A Model-based Distributed Continuous Quality Assurance Process to Enhance the Quality of Service of Evolving Performance-intensive Software Systems

Cemal Yilmaz[†], Arvind S. Krishna[‡], Atif Memon[†], Adam Porter[†], Douglas C. Schmidt[‡], Aniruddha Gokhale[‡], Balachandran Natarajan[‡]
[†]*Dept. of Computer Science, University of Maryland, College Park, MD 20742*
[‡]*Dept. of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN 37203*

Abstract

Time and resource constraints often force developers of highly configurable systems, such as that found in performance-intensive software, to assess their system's performance on very few configurations and to extrapolate from these to the entire configuration space, which allows many performance bottlenecks and sources of QoS degradation to escape detection until systems are fielded. To improve the assessment of performance across large configuration spaces, we present a model-based approach to developing and deploying a new distributed continuous quality assurance (DCQA) process. Our approach builds upon and extends the Skoll environment, which is developing and validating novel software QA processes and tools that leverage the extensive computing resources of worldwide user communities in a distributed, continuous manner to significantly and rapidly improve software quality. This paper describes how our new DCQA performance assessment process enables developers to run formally-designed screening experiments that isolate the most significant options. After that, exhaustive experiments (on the now much smaller configuration space) are conducted. We implemented this process using model-based software tools and executed it in the Skoll environment to demonstrate its effectiveness via two experiments on widely used QoS-enabled middleware. Our results show that model-based DCQA processes improves developer insight into the effect of system changes on performance at an acceptable cost.

1. Introduction

Performance-intensive software, such as that found in high-performance computing systems and distributed real-time and embedded systems, increasingly executes on a multitude of platforms and user contexts. To ensure that performance-intensive software meets its quality of service (QoS) requirements, it must often be fine-tuned to specific platforms/contexts by adjusting many (in some cases hundreds of) configuration options. Developers who write these types of systems must therefore try to ensure that their additions and modifications work across this large configuration space. In practice, however, time and resource constraints often force developers to assess performance on very few configurations and to extrapolate from these to the entire configuration space, which allows many performance bottlenecks and sources of QoS degradation to escape detection until systems are fielded.

To address these challenges in the context of performance-intensive software, we are developing and integrating the following techniques:

- **Distributed continuous quality assurance (DCQA) techniques**, which are designed to improve software quality and performance iteratively, opportunistically, efficiently, and continuously in multiple, geographically distributed locations [4]. In prior work, we have developed a prototype DCQA environment called *Skoll* (`www.cs.umd.edu/projects/skoll`) that provides a framework for executing QA tasks continuously across a grid of computing distributed around the world.
- **Model-based software development techniques**, which help to minimize the cost of QA activities by capturing the customizability of middleware within models and automatically generating configuration files from these higher level models [1]. In prior work, we have developed prototype model-based software tools including (1) the Options Configuration Modeling language (OCML) [5] that allows developers to model middleware configuration options as high-level models and (2) model-driven benchmarking tools [3] that allow developers to compose benchmarking experiments that observe QoS behavior by mixing and matching middleware configurations.

This paper expands our prior work by focusing on how we integrated our modeling tools with the Skoll environment to support more effective DCQA processes for performance-intensive software. In particular, we describe model-based enhancements to Skoll that enable it to rapidly identify a small subset of highly influential performance-related configuration options and systematically explore that subset of options empirically to estimate system performance across the entire configuration space. We then present results of using this DCQA process on ACE+TAO (`deuce.doc.wustl.edu/Download.html`), which are widely-used production QoS-enabled middleware frameworks. Our results show that (1) model-based DCQA tools and processes can correctly identify a key subset of options that affect system performance significantly and (2) monitoring only these selected options improves developer insight into the effect of system changes on performance at an acceptable cost.

2. Addressing QA Challenges for Performance-intensive Software Systems

This section describes key QA challenges faced by developers of performance-intensive software and describes

how DCQA environments and model-based software development techniques can help to resolve these challenges.

2.1. Challenge 1: Configuration Space Explosion in Performance-intensive Software Systems

Context. Performance-intensive software often provide fine-grained knobs to tune QoS behavior so it can be optimized for particular run-time contexts and application requirements. For example, high-performance web servers (*e.g.*, Apache), object request brokers (*e.g.*, TAO), and databases (*e.g.*, Oracle) have hundreds of options and configuration parameters. General-purpose, one-size-fits-all solutions often have unacceptable QoS for performance-intensive software systems.

Problem. To support customizations demanded by users, performance-intensive software must run on many hardware and OS platforms and typically have many options to configure the system at compile- and/or run-time. Highly configurable performance-intensive software can therefore yield an explosion of the *software configuration space*. While the flexibility of many options and configuration parameters promotes customization, it also creates many potential system configurations, each of which deserves extensive QA. As software configuration spaces increase in size and software development resources decrease, it becomes infeasible to handle all QA activities in-house since developers often lack all the hardware, OS, and compiler platforms on which their reusable software artifacts will run.

Solution approach → the Skoll DCQA environment. To address the QA challenges caused by the explosion of the software configuration space and the limitations of in-house QA processes, we have developed the **Skoll** environment to prototype and evaluate tools necessary to perform "around-the-world, around-the-clock" DCQA processes. Our feedback-driven Skoll environment includes languages for modeling system configurations and their constraints, algorithms for scheduling and remotely executing tasks, and analysis techniques for characterizing faults. Skoll divides QA processes into multiple subtasks that are intelligently and continuously distributed to, and executed by, a grid of computing resources contributed by end-users and distributed development teams around the world. The results of these executions are returned to central collection sites where they are fused together to identify defects and guide subsequent iterations of the DCQA process.

2.2. Challenge 2: Evaluating the QoS of Performance-intensive Software Systems

Context. Performance-intensive software systems run on a multitude of hardware/OS/compiler platforms and provide fine grained knobs to tune QoS behavior.

Problem. To evaluate key QoS characteristics of performance-intensive software, QA engineers today often handcraft individual QA tasks (*e.g.*, benchmarking experiments) by writing (1) interface definitions, (2) component implementations, (3) client test applications, and (4) scaffolding code. Manually implementing these steps is tedious and error-prone since each step may be repeated many times for every QA experiment. Further, in a handcrafted approach, QA engineers visualize experiments via application source code, which provides an excessively low level of abstraction.

Solution approach → the Benchmark Generation Modeling Language (BGML). BGML [3] is a model-driven benchmarking tool that allows component middleware QA engineers to (1) visually model interaction scenarios between configuration options and system components using domain-specific building blocks, *i.e.*, capture software variability in higher-level models rather than in lower-level source code, (2) automate benchmarking code generation and reuse QA task code across configurations, (3) generate control scripts to distribute and execute the experiments to users around the world to monitor QoS performance behavior in a wide range of execution contexts, and (4) enable evaluation of multiple performance metrics, such as throughput, latency, jitter, and other QoS criteria.

2.3. Challenge 3: Assessing QoS of Performance-intensive Software Across Large Configuration Spaces

Context. As developers create and modify their performance-intensive software systems, they often conduct benchmarking experiments to identify when changes negatively affect performance. Due to time and resource constraints, however, these experiments are typically executed on a very small number of default configurations. While this provides some data, it leaves substantial portions of the entire configuration space unevaluated, allowing performance problems to escape detection until the software is fielded. To close this gap, developers of performance-intensive software could use the BGML modeling language together with the Skoll DCQA environment to gather a much wider sampling of performance data.

Problem. Although Skoll and BGML provide an infrastructure for performing large-scale QA, the configuration spaces of performance-intensive software systems are often so large that brute force processes are still infeasible. For example, the ACE+TAO systems have ∼500 configuration options, with over 2^{500} potential combinations. To be effective for highly configurable performance-intensive software systems, therefore, DCQA processes must generally include some type of *adaptation strategy* to efficiently navigate large configuration spaces.

Solution approach → Applying experimental design theory for configuration space reduction. As software systems change, developers often run regression tests to detect unintended functional side effects. In addition to functionality, developers of performance-intensive systems must also be wary of unintended effects on QoS. They will therefore periodically run performance benchmarking tests to detect such problems. As described in Section 1, however, QA efforts can be confounded in highly configurable systems due to the enormous configuration space. Moreover, time and resource constraints (and high change frequencies) severely limit the number of configurations that can be examined.

As a result developers get a *very* limited view of their system's QoS which means that problems not readily seen in the few tested configurations can escape detection until the systems are fielded.

To address these problems, we developed the *main effects screening* DCQA process, which is performed in the following two phases:

- **Phase 1.** We execute a large-scale, formally-designed experiment across the Skoll grid. As part of this experiment, we run benchmarks on a wide-ranging, but sparsely distributed, set of configurations. These configurations are selected using a class of experimental

designs called *screening designs* [6], which are highly economical and can reveal individual options that significantly affect performance (colloquially, these are referred to as first-order or "main" effects). These designs are economical since they are not intended to detect high-order interaction effects (*i.e.*, significant interactions between, *e.g.*, five different options). The choice of significance level at which to separate significant from non-significant options can be set by QA process engineers.

- **Phase 2.** Once we have identified the main effects, we only focus on them, effectively reducing the configuration space to just these few options. The process continues executing using only in-house resources. Each time the system changes, we exhaustively benchmark all combinations of the first-order options, while using default (or random) settings for the remaining options. Our intent is that by focusing only on the first-order options, we can greatly reduce the configuration space, while at the same time capture a much more complete picture of the system's QoS. This data is plotted and maintained on the system's build scoreboard (*e.g.*, `www.dre.vanderbilt.edu/Stats`). Since the main effects might change over time, the process can be restarted periodically to recalibrate the main effects options.

2.4. Putting It All Together

Now that we described how we addressed the QA challenges of performance-intensive software systems, we explain how we have integrated BGML with the existing Skoll prototype to support the *main effects screening* DCQA process described in Section 2.3.

1. A QA engineer defines a test configuration using BGML models. The necessary experimentation details are captured in the models, *e.g.*, the configuration options examined during main effects screening, the IDL interface exchanged between the client and the server, and the benchmark metric performed by the experiment.
2. The QA engineer then uses BGML to interpret the model. The paradigm interpreter parses the modeled CORBA middleware configuration options and generates the required configuration files to configure the underlying CORBA middleware. The BGML paradigm interpreter then generates the required benchmarking code, *i.e.*, IDL files, the required header and source files, and necessary script files to run the experiment.
3. When users register with the Skoll infrastructure they obtain the Skoll client software and configuration template.
4. Clients execute steps in the main effects screening experiment and return the result to the Skoll server, which updates its internal database. When prompted by developers, Skoll displays execution results using an on demand scoreboard. This scoreboard displays graphs and charts for QoS metrics, *e.g.*, performance graphs, latency measures and foot-print metrics.

3. Feasibility Study

This section describes a feasibility study that assesses the implementation cost and the effectiveness of the main effects screening process on a large performance-intensive software system.

Experimental process. We use the following experimental process to evaluate our approach:

Step 1: Subject Application. We used ACE v5.4 + TAO v1.4 [2] + CIAO v0.4 for this study. CIAO is a QoS-enabled implementation of CCM and it supports components, which simplifies the development of DRE applications by enabling developers to declaratively provision QoS policies end-to-end when assembling a system.

Step 2: Application Scenario. Due to recent changes made to the message queueing strategy, the developers of ACE+TAO+CIAO are concerned with measuring two performance criteria: (1) the latency for each request, and (2) total message throughput (events/second) between the ACE+TAO+CIAO client and server. For this version of ACE+TAO+CIAO, the developers identified 14 run-time options they felt affected latency and throughput. Each option is binary as shown in Table 1 and the entire configuration space is $2^{14} = 16,384$.

Step 3: BGML Tool. ACE+TAO+CIAO QA engineers used the BGML tool as described below to generate the screening experiments to quantify the behavior of latency and throughput.

• Using the BGML modeling paradigm QA engineers composed the experiment.
• In the experiment modeled, QA engineers associate the QoS characteristic (in this case roundtrip latency and throughput) that will be captured in the experiment.
• Using the experiment modeled, BGML interpreters generate the benchmarking code required to set-up, run and teardown the experiment. The files generated include, component implementation files (.h, .cpp), IDL files (.idl), Component IDL files (.cidl) and Benchmarking code (.cpp) files. The generated file is executed and QoS characteristics are measured.

Step 4: Application of the Main Effects Screening Process. In this step, our main concern is to find the first-order effects of configuration options; we are not interested in higher-order (interaction) effects. We decided to use a resolution IV screening design, which means that, among other things, that no main effects are aliased with any other main effects or with any two-factor interactions. The final screening design examines 14 factors in $2^5 = 32$ runs, which is a 2^9 fraction of the exhaustive design.

Step 5: Generating Variation for the Entire Configuration Space. For comparison purposed, we also obtained the performance variation for the entire configuration space, *i.e.*, 16,384 configurations. It took 48 hours of computer time to run all the benchmarking experiments.

Results. Across the entire configuration space, only options o2 and o10 have a significant effect on the latency. This result was surprising to ACE+TAO+CIAO developers since they thought that all 14 run-time options would contribute substantially to latency. The same result appears for latency variation and for throughput. Therefore, only options 02 and o10 show a significant effect on performance.

Looking instead at just the 32 data points from the "screening" design we see that we would draw the exact same conclusion. That is the screening design gave us the same information at a fraction of the cost. Note that the time needed to run the 32 configurations was about 6 minutes.

The second phase of the process used the information that o2 and o10 are important options to generate all possible (in this case 4) configurations for the binary options o2 and o10. Default values were assigned to the remaining options. The latency and throughput were measured for these 4 configurations.

The results of the second phase are that distributions obtained from the screening experiments are very similar to the ones obtained from the exhaustive runs and the medians,

Option Index	Option Name	Option Settings
o1	ORBReactorThreadQueue	{FIFO, LIFO}
o2	ORBClientConnectionHandler	{RW, MT}
o3	ORBReactorMaskSignals	{0, 1}
o4	ORBConnectionPurgingStrategy	{LRU, LFU}
o5	ORBConnectionCachePurgePercentage	{10, 40}
o6	ORBConnectionCacheLock	{thread, null}
o7	ORBCorbaObjectLock	{thread, null}
o8	ORBObjectKeyTableLock	{thread, null}
o9	ORBInputCDRAllocator	{thread, null}
o10	ORBConcurrency	{reactive, thread-per-connection}
o11	ORBActiveObjectMapSize	{32, 128}
o12	ORBUseridPolicyDemuxStrategy	{linear, dynamic}
o13	ORBSystemidPolicyDemuxStrategy	{linear, dynamic}
o14	ORBUniqueidPolicyReverseDemuxStrategy	{linear, dynamic}

Table 1. The Options and Their Settings

Metric	Screening	Random
latency	77%	46%
latency variance	64%	30%
throughput	75%	55%

Table 2. Range of Performance Metrics Covered by Screening and Random Design

not shown, were nearly identical. In contrast, the distributions for random configurations (4 chosen at random) were very different.

Table 2 shows the percentage of observations for each performance metric in the entire configuration space that fall into the range of the observations obtained from screening and random designs. As this table indicates, the screening design covered a large portion of the system's range of performance and covered more of the performance range than the random design did.

4. Concluding Remarks

This paper described a model-based distributed continuous quality assurance (DCQA) process called main effects screening. The process is designed to improve performance assessment across the large configuration spaces found in performance-intensive software. We quickly implemented this process using BGML, executed it on Skoll, and demonstrated its effectiveness via a feasibility study involving ACE+TAO middleware, which are two large-scale performance-intensive software frameworks consisting of well over one million lines of C++ code and regression tests contained in ∼4,500 files.

The main effects screening process leverages formally-designed screening experiments to isolate the most significant individual options in the configuration space. This screening experiment is executed by users around the world using the Skoll infrastructure. After this reduced option set has been identified it can then be examined exhaustively (using local resources) as often as desired. Our feasibility study showed that this process could automatically reduce an original set of 14 options down to 2 main effects and that the information of these main effects provides much of the information that could have been gained from exhaustive testing (even though exhaustive testing is infeasible). We will continue to explore new DCQA processes, *e.g.*, we are examining how to prioritize parts of the configuration model based on end-user usage patterns, developer priorities, or other economic justifications.

The results of the work presented in this paper have also motivated research in several new directions. We are working closely with the ACE+TAO developers to generalize Skoll's processes to cover a broader range of QA activities, in particular new end-to-end QoS measures on heterogeneous DRE systems. One immediate application is to start to refactor ACE to shrink its memory footprint and enhance its run-time performance. The DCQA process will then be used to measure ACE's footprint and QoS at every check-in across different configurations, while simultaneously ensuring correctness via Skoll's automated and intelligent regression testing environment [4].

References

[1] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE*, 91(1):145–164, Jan. 2003.

[2] A. S. Krishna, D. C. Schmidt, R. Klefstad, and A. Corsaro. Real-time CORBA Middleware. In Q. Mahmoud, editor, *Middleware for Communications*. Wiley and Sons, New York, 2003.

[3] A. S. Krishna, N. Wang, B. Natarajan, A. Gokhale, D. C. Schmidt, and G. Thaker. CCMPerf: A Benchmarking Tool for CORBA Component Model Implementations. In *Proceedings of the 10th Real-time Technology and Application Symposium (RTAS '04)*, Toronto, CA, May 2004. IEEE.

[4] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed Continuous Quality Assurance. In *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering*, Edinburgh, Scotland, May 2004. IEEE/ACM.

[5] E. Turkaye, A. Gokhale, and B. Natarajan. Addressing the Middleware Configuration Challenges using Model-based Techniques. In *Proceedings of the 42nd Annual Southeast Conference*, Huntsville, AL, Apr. 2004. ACM.

[6] C. F. J. Wu and M. Hamada. *Experiments: Planning, Analysis, and Parameter Design Optimization*. Wiley, 2000.