J. Watson Research Center
dberlin@us.ibm.com

David Edelsohn
IBM T.J. Watson Research Center
dje@watson.ibm.com

Sebastian Pop
Centre de recherche en informatique, École des mines de Paris
sebastian.pop@cri.ensmp.fr

Abstract

This paper will present a design for loop optimizations using high-level loop transformations. We will describe a loop optimization infrastructure based on improved induction variable, scalar evolution, and data dependence analysis. We also will describe loop transformation opportunities that utilize the information discovered. These transformations increase data locality and eliminate data dependencies that prevent optimization. The transformations also can be used to enable automatic vectorization and automatic parallelization functionality.

The TreeSSA infrastructure in GCC provides an opportunity for high level loop transforms to be implemented. Prior to the Loop Nest Optimization effort described in this paper, GCC has performed no cache reuse, data locality, parallelization, or loop vectorization optimizations. It also had no infrastructure to perform data dependence analysis for array accesses that are necessary to apply these transformations safely. We have implemented data dependence analysis and linear loop transforms on top of TreeSSA, which provides the following features:

1. A data dependence framework for determining whether two data references have a dependence. The core of the dependence analysis is a new, low-complexity algorithm for the recognition of scalar evolutions that tracks induction variables across a def-use graph. It is used to determine the legality of various transformations, including the vectorization transforms being implemented, and the matrix based transformations.

2. A matrix-based transformation method for rearranging loop nests to optimize locality, cache reuse, and remove inner loop dependencies (to help vectorization and parallelization). This method can perform any legal combination of loop interchange, scaling, skewing, and reversal to a loop nest, and provides a simple interface to doing it.

1 Introduction

As GNU/Linux tackles high-performance scientific and enterprise computing challenges, GCC (the GNU Compiler Collection)—the GNU/Linux system compiler—is challenged as well. Modern computer processors and systems are implemented with advanced features that require greater compiler assistance to achieve high performance. Many techniques developed for vector and parallel architectures have found new application to superscalar and VLIW computer architectures, and to systems with large memory latencies, more complicated function unit pipelines, and multiple levels of memory caches.

The TreeSSA optimization infrastructure[11] in GCC provides an enhanced framework for program analysis. Improved data dependence information allows the compiler to transform an algorithm to achieve greater locality and improved resource utilization leading to improved throughput and performance.

The GCC Loop Nest Optimizer joins a powerful loop nest analyzer with a matrix transformation engine to provide an extensible loop transformation optimizer that addresses unimodular and scaling operations. The data dependence analyzer is based on a new algorithm to track induction variables without being limited to specific patterns. The matrix transformation functionality uses a building block design that allows many of the standard toolbox of optimizations to be implemented. A similar matrix toolkit is used by proprietary commercial compilers. The pieces form a clean and maintainable design, avoiding an ad hoc set of optimizers with similar technical requirements.

2 Scalar Evolutions

After the *genericization* and *gimplification*, the loop structures of the compiled language are transformed into lower level constructs that are common to the imperative languages: three address assignments, gotos and labels. In order to retrieve the classic representation of loops from the GIMPLE representation[9], the natural loop structures are detected, as described in the Dragon Book [1], then based on the analysis of the instructions contained in the loops bodies, the indexes and the bounds of loops are detected.

We describe in this section the algorithm used for analyzing the properties of the scalar variables updated in a loop. The main extracted properties are the number of iterations of a loop, and a form that allows a fast evaluation of the values of a variable for a given iteration. Based on these two properties, it is possible to extend the copy constant propagation pass after the crossing of a loop, and the elimination of redundant checks. A further analysis extracts a representation of the relations between the reads and the writes to the memory locations referenced by arrays, and the classic data dependence tests.

2.1 Representation of the Program

The analyzed program is in *Static Single Assignment* form [10, 5], that ensures the uniqueness of a variable definition, and a fast retrieval of the definition from a use. These properties have lead to the design of an efficient algorithm that extracts the scalar evolutions in a bidirectional, non-iterative traversal of the control-flow graph.

2.2 Chains of Recurrences

The information extracted by the analyzer is encoded using the chains of recurrences (chrecs) representation proposed in [3, 6, 17, 14, 13]. This representation permits fast evaluations of a function for a given integer point, using the Newton's interpolation formula. In the following, we present an intuitive description of the chrecs based on their interpretation, then the link between the notation of the chrecs and the semantics of the polynomial functions.

```
r1 = 0
r2 = 1
r3 = 2
loop (ℓ1)
| r1 += r2
| r2 *= r3
end1:
```

Figure 1: Univariate evolution

$$\begin{tikzcd}\boxed{2} \arrow[r, "*"] & \boxed{1} \arrow[r, "+"] & \boxed{0} \arrow[r, "\text{copy}"] & \text{result}\end{tikzcd}$$

Figure 2: Data-flow interpretation

```
r4 = 9
r5 = 8
r6 = 7
loop (ℓ2)
| loop (ℓ3)
| | r6 += r5
| end3:
| r5 += r4
end2:
```

Figure 3: Multivariate

2.2.1 Interpretation of Chrecs

The main property modeled by the chrecs is the effect of the iterative execution of a program on storage points. Each storage point contains an initial value on the entry of a loop. The stored value evolves during the execution of the loop following the operations of the updating statements. The description of the updating expressions is embedded in the chrecs representation, such that it is possible to retrieve a part of the original program from the chrec representation. In other words, only the interesting scalar properties are selected, and the undecidable scalar properties are abstracted into the unknown element. In the following, the chrecs representation is illustrated by intuitive examples based on two interpretation models: using a register based machine, and a data-flow machine.

In the register based machine, the coefficients of a chrec are stored in registers. Then, the value of a register is updated at each iteration of a loop, using the operation specified in the chrec on its own value and the value of the register on its right. The first example illustrates the interpretation of a chrec that vary in a single loop.

Example 1 (Univariate chrec on register machine) *Figure 2.2.1 illustrates the interpretation of the chrec $\{0, +, \{1, *, 2\}_1\}_1$. The registers $r1$, $r2$, and $r3$ are initialized with the coefficients of the chrec. Then, the registers are updated in the loop specified in index of the chrec: loop 1. The register $r2$ is updated in the loop, and its evolution is described by the chrec $\{1, *, 2\}_1$. $r1$ is accumulating the successive values of $r2$ starting from its initial value 0, and consequently it is described by the chrec $\{0, +, \{1, *, 2\}_1\}_1$.*

Another intuitive description of the chrecs is given by the data-flow model: the nodes of an oriented graph contain the initial conditions of the chrec, while the oriented edges transfer information from a node to another and perform an operation on the operands. Figure 2 illustrates the data-flow machine that interprets the chrec from Example 1.

Finally, the last example illustrates the interpretation of a chrec that vary in two loops.

Example 2 (Multivariate chrec on register machine) *In Figure 2, the register $r6$ can be described by the multivariate scalar evolution $\{7, +, \{8, +, 9\}_2\}_3$. The value of $r6$ is incremented at each iteration of loop 3 by the value contained in $r5$ that vary in loop 2.*

In the register based machine, the value of a chrec at a given integer point is computed by successively evaluating all the intermediate values. The initial values of the chrec are stored in registers that are subsequently updated at each iteration step. One of the goals of the analyzer is to detect these iterative patterns, and then to recognize, when possible, the computed function. The link between the chrecs and the classic polynomial functions is described in the next subsection.

2.2.2 Semantics of Chrecs

As described in the previous works [3] Newton's interpolation formula is used for fast evaluation of the chrec at a given integer point. The evaluation of the chrec $\{c_0, +, \ldots, +, c_k\}$, at an integer point x is given by the formula

$$\{c_0, +, \ldots, +, c_k\}(x) = \sum_{i=0}^{k} c_i \binom{x}{i}$$

with $c_0, \ldots, c_k$ integer coefficients. In the peculiar case of linear chrecs, this formula gives

$$\{base, +, step\}(x) = base + step \cdot x$$

where $base$ and $step$ are two integer constants. As we will see, it is possible to handle symbolic coefficients, but the above formula for evaluating the chrecs is not always true.

2.2.3 Symbolic Chrecs

We have extended the classic representation of the scalar evolution functions by the use of parameters, that correspond to unanalyzed variables. The main purpose of this extension is to free the analyzer from the ordering constraints that were proposed in the previous versions of the analyzer. The parameters allow the analyzer to postpone the analysis of some scalar variables, such that the analyzer establishes the order in which the information is discovered in a natural way.

However, this extension leads to a more expressive representation, on which the Newton interpolation formula cannot be systematically used for fast evaluation of the chrec, because some of the parameters can stand for a function. In order to guarantee that all the coefficients of the chrec have scalar (non varying) values, the last step of the analysis fully instantiate all the parameters. When the instantiation fails, the remaining parameters are all translated into the unknown element, $\top$.

2.2.4 Peeled Chrecs

We have proposed another extension of the classic chrecs representation in order to model the variables that have an initial value that is overwritten during the first iteration. For representing the peeled chrecs, we have chosen a syntax close to the syntax of the SSA phi nodes because the symbolic version of the peeled chrec is the loop phi node itself. The semantics of the peeled chrecs is as follows:

$$(a, b)_k = \begin{cases} a, & \text{during the first iteration of loop k,} \\ b & \text{otherwise.} \end{cases}$$

where a and b are two chrecs that can be in a symbolic form. The peeled chrecs are built whenever the loop phi node does not define a strongly connected component over the SSA graph. The next section describes in more details the extraction algorithm.

2.3 Extraction Algorithm

Figure 4 presents the algorithm that computes the scalar evolutions for all the loop-ϕ nodes of the loops. The scalar evolution analyzer is composed of two parts: ANALYZEEVOLUTION returns a symbolic representation of the scalar evolution, and the second part INSTANTIATEEVOLUTION completes the analysis by instantiating the symbolic parameters. The

```
Algorithm: ComputeEvolutions
Input: SSA representation of the procedure
Output: a chrec for every variable defined by loop-$\phi$ nodes
  For each loop $l$
    For each loop-$\phi$ node $n$ in loop $l$
      InstantiateEvolution(AnalyzeEvolution($l$, $n$), $l$)

Algorithm: AnalyzeEvolution($l$, $n$)
Input: $l$ the current loop, $n$ the definition of an SSA name
Output: chrec for the variable defined by $n$ within $l$
  v ← variable defined by $n$
  $ln$ ← loop of $n$
  If $n$ was analyzed before Then
    $res$ ← evolution of $n$
  Else If $n$ matches "v = constant" Then
    $res$ ← constant
  Else If $n$ matches "v = a" Then
    $res$ ← AnalyzeEvolution($l$, a)
  Else If $n$ matches "v = a ⊙ b" (with $\odot \in \{+, -, *\}$) Then
    $res$ ← AnalyzeEvolution($l$, a) ⊙ AnalyzeEvolution($l$, b)
  Else If $n$ matches "v = loop-$\phi$(a, b)" Then
    (notice a is defined outside loop $ln$ and b is defined in $ln$)
    Search in depth-first order a path from b to v:
    (exist, update) ← DepthFirstSearch($n$, definition of b)
    If (not exist) (i.e., if such a path does not exist) Then
      $res$ ← $(\mathtt{a}, \mathtt{b})_l$
    Else If $update$ is $\top$ Then
      $res$ ← $\top$
    Else
      $res$ ← $\{\mathtt{a}, +, update\}_l$
  Else If $n$ matches "v = condition-$\phi$(a, b)" Then
    $eva$ ← InstantiateEvolution(AnalyzeEvolution($l$, a), $ln$)
    $evb$ ← InstantiateEvolution(AnalyzeEvolution($l$, b), $ln$)
    If $eva = evb$ Then
      $res$ ← $eva$
    Else
      $res$ ← $\top$
  Else
    $res$ ← $\top$
  Save the evolution function $res$ for $n$
  Return the evaluation of $res$ in loop $l$

Algorithm: DepthFirstSearch($h$, $n$)
Input: $h$ the halting loop-$\phi$, $n$ the definition of an SSA name
Output: (exist, update), exist is true if $h$ has been reached
  If ($n$ is $h$) Then
    Return (true, 0)
  Else If $n$ is a statement in an outer loop Then
    Return (false, $\bot$),
  Else If $n$ matches "v = a" Then
    Return DepthFirstSearch($h$, definition of a)
  Else If $n$ matches "v = a + b" Then
    (exist, update) ← DepthFirstSearch($h$, a)
    If exist Then Return (true, update + b),
    (exist, update) ← DepthFirstSearch($h$, b)
    If exist Then Return (true, update + a)
  Else If $n$ matches "v = loop-$\phi$(a, b)" Then
    $ln$ ← loop of $n$
    (notice a is defined outside $ln$ and b is defined in $ln$)
    If a is defined outside the loop of $h$ Then
      Return (false, $\bot$)
    $s$ ← Apply($ln$, AnalyzeEvolution($ln$, $n$),
        NumberOfIterations($ln$))
    If $s$ matches "a + t" Then
      (exist, update) ← DepthFirstSearch($h$, a)
      If exist Then
        Return (exist, update + $t$)
  Else If $n$ matches "v = condition-$\phi$(a, b)" Then
    (exist, update) ← DepthFirstSearch($h$, a)
    If exist Then Return (true, $\top$)
    (exist, update) ← DepthFirstSearch($h$, b)
    If exist Then Return (true, $\top$)
  Return (false, $\bot$)

Algorithm: InstantiateEvolution(chrec, $l$)
Input: chrec a symbolic chrec, $l$ the instantiation loop
Output: an instantiation of chrec
  If chrec is a constant c Then Return c
  Else If chrec is a variable v Then
    Return AnalyzeEvolution($l$, v)
  Else If chrec is of the form $\{e_1, +, e_2\}_{l'}$ Then
    $i_1$ ← InstantiateEvolution($e_1$, $l$)
    $i_2$ ← InstantiateEvolution($e_2$, $l$)
    Return $\{i_1, +, i_2\}_{l'}$
  Else If chrec is of the form $(e_1, e_2)_{l'}$ Then
    $i_1$ ← InstantiateEvolution($e_1$, $l$)
    $i_2$ ← InstantiateEvolution($e_2$, $l$)
    Return $(i_1, i_2)_{l'}$
  Else Return $\top$
```

Figure 4: Algorithm to compute scalar evolutions

main analyzer is allowed to discover only a part of the evolution information. The missing information is stored under a symbolic form, waiting for a full instantiation. The role of the instantiation is to determine an order for assembling the discovered information. After full instantiation, the extracted information corresponds to the classic chains of recurrences. In the rest of the section we analyze in more details the components of this algorithm, and give two illustration examples.

2.3.1 Description of the Algorithm

The cornerstone of the algorithm is the search and reconstruction of the symbolic update expression on a path of the SSA graph. Let us start with the description of the DepthFirstSearch algorithm. Each step is composed of a look-up of an SSA definition, and then followed by a recursive call of the search algorithm on the symbolic operands. The search halts when the starting loop-ϕ node is reached. When analyzing an assignment whose right-hand side is a sum, the search algorithm examines the first operand, and if the starting loop-ϕ node is not reachable through this path, it examines the second operand. When one of the operands contains a path to the starting loop-ϕ node, the other operand of the sum is added to the update expression, and the result is propagated to the lower search steps together with the reconstructed update expression. If the starting loop-ϕ node cannot be found by depth-first search, i.e., when DepthFirstSearch returns (false, $\perp$), we know that the definition does not belong to a cycle of the SSA graph: a peeled chrec is returned.

InstantiateEvolution substitutes symbolic parameters in a chrec. It computes their statically known value, i.e., a constant, a periodic function, or an approximation with intervals, possibly triggering other computations of chrecs in the process. The call to InstantiateEvolution is postponed until the end of the depth-first search, ensuring termination of the recursive nesting of depth-first searches, and avoiding early approximations in the computation of update expressions. Combined with the introduction of symbolic parameters in the chrec, postponing the instantiation alleviates the need for a specific ordering of the computation steps. This is a strong advantage with respect to the method by Engelen [14] based on a topological sort of all definitions. Furthermore, it becomes possible to recognize evolutions in every possible SSA graph, although some of them may not yield a closed form.

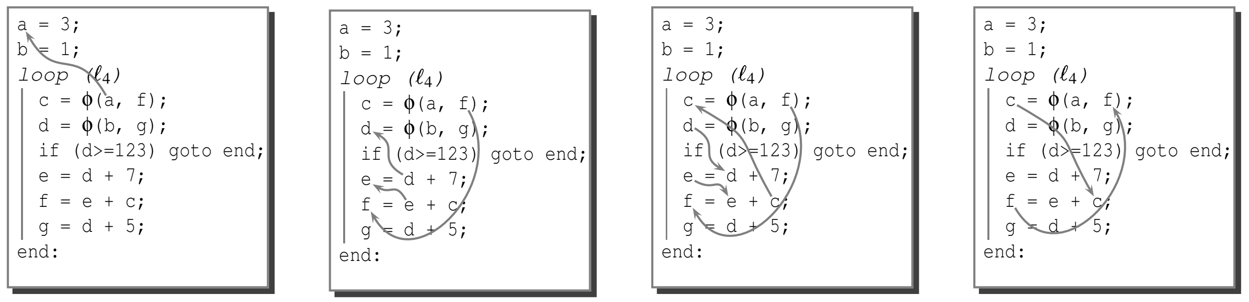
The overall effect of an inner loop may only be computed when the exit value of the variable is a function of the entry value. In such a case, the whole loop is behaving as a macro-increment operation. When the exit condition depends on affine chrec only, function NumberOfIterations deduces the number of iterations of the loop. Then we call Apply to evaluate the overall effect of the inner loop. Apply implements the efficient evaluation scheme for chrec based on Newton interpolation series (see Section 2.2.2). As a side-effect, the algorithm does indeed compute the loop-trip count for many natural loops in the control-flow graph. Our method recovers information that was lost during the lowering process or syntactically hidden in the source program.

2.3.2 Illustration Examples

Let us now illustrate the algorithm on two examples in Figures 5 and 6. In addition to clarifying the depth-first search and instantiation phases of the algorithm, this will exercise the recognition of polynomial and multivariate evolutions.

First example. The depth-first search is best understood with the analysis of `c = `ϕ`(a, f)` in the first example. The SSA edge of the initial value exits the loop, as represented in Figure 5.(1). Here, the initial value is left in a symbolic form, but GCC would replace it by 3 through constant propagation.

To compute the parametric evolution function of `c`, the analyzer starts a depth-first search algorithm, as illustrated in Figure 5.(2). We follow the update edge `c`$\rightarrow$`f` to the definition of `f` in the loop body: assignment `f = e + c`. The depth-first algorithm follows the first operand, `f`$\rightarrow$`e`, reaching the assignment `e = d + 7`, and finally follows the edge `e`$\rightarrow$`d` that leads to a loop-ϕ node of the same loop.



```
a = 3;
b = 1;
loop (ℓ4)
| c = φ(a, f);
| d = φ(b, g);
| if (d>=123) goto end;
| e = d + 7;
| f = e + c;
| g = d + 5;
end:
```

(1) Initial condition edge

```
a = 3;
b = 1;
loop (ℓ4)
| c = φ(a, f);
| d = φ(b, g);
| if (d>=123) goto end;
| e = d + 7;
| f = e + c;
| g = d + 5;
end:
```

(2) Search "c"

```
a = 3;
b = 1;
loop (ℓ4)
| c = φ(a, f);
| d = φ(b, g);
| if (d>=123) goto end;
| e = d + 7;
| f = e + c;
| g = d + 5;
end:
```

(3) Found the halting phi

```
a = 3;
b = 1;
loop (ℓ4)
| c = φ(a, f);
| d = φ(b, g);
| if (d>=123) goto end;
| e = d + 7;
| f = e + c;
| g = d + 5;
end:
```

(4) the "returning path"

Figure 5: The first example

Since this is not the loop-ϕ node from which the analyzer has started the depth-first search, the search continues on the other operands that were not yet analyzed: back on `e = d + 7`, operand `7` is a scalar and there is nothing more to do, then back on `f = e + c`, the edge `f`$\rightarrow$`c` is followed to the starting loop-ϕ node, as illustrated in Figure 5.(3).

At this point, the analyzer has found the strongly connected component that corresponds to the path of iterative updates. Following this path in execution order, as illustrated in Figure 5.(4), the analyzer builds the update expression as an aggregation of the operands that are not on the updating path: in this example, the update expression is just `e`. As a result, the analyzer assigns to the definition of `c` the parametric evolution function $\{\texttt{a}, +, \texttt{e}\}_1$.

The instantiation of the parametric expression $\{\texttt{a}, +, \texttt{e}\}_1$ starts with the substitution of the first operand of the chrec: $\texttt{a} = 3$, then the analysis of `e` is triggered. First the assignment `e = d + 7` is analyzed, and since the evolution of `d` is not yet known, the edge `e`$\rightarrow$`d` is taken to the definition `d =` ϕ`(b, g)`. Since this is a loop-ϕ node, the depth-first search algorithm is used as before and yields the evolution function of `d`, $\{b, +, 5\}_1$, and after instantiation, $\{1, +, 5\}_1$. Finally the evolution of `e = d + 7` is computed: $\{8, +, 5\}_1$. The final result of the instantiation yields the polynomial chrec of `c`: $\{3, +, 8, +, 5\}_1$.

```
h = 3;
loop (ℓ5)
| i = φ(h, x);
| loop (ℓ6)
| | j = φ(i, k);
| | k = j + 1;
| | t = j - i;
| | if (t>=9) goto end6;
| end6:
| x = k + 3;
| if (x>=123) goto end5;
end5:
```

Figure 6: Second example

Second example. We will now compute the evolution of `x` in the nested loop example of Figure 6, to illustrate the recognition of multivariate induction variables and the computation of the trip count of a loop. The first step consists in following the SSA edge to the definition of `x`. Consider the right-hand side of the definition: since the evolution of `k` along loop 5 is not yet analyzed, we follow the edge `x`$\rightarrow$`k` to its definition in loop 6, then `k`$\rightarrow$`j` ending on the definition of a loop-ϕ node.

At this point we know that `j` is updated in loop 6. The initial condition `i` is kept under a symbolic form, and the iteration edge `j`$\rightarrow$`k` is followed in the body of loop 6. The depth-first search algorithm starts from right-hand side of the assignment `k = j + 1`: following the edge `k`$\rightarrow$`j` we end on the loop-ϕ node from which we have started the search, meaning that the search succeeded. Back on the path `j`$\rightarrow$`k`$\rightarrow$`j`, the analyzer gathers the evolution of `j` along the whole loop, an increment of 1, and ends on the following symbolic chrec: $\{\texttt{i}, +, 1\}_6$.

From the evolution of `j` in the inner loop, the analyzer determines the overall effect of loop 6 on `j`, that is the evaluation of function $f(n) = n + \texttt{i}$ for the number of iterations of loop 6. Fortunately, the exit condition is the simple expression `t>=9`, and the chrec for `t` (or `j - i`) is $\{0, +, 1\}_6$, an affine (non-symbolic) expression. It comes that 10 iterations of loop 6 will be executed for each iterations of loop 5. Calling APPLY(6, $\{\texttt{i}, +, 1\}_6$, 10) yields the overall effect `j = i + 10`.

The analyzer does not yet know the evolution function of `i`, and consequently it follows the SSA edge to its definition: `i = `ϕ`(h, x)`. Since this is a loop-ϕ node, the analyzer must determine its evolution in loop 5. We ignore the edge to the initial condition, and walk back the update edge, searching for a path from `i` to itself.

First, edge `i`$\rightarrow$`x` leads to the statement `x = k + 3`, then following the SSA edge `x`$\rightarrow$`k`, we end on a statement of the loop 6. Again, edge `k`$\rightarrow$`j` is followed, ending on the definition of `j` that we have already analyzed: $\{\texttt{i}, +, 1\}_6$. The depth-first search selects the edge `j`$\rightarrow$`i`, associated with the overall effect statement `j = i + 10` that summarizes the evolution of the variable in the inner loop. We finally reached the starting loop-ϕ node `i`. From this point, the path is walked back gathering the stride of the loop: 10 from the assignment `j = i + 10`, then 1 from the assignment `k = j + 1`, and 3 from the last assignment on the return path. We have computed the symbolic chrec of `i`: $\{\texttt{h}, +, 14\}_5$.

The last step consists in the propagation of this evolution function from the loop-ϕ node of `i` to the original node of the computation: the definition of `x`. Back from `i` to `j`, we can partially instantiate its evolution: a symbolic chrec for `j` is $\{\{\texttt{h}, +, 14\}_5, +, 1\}_6$. Then back to `k = j + 1` we get a symbolic chrec for `k`: $\{\{\texttt{h} + 1, +, 14\}_5, +, 1\}_6$; and finally back to `x = k + 3`, we get a symbolic chrec for `x`: $\{\texttt{h} + 14, +, 14\}_5$. A final instantiation of `h` yields the closed form of `x` and all other variables.

As we have seen, the analyzer computes the evolution functions on demand, and caches the discovered informations for later queries occurring in different analyzes or optimizations that make use of the scalar evolution information. In the next section, we describe the applications that use the informations extracted by the analyzer.

2.4 Applications

Scalar optimizations have been proposed in the early days of the optimizing compilers, and have evolved in speed and in accuracy with the design of new intermediate representations, such as the SSA. In this section we describe the extensions to the classic scalar optimization algorithms that are now enabled by the extra information on scalar evolutions. Finally, we give a short description of the classic data dependence tests.

2.4.1 Condition Elimination

In order to determine the number of iterations in a loop, the algorithm computes the first iteration that does not satisfy the condition that keeps the execution inside the loop. This same algorithm can be used on other condition expressions that don't keep the loop exit, such that the algorithm determines the number of iterations that fall in the then or in the else clauses. Based on the total number of iterations in the loop it is then possible to determine whether a branch is always taken during the execution of the loop, in which case the condition can be eliminated together with the dead branch.

Another approach for the condition elimination consists in using symbolic techniques for proving that two evolutions satisfy some comparison test for all the iterations of the loop. In the case of an equality condition, the algorithm is close to the value numbering technique, and is described in the next subsection.

2.4.2 Value Numbering

The value numbering is a technique based on a compile-time classification of the values taken at runtime by an expressions. The compiler determines the inclusion property of an expression into a class based on the results of an analysis: in the classic algorithms, the analysis is a propagation of symbolic AST trees [10, 12].

Using the information extracted by the scalar evolution, the classification can be performed not only on constants and symbols, but also on evolution functions, or on the scalar values determined after crossing the loop.

2.4.3 Extension of the Constant Propagation

The field of action of the classic conditional constant propagation (CCP) is limited to code that does not contain loop structures. When the scalar evolution analyzer is asked for the evolution function of a variable after crossing a loop with a static count, it computes a scalar value, that can be further propagated in the rest of the program. This removes the restriction of the classic CCP, where constants are only propagated from their definition to the dominance frontier.

2.4.4 Data Dependence Analysis

Several approaches have been proposed for computing the relations between the reads and the writes to the memory locations referenced by arrays. The compiler literature [4, 15, 10] describes loop normalizations, then the extraction of access functions by pattern matching techniques, while more recent works [16], rely on the discovery of monotonicity properties of the accessed data. An important part of the efficiency of these approaches resides in the algorithm used for determining the memory access patterns, while the subscript intersection techniques remain in the same range of complexity.

Our data dependence analyzer is based on the classic methods described in [4, 2]. These techniques are well understood and quite efficient with respect to the accuracy and the complexity of the analysis. However, our data dependence analyzer can be extended to support the newer developments on monotonicity properties proposed by Peng Wu *et al.* [16], since the scalar evolution analyzer is able to extract not only chrecs with integer coefficients, but also evolution envelopes, that occur whenever a loop contains updating expressions in a condition clause. In the following we shortly describe the classic data dependence analyzer, and show how to extend it for handling the monotonicity informations exposed by the scalar analyzer.

A preliminary test, that avoids unnecessary further computations, classifies the relation between two array accesses as *non dependent* when their base name differ. Thus, the remaining dependence tests consider only tuples of accesses to the same base name array.

The first test separately analyzes each tuple of access functions in each dimension of the analyzed array. This tuple is in general called a subscript. A basic test classifies a subscript following the number of loops in which it is varying. The three classes of subscripts, constants, univariate, or multivariate, have different specific dependence tests that avoids the use of the multivariate generic solver.

The iterations for which a subscript access the same element, or conflicting iterations, are computed using a classic Diophantine[1] equation solver. The resulting description is a tuple of functions that is encoded yet again using the chrecs representation. Banerjee presents a formal description [4] of the classic data dependence tests that we just sketch in this paper. The basic idea is to find a first solution (or the first conflicting iteration) to the Diophantine equation, then to deduce all the subsequent solutions from this initial one: this is represented as a linear function under the form of a chrec as base plus step. The gcd test provides an easy way to prove that the initial solution does not exist, and consequently it proves the *non dependence* property and stops the algorithm before the resolution of the Diophantine equation. The most costly part of this dependence test is effectively the resolution of the Diophantine equation, and more precisely the determination of the initial solution.

Once the conflicting iterations are known, the analyzer is able to abstract this information into a less precise representation: the distance per subscript information. When the conflicting iterations have a same evolution step, the difference of their base gives the distance at which the conflicts occur. When the steps of the conflicting iterations are not equal, the dependence relation is not captured by the distance description.

In a second step, the analyzer refines the dependence relations using the information on several subscripts. The subscript coupling technique allows the disambiguation of more non dependent relations in the case of multidimensional arrays. The classic per loop distances are computed based on the per subscript distance information. When a loop carries two different distances for two different subscripts, the relation is classified to be *non dependent*.

As we have seen, the current implementation of the dependence analyzer is based on the classic dependence tests. For this purpose, only the well formed linear access functions were selected for performing the dependence analysis. Among the rejected access functions are all those whose evolution is dependent on an element that was left under a symbolic form, or contain intervals. For all these cases, the conservative result of the analyzer is the *unknown dependence* relation. In the case of evolution envelopes, it is possible to detect independent data accesses based on the monotonicity properties, as proposed by Peng Wu *et al.* [16].

3 Matrix Transformations

3.1 Purpose

The reason for using matrix based transformations as opposed to separate loop transformations in conjunction are many. First, one can composite transformations in a much simpler way, which makes it very powerful. While any of the transformations described could be written as a sequence of simple loop transforms, determining the order in which to apply them to achieve the desired transformation is

[1]A Diophantine equation is an equation with integer coefficients.

non-trivial. However, with a matrix transform, one can generate the desired transformation directly. In addition, determining the legality of a given transformation is a simple matter of multiplication. The algorithm used also allows for completion of partial transforms.

3.2 Algorithm

The code generation algorithm implemented for GCC is based on Wei Li's Lambda Loop Transformation Toolkit [8]. It uses integer lattices as the model of loop nests and uses non-singular matrices as the model of the transforms. The implemented algorithm supports any loop whose bounds can be expressed as a system of linear expressions, where each linear expression can include loop invariants in the expression. This algorithm is in use by several commercial compilers (Intel, HP), including those known to perform these transformations quite well. This was a consideration in choosing it. Using this algorithm, we can perform any combination of the following transformations, simply by specifying the applicable transformation matrix.

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

```
DO I=1,3
   DO J=1,3
      A(I, 2*J) = J
   END DO
END DO
```

Figure 7: Original loop

$$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$$

```
DO U=1,3
   DO V=2,6,2
      A(U, V) = V/2
   END DO
END DO
```

Figure 8: Loop scaling

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

```
DO U=1,3
   DO V=1,3
      A(V, 2*U) = U
   END DO
END DO
```

Figure 9: Interchanged loop

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

```
DO U=1,3
   DO V=U + 1,U + 3
      A(U, 2*(V-U)) = 2*(V-U)
   END DO
END DO
```

Figure 10: Skewed loop

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

```
DO U=1,3
   DO V=-3,-1
      A(U, -2*V) = -V
   END DO
END DO
```

Figure 11: Reversed loop

The loops produced by applying these transforms to the loop in 7 can be seen in Figures 8, 9, 10, and 11 respectively.

This set of operations includes every unimodular operation (interchange, reversal, and skewing) plus scaling. The addition of scaling to the applicable transforms means that any non-singular transformation matrix can be applied to a loop, because they can all be reduced to some combination of the above. Scaling is useful in the context of loop tiling, and distributed memory code generation.

Legality testing is performed simply by multiplying the dependence vectors of the loop by the transformation matrix, and verifying that the resulting dependence vectors are lexicographically positive. This will guarantee that the data dependencies are respected in the loop nest generated.

The completion procedures allows completion of transformation matrices that contain the desired transformation for some portion of the loop, in a way that respects loop dependencies for the entire loop.

Consider the following loop:

```
DO I=4,8
   DO J=3,8
      A(I, J) = A(I-3, J-2) + 1
   END DO
END DO
```

The dependence matrix for this loop is

$$D = \binom{3}{2}$$

The outer loop can be made parallel if and only if it does not carry any dependences, i.e., the first entry of every dependence vector is 0. In its current form, this is obviously not true. We can make it parallel if we can find a transformation T such that every entry in the first row of TD is 0. We can easily satisfy that with the partial transform $\begin{pmatrix} 2 & -3 \end{pmatrix}$. However, this is not a complete transformation matrix because it does not specify what to do with the inner loop. The completion algorithm will complete this partial transform in a way that maintains the legality of the transform, i.e., respects dependences.

The full completion procedure is specified in [8]. It works by generating vectors that are independent of the existing row vectors in the partial transformation and within 90 degrees of each dependence vector.

3.3 Implementation

The GCC implementation of linear loop transforms is decomposed into several pieces: a matrix math engine, a transformation engine, and converters.

The matrix math engine implements various vector and matrix math routines necessary to perform the transformations (inversion, computation of Hermite form, multiplication, etc).

The transformation engine implements legality testing, rewriting of loop bounds, rewriting of loop bodies, and completion of partial transforms.

To transform a loop using GCC, we first need to convert it to a form usable by the code generation algorithm. There is a simple function which takes a GCC loop structure and produces a loopnest structure usable by the transformation engine. This loopnest structure consists of a system of linear equations representing the bounds of each loop.

Next, we perform legality testing. We have provided a function that takes the loopnest structure and a transformation matrix, and returns true if it is legal. This mainly is useful for transformations that were not produced by the completion algorithm, because that component only produces legal transforms.

Third, The loop bounds of the loopnest structure are rewritten using the aforementioned code generation algorithm.

Finally, we transform the loopnest structure back into real GIMPLE/Tree-SSA code. The subroutine accepts a loopnest structure and rewrites the actual loop nest code to match it. This involves two steps: first the new iteration variables, bounds, and exit condition are generated. Next, the body of the loop is transformed to eliminate uses of the old iteration variables. This procedure is straightforward: given a vector of source iteration variables S_i and a vector of the target iteration variables S_j, and the transformation matrix T, the function computes the source iteration variables in terms of the target iteration variables using the equation $S_i = T^{-1}S_j$. This calculation is performed for each statement in the loop, and the old uses are replaced with the new equations.

As a side note, all of these functions work independently of one another. In other words, as long as one supplies the function that rewrites loopnest structures into GCC code, one can reuse the components for other transformations.

3.4 Applications

Matrix based loop transforms can be used to improve effectiveness of parallelization and vectorization by removing inner loop dependencies that inhibit their substitution. They can also be used to perform spatial and temporal locality optimizations that optimize cache reuse [7].

These types of optimizations have the potential to significantly improve both application and benchmark scores. Memory locality optimizations are observed to produce speedup factors from 2 to 50 relative to the unmodified algorithm, depending on the application.

As an example of such a speedup, we'll take a well known SPEC® CPU2000 benchmark, SWIM[2].

SWIM spends most of its time in a single loop. By simply interchanging this loop, the performance can be improved sevenfold, as shown in Figure 12.

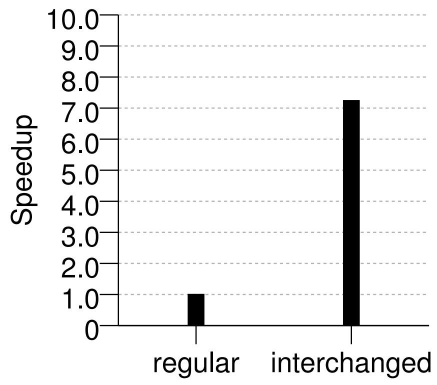


Figure 12: Effect of interchanging loop on SWIM

3.5 Future plans

Determination of a good transformation matrix for optimizing temporal and spatial locality is work in progress. There are many potential algorithms from which to choose. The authors are investigating research literature and other compiler implementations in order to choose a good algorithm to implement in GCC. An optimal transform matrix can be calculated in polynomial time for most loops encountered. The matrix transform method can be extended to perform loop alignment transforms, statement-based iteration space transforms, and other useful operations.

[2]`http://www.spec.org/`

4 Optimizations

4.1 Loop Optimizations

The new data dependence and matrix transformation functionality allows GCC to implement loop nest optimizations that can significantly improve application performance. These optimizations include loop interchange, unroll and jam, loop fusion, loop fission, loop reversal, and loop skewing.

Loop interchange exchanges the order of loops to better match use of loop operands to system characteristics, e.g., improved memory hierarchy access patterns or exposing loop iterations without dependencies to allow vectorization. When the transformation is safe to perform, the optimal ordering of loops depends on the target system. Depending on the intended effect, interchange can swap the loop with the greatest dependencies to an inner position within the loop nest or to an outer position within the nest. The effectiveness of the optimization is limited by alias and data dependence information.

The Unroll and jam transformation unrolls iterations of an outer loop and then fuses copies of the inner loop to achieve greater value reuse and to hide function unit latency. The optimal unrolling factor is a balance between scheduling and register pressure. The optimization is related to loop interchange and unrolling, so it similarly requires accurate alias and data dependence information.

Loop fusion combines loops to increase computation granularity and create asynchronous parallelism by merging independent computations with the same bounds into a single loop. This allows dependent computations with independent iterations to execute in parallel. Loop fusion requires appropriate alias and data dependence information, and also requires *countable loops*.

```
DO I=1,N
   A(I) = F(B(I))
END DO
Q = . . .
DO J=2,N
   C(I) = A(I-1) + Q*B(I)
END DO

      ⇓

Q = . . .
A(1)=F(B(1))
DO I=2,N
   A(I) = F(B(I))
   C(I) = A(I-1) + Q*B(I)
END DO
```

Figure 13: Example of Loop Fusion

Loop fission or distribution is the opposite of loop fusion: breaking multiple computations into independent loops. It can enable other optimizations, such as loop interchange and blocking. Another benefit is reduction of register pressure and isolation of vectorizable operations, e.g., exposing the opportunity to invoke a specialized implementation of an operator for vectors or using a vector/SIMD instruction. Vectorization is a balance between vector speedup and memory locality. Again, alias information, data dependence, and *countable loops* are prerequisites.

```
DO I=1,N
   S = B(I) / SQRT(C(I))
   A(I) = LOG(S)*C(I)
END DO

      ⇓

CALL VRSQRT(A,C,N)
DO I=1,N
   A(I) = B(I)*A(I)
END DO
CALL VLOG(A,A,N)
DO I=1,N
   A(I) = A(I)*C(I)
END DO
```

Figure 14: Example of Loop Fission

Loop reversal inverts the direction of iteration and loop skewing rearranges the iteration space to create new dependence patterns. Both optimizations can expose existing parallelism and aid other transformations.

4.1.1 Future Plans

After addressing the optimizations that can be implemented with initial loop transformation infrastructure, the functionality will be expanded to other well-known loop optimizations, such as loop tiling, interleaving, outer unrolling, and support for triangular and trapezoidal access patterns.

The goal function for high-level loop transformations is dependent on the target system. Communicating the system characteristics to the GCC loop optimizer is an ongoing area of investigation.

GCC's high-level loop optimization framework will not implement all, or even most, loop transformations in the first release—it is a work in progress, but an effective starting point from which to grow. Future enhancements to the framework will expand the functionality in two directions: implementing additional optimizations and reducing the restrictions on existing optimizations. The transformations first must be safe to enable for any application with well-defined numerical behavior. The optimizations will be enhanced to recognize more and different types of loops that can benefit from these techniques and improve application performance.

4.1.2 Helping the Compiler

The programmer can assist the compiler in its optimization effort while ensuring that the source code is easy to understand and maintain. This primarily involves simplifying memory analysis, loop structure, and program structure to aid the compiler.

Limiting the use of global variables and pointers allow the compiler to compute more thorough alias information, allowing the safety of transformations to be determined. Replacing pointers by arrays and array indexing is one such example.

Simplified loop structure permits more extensive analysis of the loops and allows easier modification of loops. Some loop transformation optimizations require *perfect loop nesting*, meaning no other code is executed in the containing loop, and most loop optimizations are limited to *countable loops*. A countable loop has a single entry point, a single exit point, and an iteration count that can be determined before the loop begins. A loop index should be a local variable whose address is not taken and avoids any aliasing ambiguity.

Programmers are encouraged to nest loops where possible and restructure loops to avoid branches within, into, or out of loops. Additionally, the programmer manually can perform loop fission to generate separate loops with simple bounds instead of a single loop with complicated bounds and conditionally-executed code within the loop.

4.2 Interacting with the Compiler: towards an OpenMP implementation

The OpenMP[3] standard can be seen as an extension to the C, C++, and Fortran programming languages, that provides a syntax to express parallel constructs. Because the OpenMP does not specify the compiler implementation, implementations range from the simple source to source preprocessors such as `OdinMP`[4] and `Omni`[5] to the optimizing compilers like `ORC`[6], that exploit the extra information provided by the programmer for better optimizing loop nests. Based on these implementations of the OpenMP norm, we give some reflections on a possible implementation of OpenMP in GCC.

[3]`http://www.openmp.org/`
[4]`http://odinmp.imit.kth.se/`
[5]`http://phase.hpcc.jp/Omni/`
[6]`http://ipf-orc.sourceforge.net/`

4.2.1 Source to Source Implementations

The source to source implementations of OpenMP include a parser that constructs an abstract syntax tree (AST) of the program, then a pretty printer that generates a source code from the AST. The AST is rewritten using the information contained in the OpenMP directives. The transformations involved in the rewriting of the AST are principally insertions of calls to a thread library, the creation of new functions, and restructuring of loop bounds and steps. The main benefit of this approach is that it requires a reduced compiler infrastructure for translating the OpenMP directives.

For implementing this source to source approach in GCC, two main components have to be designed:

- *a directive parser*, that is an extension of the parser for generating AST nodes for each directive, and
- *a directive rewriter*, that transforms the code in function of the directives.

In order to keep the code generation part generic for all the front-ends, a specific `OMP_EXPR` node could contain the information about the directives, until reaching the GENERIC, or the GIMPLE levels, the GIMPLE level having the benefit of being simpler, and more flexible for restructuring the code.

In the source to source model, the rewrite of the directives directly generates calls to a threading library, and the rest of the compiler does not have to handle the `OMP_EXPR` nodes. This kind of transformation tends to obfuscate the code by inserting calls to functions in place of the loop bodies, rendering the loop optimizations ineffective. In order to avoid this drawback we have to make the optimizers aware about the parallel constructs used by the programmer.

4.2.2 An Optimizing Compiler Approach

In the C, C++, and Fortran programming languages, the parallelism is expressed mainly using calls to libraries that implement threading or message passing interfaces. The compiler is not involved in the process of optimizing parallel constructs because the parallel structures are masked by the calls to the parallel library. In other programming languages, such as Ada and Java, parallel constructs are part of the language specification, and allow the compiler to manage the parallel behavior of the program. OpenMP directives fill a missing part of the C, C++, and Fortran programming languages with respect to the interaction of the programmer with the compiler for concurrent programming. It is in this extent that the OpenMP norm is interesting from the point of view of an optimizing compiler.

In order to allow the optimizers to deal with the parallel constructs in a generic way, the compiler has to provide a set of primitives for the parallel constructs. For the moment, the GENERIC level does not contain parallel primitives, and consequently the front-end languages have to lower their parallel constructs before generating GENERIC trees. In this respect, the OpenMP directives should not be different than other languages parallel constructs, and should not have a specific `OMP_EXPR` that allow these constructs to be propagated to the GIMPLE level for their expansion as described in section 4.2.1. The support of OpenMP in this context is to *genericize* the directives to their equivalent constructs in GENERIC and let the optimizers work on this representation. Using this approach would allow the compiler to choose the right degree of parallelism based on a description of the underlying architecture.

In the previous discussions on the GCC mailing lists, there were some good questions on whether we want to support the OpenMP standard in GCC, but rather than asking again this question, the authors would like to ask another question: do we want the generic optimizer to deal with concurrency aspects, and the programmer to be able to interact with the optimizer on parallel constructs?

5 Conclusions

The Loop Nest Optimizer provides an effective and modular framework for implementing high-level loop optimizations in GCC. Initial loop optimizations are built on a new loop data dependence analysis and matrix transformation engine infrastructure.

This work allows GCC to expand into a number of areas of optimization for high performance computing. The loop optimizations improve performance directly and provide a base on which to develop auto-vectorization and auto-parallelization facilities.

6 Acknowledgements

This work has been supported by IBM Research and the IBM Linux Technology Center. We wish to thank Zdenek Dvorak *SUSE* and Devang Patel and Andre Pinski *Apple Computer* for their collaboration, Bob Blainey *IBM Toronto* for insightful discussions, and the developers of Tree-SSA who created the opportunity for new enhancements to GCC.

Sebastian Pop would like to thank *École des mines de Paris (ENSMP)* who supports his PhD thesis, Georges-André Silber *(CRI-ENSMP)*, Albert Cohen *(Alchemy-INRIA)*, and Philippe Clauss *(ICPS-LSIIT)* to whom he is grateful for their guidance and collaboration.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence Based Approach*. Morgan Kaufman, San Francisco, 2002.

[3] O. Bachmann, P. S. Wang, and E. V. Zima. Chains of recurrences-a method to expedite the evaluation of closed-form functions. In *Proceedings of the international symposium on Symbolic and algebraic computation*, pages 242–249. ACM Press, 1994.

[4] U. Banerjee. *Loop transformations for restructuring compilers*. Kluwer Academic Publishers, Boston, 1994.

[5] 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. Program. Lang. Syst.*, 13(4):451–490, 1991.

[6] V. Kislenkov, V. Mitrofanov, and E. Zima. Multidimensional chains of recurrences. In *Proceedings of the 1998 international symposium on Symbolic and algebraic computation*, pages 199–206. ACM Press, 1998.

[7] W. Li. Compiler optimizations for cache locality and coherence. Technical Report TR504, Department of Computer Science, University of Rochester, 1994.

[8] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *1992 Workshop*

on Languages and Compilers for Parallel Computing, number 757, pages 391–405, New Haven, Conn., 1992. Berlin: Springer Verlag.

[9] J. Merrill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers' Summit*, pages 171–179, 2003.

[10] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[11] D. Novillo. Tree ssa: A new otimization infrastructure for gcc. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193, 2003.

[12] Y. N. Srikant and P. Shankar, editors. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 2002.

[13] R. van Engelen. Symbolic evaluation of chains of recurrences for loop optimization. Technical Report TR-000102, Computer Science Department, Florida State University, 2000.

[14] R. van Engelen. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the International Conference on Compiler Construction, ETAPS 2001, LNCS 2027*, pages 118–132, 2001.

[15] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, Reading, Mass., 1996.

[16] P. Wu, A. Cohen, D. Padua, and J. Hoeflinger. Monotonic evolution: an alternative to induction variable substitution for dependence testing. In *Proceedings of the 15th ACM International Conference on Supercomputing (ICS-01)*, pages 78–91, New York, June 17–21 2001. ACM Press.

[17] E. V. Zima. On computational properties of chains of recurrences. In *Proceedings of the 2001 international symposium on Symbolic and algebraic computation*, page 345. ACM Press, 2001.