Main Effects Screening: A Distributed Continuous Quality Assurance Process for Monitoring Performance Degradation in Evolving 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 37235

Abstract

Developers of highly configurable performance-intensive software systems often use a type of in-house performance-oriented "regression testing" to ensure that their modifications have not adversely affected their software's performance across its large configuration space. Unfortunately, time and resource constraints often limit developers to in-house testing of a small number of configurations and unreliable extrapolation from these results to the entire configuration space, which allows many performance bottlenecks and sources of QoS degradation to escape detection until systems are fielded. To improve performance assessment of evolving systems across large configuration spaces, we have developed a distributed continuous quality assurance (DCQA) process called main effects screening that uses in-the-field resources to execute formally designed experiments to help reduce the configuration space, thereby allowing developers to perform more targeted in-house QA. We have evaluated this process via several feasibility studies on several large, widely-used performance-intensive software systems. Our results indicate that main effects screening can detect key sources of performance degradation in large-scale systems with significantly less effort than conventional techniques.

1. Introduction

The quality of performance-intensive software systems, such as high-performance scientific computing systems and distributed real-time and embedded (DRE) systems, depend heavily on their infrastructure platforms, such as the hardware, operating system, middleware, and language processing tools. Developers of such systems often need to tune the infrastructure and their software applications to accommodate the (often changing) platform environments and performance requirements. This tuning is commonly done by (re)adjusting a large set (10's-100's) of compile- and run-time configuration options that record and control variable software parameters, such as different operating systems, resource management strategies, middleware and application feature sets; compiler flags; and/or run-time optimization settings. For example, SQL Server 7.0 has 47 configuration options, Oracle 9 has 211 initialization parameters, and Apache HTTP Server Version 1.3 has 85 core configuration options.

Although the existence of all these software parameters promotes flexibility and portability, it also means the software must be tested over an enormous number of different configurations, which yields the following challenges for developers who must ensure that their decisions, additions, and modifications work across this large (and often changing) configuration space:

- Settings that maximize performance for a particular platform/context may not be suitable for different ones and certain groups of option settings may be semantically invalid due to subtle dependencies between options.
- Limited QA budgets and rapidly changing code bases mean that developers' QA efforts are often limited to just a few software configurations, forcing them to extrapolate their findings to the entire configuration space.
- The configurations that are tested are often selected in an *ad hoc* manner, so quality is not evaluated systematically and many quality problems escape detection until systems are fielded.

Since exhaustive testing of performance-intensive software is infeasible under the circumstances listed above, what developers need is a quick way to estimate how their changes and decisions affect software performance across its entire configuration space. To provide this capability, we have developed and evaluated a new hybrid (*i.e.*, partially in-the-field and partially in-house) *distributed continuous quality assurance* (DCQA) process that improves software quality iteratively, opportunistically, and efficiently by executing QA tasks continuously across a grid of computing resources provided by end-users and distributed development teams.

In prior work [15], we implemented a prototype DCQA support environment called *Skoll* that helps developers create, execute, and analyze their own DCQA processes, as described in Section 2. To make it easier to implement DCQA processes, we also integrated model-based software development tools with Skoll, which help developers capture the variant and invariant parts of DCQA processes and the software systems they are applied to within high-level models that can be processed to automatically generate configuration files and other supporting code artifacts [8]. Some model-based tools integrated with Skoll include the

Options Configuration Modeling language (OCML) [17] that models configuration options and inter-option constraints and the Benchmark Generation Modeling Language (BGML) [10] that composes benchmarking experiments to observe QoS behavior under different configurations and workloads.

This paper extends our earlier work by developing a new model-based *hybrid* DCQA process that leverages the extensive (albeit less dedicated) *in-the-field* computing resources provided by the Skoll grid, weeding out unimportant options to reduce the configuration space, thereby allowing developers to perform more targeted QA using their very limited (but dedicated) *in-house* resources. This hybrid DCQA process first runs formally designed experiments across the Skoll grid to identify a subset of important performance-related configuration options. Whenever the system changes thereafter, this process then exhaustively explores all configurations of the important options using in-house computing resources to estimate system performance across the entire configuration space. This hybrid approach is feasible because the new configuration space is much smaller than the original, and hence more tractable using in-house resources.

This paper presents an evaluation of our new hybrid DCQA process on ACE, TAO, and CIAO (`deuce.doc.wustl.edu/Download.html`), which are widely-used production quality, performance-intensive middleware frameworks. Our results indicate that (1) hybrid model-based DCQA tools and processes can correctly identify the subset of options that are important to system performance, (2) monitoring only these selected options helps to quickly detect key sources of performance degradation at an acceptable level of effort, and (3) alternative strategies with equivalent effort give less reliable results.

2. The Model-based Skoll DCQA Environment

To maintain and evaluate the quality of performance-intensive software across large configuration spaces, we are developing and evaluating *distributed continuous quality assurance* (DCQA) processes [15] that evaluate various software qualities, such as portability, performance characteristics, and functional correctness, "around-the-world, around-the-clock." To accomplish this, DCQA processes are divided into multiple subtasks, such as running regression tests on a particular system configuration, evaluating system response time under different input workloads, or measuring usage errors for a system with several alternative GUI designs. As illustrated in Figure 1, these subtasks are then intelligently and continuously distributed to – and executed by – clients across a grid of computing resources contributed largely by end-users and distributed development teams. The results of these evaluations are returned to servers at central collection sites, where they are fused together to guide subsequent iterations of the DCQA processes.

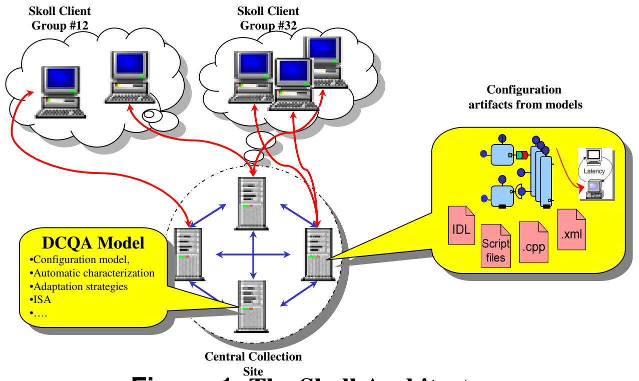

Figure 1. The Skoll Architecture

To help implement, execute, and analyze DCQA processes, we have developed *Skoll*, which is a model-based DCQA environment described at `www.cs.umd.edu/projects/skoll`. For completeness, this section describes some of Skoll's components and services, which include languages for modeling system configurations and their constraints, algorithms for scheduling and remotely executing tasks, and planning technology that analyzes subtask results and adapts the DCQA process in real time.

The cornerstone of Skoll is its formal model of a DCQA process's configuration space, which captures the different configuration options and their settings. Since in practice not all combinations of options make sense (*e.g.*, feature X may not be supported on operating system Y), we define *inter-option constraints* that limit the setting of one option based on the settings of others. A *valid configuration* is one that violates no inter-option constraints (for the feasibility study in Section 4, we used the OCML modeling tool [17] to visually define the configuration model and to generate the low-level formats used by other Skoll components). Skoll uses this configuration space model to help plan global QA processes, adapt these processes dynamically, and aid in analyzing and interpreting results from various types of functional and performance regression tests.

Since the configuration spaces of performance-intensive software can be quite large, Skoll has an *Intelligent Steering Agent* (ISA) that uses AI planning techniques to control DCQA processes by deciding which valid configuration to allocate to each incoming Skoll client request. When a client is available to perform QA activities, the ISA decides which subtask to assign it by considering many factors, including (1) *the configuration model*, which characterizes the subtasks that can legally be assigned, (2) *the results of previous subtasks*, which capture what tasks have already been done and whether the results were successful, (3) *global process goals*, such as testing popular configurations more than rarely used ones or testing recently changed features more heavily than unchanged features, and (4) *client characteristics and preferences*, *e.g.*, the selected configuration must be compatible with the OS running on the client machine or configurations must run with user-level – rather than superuser-level – protection modes.

After a valid configuration is chosen, the ISA packages the corresponding QA subtask into a *job configuration*, which consists of the code artifacts, configuration parameters, build instructions, and QA-specific code (*e.g.*, developer-supplied regression/performance tests) associated with a software project. Each job configuration is then sent to a Skoll client, which executes the job configuration and returns the results to the ISA (for the feasibility studies described in Section refexperiment, we used the BGML modeling tools [11] to generate most of the code that comprises a job configuration). The ISA can learn from the results and adapt the process, *e.g.*, if some configurations fail to work properly, developers may either want to pinpoint the source of the problems or refocus on other unexplored parts of the configuration space. To control the ISA, Skoll DCQA process designers can develop customized *adaptation strategies* that monitor the global process state, analyze it, and use the information to modify future subtask assignments in ways that improve process performance.

Since DCQA processes can be complex, Skoll users often need help to interpret and leverage process results. Skoll therefore supports a variety of pluggable analysis tools, such as Classification Tree Analysis (CTA) [1]. In previous work [15, 19], for example, we used CTA to diagnose options and settings that were the likely causes of specific test failures. For the work presented in this paper, we developed statistical tools to analyze data from the formally-designed experiments described in the following section.

3 Performance-Oriented Regression Testing

As software systems change, developers often run regression tests to detect unintended functional side effects. Developers of performance-intensive systems must also be wary of unintended side effects on their end-to-end QoS. To detect such performance problems, developers often run benchmarking regression tests periodically. As described in Section 1, however, in-house QA efforts can be confounded by the enormous configuration space of highly configurable performance intensive systems, where time and resource constraints (and often high change frequencies) severely limit the number of configurations that can be examined. For example, our earlier experience with applying Skoll to the ACE+TAO middleware [15] found that only a small number of default configurations are benchmarked routinely by the core ACE+TAO development team, who thus get a *very* limited view of their middleware's QoS. Problems not readily seen in these default configurations therefore often escape detection until systems based on ACE+TAO are fielded by end-users.

This section describes how we address this problem by using the model-based Skoll environment (Section 2) to develop and implement a new hybrid DCQA process called *main effects screening*. We also describe the formal foundations of our approach, which is based on *design of experiments theory*, and give an example that illustrates key aspects of our approach.

3.1. The Main Effects Screening Process

Main effects screening is a technique for rapidly detecting performance degradation across a large configuration space as a result of system changes. Our approach relies on a class of experimental designs called *screening designs* [18], which are highly economical and can reveal important *low order effects* (such as individual option settings and option pairs/triples) that strongly affect performance. We call these most influential option settings "main effects."

At a high level, main effects screening involves the following steps: (1) *compute* a formal experimental design based on the system's configuration model, (2) *execute* that experimental design across fielded computing resources in the Skoll DCQA grid by running and measuring benchmarks on specific configurations dictated by the experimental design devised in step 1, (3) *collect, analyze and display* the data so that developers can identify the main effects, (4) *estimate* overall performance whenever the software changes by evaluating all combinations of the main effects (while defaulting or randomizing all other options), and (5) *recalibrate* the main effects options by restarting the process periodically since the main effects can change over time, depending on how fast the system changes.

The assumption behind this five step process is that since main effects options are the ones that affect performance most, evaluating all combinations of these option settings (which we call the "screening suite") can reasonably estimate performance across the entire configuration space. If this assumption is true, testing the screening suite should provide much the same information as testing the entire configuration space, but at a fraction of the time and effort since it is much smaller than the entire configuration space.

3.2. Technical Foundations of Screening Designs

For main effects screening to work we need to identify the main effects, *i.e.*, the subset of options whose settings account for a large portion of performance variation across the system's configuration space. One obvious approach is to test every configuration exhaustively. Since exhaustive testing is infeasible for large-scale, highly configurable performance-intensive software systems, however, developers often do some kind of random or *ad hoc* sampling based on their knowledge of the system. Since our experience indicates that these approaches can be unreliable [15, 10], we need an approach that samples the configuration space, yet produces reasonably precise and reliable estimates of overall performance.

The approach we chose for this paper uses formally-designed experiments, called *screening designs*, that are highly economical and whose primary purpose is to identify important low-order effects, *i.e.*, first-, second-, or third-order effects, where an n^{th}-order effect is an effect caused by the simultaneous interaction of n factors. For instance, for certain web server applications, a 1^{st}-order effect might be that performance slows considerably when logging is turned on and another might be that it also slows when few server threads are used. A 2^{nd} order effect involves the interaction of two options, *e.g.*, web server performance may slow when caching is turned off *and* the server performs blocking reads.

In the work presented in this paper, we compute specific screening designs by extending fractional factorial designs. A *full* factorial design implies exhaustively testing the configuration space, whereas *fractional* factorial designs are factorial designs that require a specific fraction (such as $1/2$ or $1/4$) of full factorial designs. As a result, fractional factorial designs are less costly than full factorial designs, but lose the ability to measure some higher-order effects. Since their run size (*i.e.*, number of observations required) grows exponentially in the number of configuration options, however, they can still grow too costly for systems with many options.

Screening designs reduce run sizes even more and at their smallest can require roughly the same number of observations as the number of effects one wishes to calculate (experimenters will often use more than the minimum number of observations to improve precision or to deal with noisy processes). The reductions come from aliasing the effects of lower-order interactions with higher-order ones, *i.e.*, by making it impossible to distinguish between certain high- and low-order effects. While this may seem problematic, screening designs have been used extensively to understand and improve products and processes developed in manufacturing, engineering, and physical sciences. Their success stems largely from the ability to use them in an iterative, "quick and dirty" fashion, *i.e.*, to focus on major problem sources, a few at a time, rather than trying to understand and fix all problems simultaneously. Since our objective with main effects screening is also to produce a rough – but reliable – estimate of overall performance, we hypothesize that screening designs provide the appropriate foundation for our hybrid model-based DCQA processes.

3.3. Screening Designs in Action

To show how screening designs are computed, we now present a hypothetical example of a performance-intensive software system with 4 binary configuration options, A through D, with no inter-option constraints. The full configuration space therefore has $2^4 = 16$ configurations. To create a screening design, developers must decide how many observations they can afford, which level of effects they want to analyze, and how they will alias effects to fill out the design. In our example, developers decide they can afford to evaluate 8 configurations and that they will focus only on 1^{st}-order effects.

Given this information, we begin by creating a 2^3 full factorial design for options A, B, and C because this design has the maximum number of configurations we wanted to observe. This design is shown in Table 1(a), where the binary option settings are encoded as (−) or (+). This design is referred to as a 2^{4-1} design, where 4 refers to the total number of options we will examine and the -1 ($2^{-1} = 1/2$) indicates the fraction of the full factorial over which we will collect data.

A	B	C
-	-	-
+	-	-
-	+	-
+	+	-
-	-	+
+	-	+
-	+	+
+	+	+

(a)

A	B	C	D
-	-	-	-
+	-	-	+
-	+	-	+
+	+	-	-
-	-	+	+
+	-	+	-
-	+	+	-
+	+	+	+

(b)

Table 1. (a) 2^3 Design and (b) 2^{4-1}_{IV} Design

This design has $2^3 - 1 = 7$ degrees of freedom. We use 3 degrees of freedom to estimate the effects of A, B, and C. The remaining degrees of freedom would normally be used to estimate higher-order effects of these options, but since we are only interested in the 1^{st} order effects, we can instead use them to estimate the effect of option D, *i.e.*, we can extend the design and estimate the effect of option D without going to a 2^4 full factorial design.

The final remaining issue is selecting the settings for option D. We do this using a *design generator*, which specifies the aliasing patterns used to build the design. For this example, we select the design generator $D = ABC$, which means that D's settings are computed by multiplying the settings for options A, B, and C (think of + as 1 and − as -1).

The design we described above is a *resolution IV* design. In resolution R designs, no effects involving i factors are aliased with effects involving less than $R - i$ factors. Developers therefore need to choose what order of effects they wish to observe. Table 1(b) gives the final design, which is identified uniquely as a 2^{4-1}_{IV} design with the design generator $D = ABC$.

After defining the screening design, we can execute it across the Skoll grid. For our process, each observation involves measuring a developer-supplied benchmarking regression test while the system runs in a particular configuration. We would next analyze the data to calculate the effects. For binary options (with settings − or +), the main effect of option A, ME(A), is

$$ME(A) = z(A-) - z(A+) \tag{1}$$

where z(A-) and z(A+) are the mean values of the observed data over all runs where option A is (−) and where option A is (+), respectively.

If desired, 2^{nd} order effects can be calculated in a similar way. The interaction effect of option A and B, INT(A, B) is:

$$INT(A, B) = 1/2\{ME(B|A+) - ME(B|A-)\} \quad (2)$$
$$= 1/2\{ME(A|B+) - ME(A|B-)\} \quad (3)$$

Here $ME(B|A+)$ is called the conditional main effect of B at the + level of A. The effect of one factor (*e.g.*, B) therefore depends on the level of the other factor (*e.g.*, A). Similar equations exist for higher order effects.

As shown in Section 4, once the effects are computed we display them graphically using Skoll visualization tools [15], allowing developers to decide which effects they consider important.

4. Feasibility Study

This section describes a feasibility study that assesses the implementation cost and the effectiveness of the main effects screening process described in Section 3 on a suite of large, performance-intensive software systems.

4.1. Experimental Design

Hypotheses. Our feasibility study explores the following hypotheses: (1) our model-based Skoll environment cost-effectively supports the definition, implementation and execution of our main effects screening process described in Section 3, (2) the screening design used in main effects screening correctly identifies a small subset of options whose effect on performance is important, and (3) exhaustively examining just the options identified by the screening design gives performance data that (a) is representative of the system's performance across the entire configuration space, but less costly to obtain and (b) is more representative than a similarly-sized random sample.

Subject applications. The experimental subject applications for this study were based on a suite of performance-intensive software: ACE v5.4 + TAO v1.4 + CIAO v0.4. ACE provides reusable C++ wrapper facades and framework components that implements core concurrency and distribution patterns [16] for communication software. TAO is a high-performance, highly configurable Real-time CORBA ORB built atop ACE to meet the demanding QoS requirements of DRE systems. CIAO is QoS-enabled middleware that extends TAO to support components, which enables developers to declaratively provision QoS policies end-to-end when assembling a DRE system.

ACE, TAO, and CIAO are ideal subjects for our feasibility study since they share many characteristics with other highly configurable performance-intensive software systems. For example, they collectively contain over 2M+ lines of source code, functional regression tests, and performance benchmarks contained in ∼4,500 files that average over 300 CVS commits per week. They also run on a wide range of OS platforms, including all variants of Windows, most versions of UNIX, and many real-time operating systems, such as LynxOS and VxWorks.

Application scenario. Due to recent changes made to the message queuing strategy, the developers of ACE+TAO+-CIAO were 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 binary run-time options they felt affected latency and throughput (See Table 2). Thus, the entire configuration space has $2^{14} = 16,384$ different configurations.

Option Index	Option Name	Option Settings
A	ORBReactorThreadQueue	{FIFO, LIFO}
B	ORBClientConnectionHandler	{RW, MT}
C	ORBReactorMaskSignals	{0, 1}
D	ORBConnectionPurgingStrategy	{LRU, LFU}
E	ORBConnectionCachePurgePercentage	{10, 40}
F	ORBConnectionCacheLock	{thread, null}
G	ORBCorbaObjectLock	{thread, null}
H	ORBObjectKeyTableLock	{thread, null}
I	ORBInputCDRAllocator	{thread, null}
J	ORBConcurrency	{reactive, thread-per-connection}
K	ORBActiveObjectMapSize	{32, 128}
L	ORBUseridPolicyDemuxStrategy	{linear, dynamic}
M	ORBSystemidPolicyDemuxStrategy	{linear, dynamic}
N	ORBUniqueidPolicyReverseDemuxStrategy	{linear, dynamic}

Table 2. Some ACE+TAO Options and their Settings

Experimental process. After selecting the ACE+TAO+-CIAO subject application and our application scenario, our experimental process used Skoll's model-driven tools to implement the main effects screening process and evaluate our three hypotheses. To accomplish this, we executed the main effects screening process across a prototype Skoll grid of dual processor Xeon machines running Red Hat 2.4.21 with 1GB of memory in the real-time scheduling class. The experimental task involved running a benchmark application in a particular configuration, which evaluated the application scenario outlined above by creating an ACE+TAO+-CIAO client and server. For each task we measured message latency and overall throughput between the client and the server. The client sends 300K requests to the server, where after each request it waits for a response from the server and records the latency measure. At the end of 300K requests, the client computes the throughput achieved in terms of number of requests served per second. We finally analyzed the resulting data to evaluate our hypotheses. Section 6 describes the limitations with our current experimental process.

4.2. The Full Data Set

To evaluate main effects screening, we first generated performance data for all 16,000+ valid configurations, which we refer to as the "full suite" and the performance

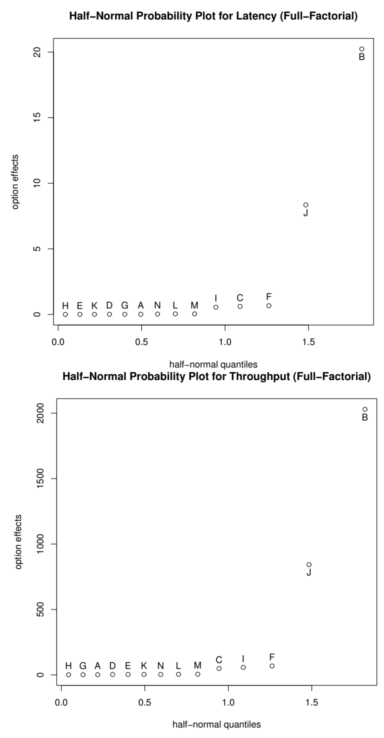

Figure 2. Option effects based on full data

data as the "full data set." We then examined the effect of each option and judged whether they had important effects on performance using a graphical method called *half-normal probability plots*, which show each option's effect against their corresponding coordinates on the half-normal probability scale. If $|\theta|_1 \leq |\theta|_2 \leq ... \leq |\theta|_I$ are the ordered set of effect estimations, the half-normal plot then consists of the points

$$(\Phi^{-1}(0.5 + 0.5[i - 0.5]/I), |\theta|_i) \ for \ i = 1, ..., I \qquad (4)$$

where Φ is the cumulative distribution function of a standard normal random variable.

The rationale behind half-normal plots is that unimportant options will have effects whose distribution is normal and centered near 0. Important effects will also be normally distributed[1] with means different that 0. If no effects are important, the resulting plot will show a set of points on a rough line near $y = 0$. Options whose effects deviate substantially from 0 are considered important.

Note that "importance" is not defined formally and differs in spirit from the traditional notion of statistical significance. In particular, developers must decide for themselves how large effects must be to warrant their attention. While this has some downsides (see Section 6), even with traditional statistical tests that measure statistical significance developers still must make judgments as to the magnitude of effects.

[1]Since the effects are averages over numerous observations, the central limit theorem guarantees normality.

Figure 2 plots the effect of each of the 14 ACE+TA+-CIAO options on latency and throughput respectively. The effects are calculated from the full data set. We see that options B and J are clearly important, whereas options I, C and F are arguably important, and the remaining options are not important.

4.3. Evaluating Screening Designs

We now evaluate whether the remotely executed screening designs can correctly identify the important options discovered during our analysis of the full data set. To accomplish this, we calculated and executed three different screening designs, whose specifications appear in Appendix A. These designs examined all 14 options using increasingly larger run sizes (32, 64, or 128 observations). We refer to the screening designs as Scr_{32}, Scr_{64} and Scr_{128}, respectively.

Figure 3 shows the half-normal probability plots obtained from our screening designs. The figures show that all screening designs correctly identify options B and J as being important (as is the case in full-factorial experiment). Scr_{128} also identifies the possibly important effect of options C, I, and F. Due to space considerations in the paper we only present data on latency. Throughput analysis shows identical results unless otherwise stated.

These results suggest that (1) screening designs can detect important options at a large fraction of the cost of exhaustive testing, (2) the smaller the effect, the larger the run size needed to identify it, and (3) developers should be cautious when dealing with options that appear to have an important, but relatively small effect, as they may actually be seeing normal variation (Scr_{32} and Scr_{64} both have examples of this).

4.4. Estimating Performance with Screening Suites

We now evaluate whether screening all the combinations of the most important options can be used to estimate performance quickly across the entire configuration space we are studying. The estimates are generated by examining all combinations of the most important options, while defaulting the settings of the unimportant options (developers could choose to randomize the settings of unimportant options, as well). In the previous section, we determined that options B and J were clearly important and that options C, I, and F were arguably important. Developers will therefore make the estimates based on benchmarking either 4 (all combinations of options B and J) or 32 (all combinations of options B, J, C, I, and F) configurations. We will refer to the set of 4 configurations as the top-2 screening suite and the set of 32 configurations as the top-5 screening suite.

Figure 4 shows the distributions of latency for the full suite vs the top-5 screening suite and for the full suite vs the top-2 screening suite. From the figure, we see that the distributions of the top-5 and top-2 screening suites closely track

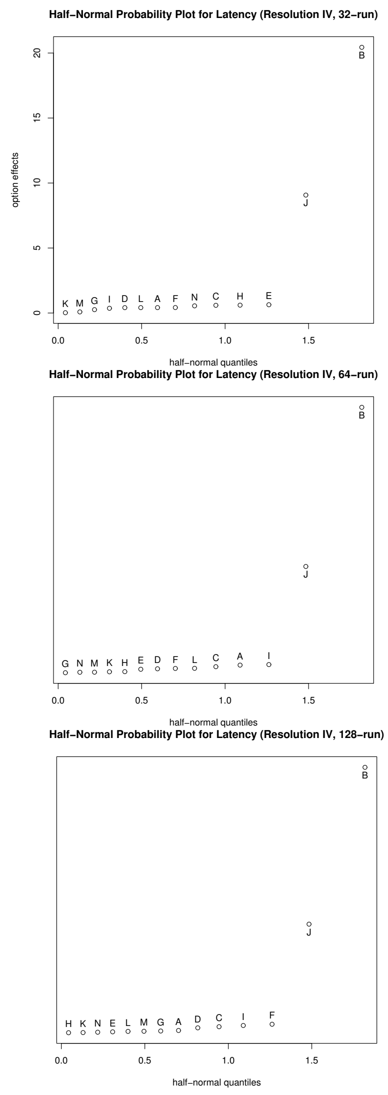

Figure 3. Option Effects Based on Screening Designs

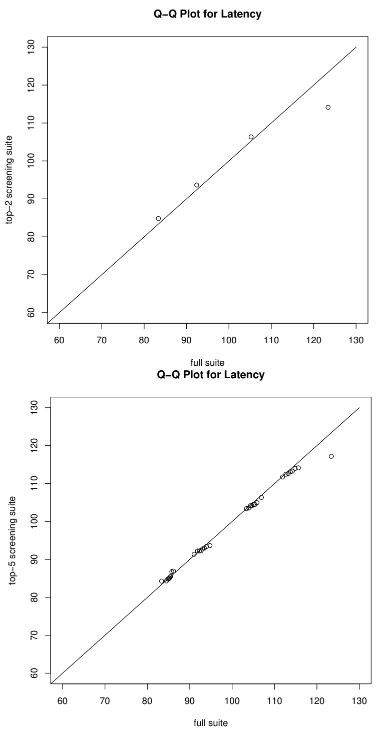

Figure 4. Q-Q plots for the top-2 and top-5 screening suites

the overall performance data. Such plots, called quantile-quantile (Q-Q) plots, are used to see how well two data distributions correlate. To do this they plot the quantiles of the first data set against the quantiles of the second data set. If the two sets share the same distribution, the points should fall approximately on $x = y$ line.

This data suggests that the screening suites computed at step 4 of the main effects screening process (Section 3) can be used to estimate overall performance in-house at extremely low time/effort, *i.e.*, running 4 benchmarks takes 40 seconds, running 32 takes 5 minutes, running 16,000+ takes 2 days.

4.5. Screening Suites vs. Random Sampling

Another question is whether our main effects screening process was any better that other obvious low-cost estimation processes. In particular, we compared latency distributions of several random samples of 4 configurations to that of the top-2 screening suite found by our process. The results of this test are summarized in Figure 5. These box plots show the distributions of latency metric obtained from exhaustive testing, top-2 screening suite testing, and random testing. These graphs suggest the obvious weakness of random sampling, *i.e.*, while sampling distributions tend toward the overall distribution as the sample size grows, individual small samples may show wildly different distributions.

4.6. Dealing with Evolving Systems

A primary goal of main effects screening is to detect performance degradations in evolving systems quickly. Our experiments discussed above do not address whether – or for how long – screening suites remain useful as a system evolves. To better understand this issue, we measured latency on the top-2 screening suite, once a day, using CVS snapshots of ACE+TAO+CIAO. We used historical snapshots for two reasons: (1) the versions are from the time period for which we already calculated the main effects and (2) developer testing and in-the-field usage data have already been collected and analyzed for this time period (see `www.dre.vanderbilt.edu/Stats/`), allowing us to reasonably assess the system's performance characteristics without having to exhaustively test all configurations

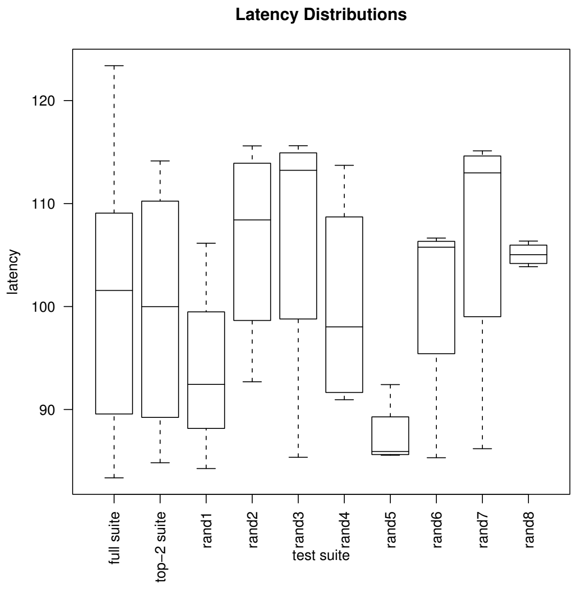

Figure 5. **Latency Distribution from full, top-2, and random suites**

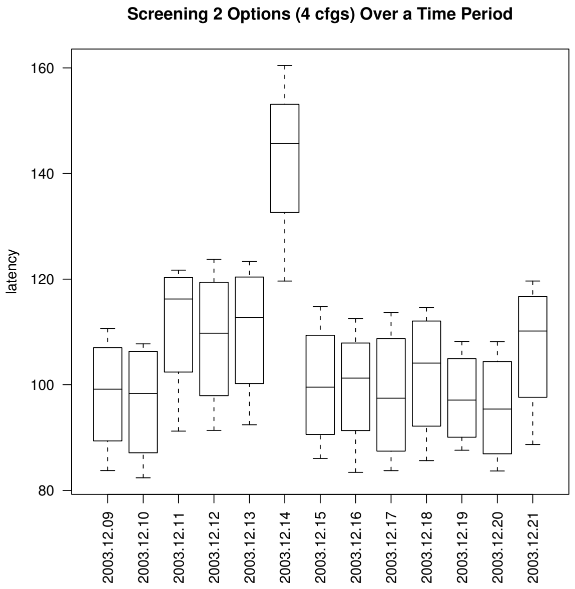

Figure 6. **Performance estimates across time**

for each system change.

Figure 6 depicts the data distributions for the top-2 screening suites broken down by date (higher latency measures are worse). We see that the distributions were stable the first two days, crept up somewhat for 3 days and then shot up the 4^{th} day (12/14/03). They were brought back under control for several more days, but then moved up again on the last day. Developer records and problem reports indicate that problems were noticed on 12/14/03, but not before then.

Another interesting finding was that the limited testing done by ACE+TAO+CIAO developers measured a performance drop of only around 5% on 12/14/03. In contrast, our screening process showed a much more dramatic drop – closer to 50%. Further analysis by system developers indicated that their unsystematic testing failed to evaluate configurations where the degradation was much more pronounced.

4.7. Higher-Order Effects

The analyses done so far only calculated first-order effects, which worked well for our subject application and scenario, but might not be sufficient for other situations. Figure 7 shows the effects of all pairs of options in the subject systems based on the full data set and on a screening design. We used a resolution VI design here (rather than resolution IV as in the previous sections) and increased the run size to 2,048 to capture the second-order effects.

Figure 7 shows several things. First, the important interaction effects involve only options that are already considered important by themselves, which supports the idea that monitoring only first-order effects was sufficient for our subject systems. Second, we see that the screening design correctly identifies the 5 most important pairwise interactions at $1/8^{th}$ the cost of exhaustive testing.

5. Related Work

Applying DOE to software engineering. As far as we can tell, no one has used screening designs for software performance assessment. The use of design of experiment (DoE) theory within software engineering has mostly been limited to *interaction testing*. The goal of interaction testing is largely to compute and sometimes generate minimal test suites that cover all combinations of specified program inputs, typically by computing orthogonal arrays or covering arrays. Some examples of this work include Dalal *et al.* [5], Burr *et al.* [3], Dunietz *et al.* [6], and Kuhn *et al* [12]. Yilmaz *et al.* [19] used covering arrays as a configuration space sampling technique to support the characterization of failure-inducing option settings.

Other relevant literature on performance monitoring includes:

• **Offline analysis**, which has been applied to program analysis to improve compiler-generated code. For example, the ATLAS [7] numerical algebra library uses an empirical optimization engine to decide the values of optimization parameters by generating different program versions that are run on various hardware/OS platforms. The output from these runs are used to select parameter values that provide the best performance. Mathematical models are also used to estimate optimization parameters based on the underlying architecture, though empirical data is not fed into the models to refine it.

• **Online analysis**, where feedback control is used to dynamically adapt QoS measures. An example of online analysis is the ControlWare middleware [20], which uses

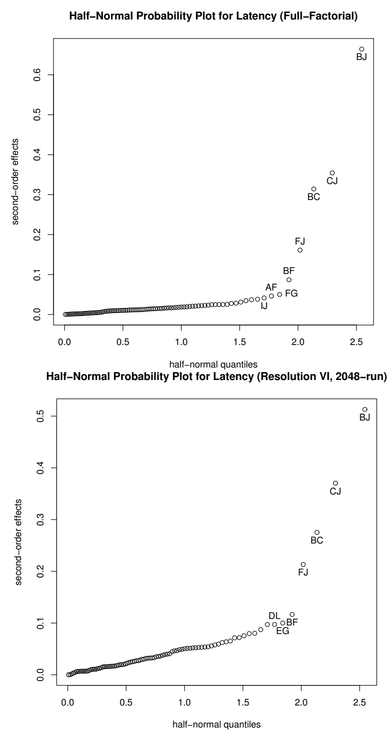

Figure 7. Pairwise Effects Based on Full and Screening Suite

feedback control theory by analyzing the architecture and modeling it as a feedback control loop. Actuators and sensors then monitor the system and affect server resource allocation. Real-time scheduling based on feedback loops has also been applied to Real-time CORBA middleware [13] to automatically adjust the rate of remote operation invocation transparently to an application.

• **Hybrid analysis**, which combines aspects of offline and online analysis. For example, the continuous compilation strategy [4] constantly monitors and improves application code using code optimization techniques. These optimizations are applied in four phases, including (1) *static analysis*, in which information from training runs is used to estimate and predict optimization plans, (2) *dynamic optimization*, in which monitors apply code transformations at run-time to adapt program behavior, (3) *offline adaptation*, in which optimization plans are actually improved using actual execution, and (4) *recompilation*, where the optimization plans are regenerated.

6 Concluding Remarks

This paper presents a new distributed continuous quality assurance (DCQA) process called *main effects screening* that is designed to detect performance degradation efficiently in performance-intensive software systems that have large configuration spaces. To evaluate this process, we conducted a formally-designed experiment across a grid of in-house and in-the-field computers in the Skoll environment. The results of this experiment formed the basis for estimating the performance across the large configuration space of ACE, TAO, and CIAO, which are widely used, large-scale, performanceintensive software systems.

All empirical studies suffer from threats to their internal and external validity. For the work presented here, we were primarily concerned with threats to external validity since they limit our ability to generalize the results of our experiment to industrial practice. For instance, potential threat is that several steps in our hybrid in-the-field/in-house DCQA process requires human decision making and input, *e.g.*, developers must provide reasonable benchmarking applications and must also decide for themselves at what point they will consider an effect to be important. Bad choices in these stages make it hard to use main effects screening successfully.

Another possible threat to external validity concerns the representativeness of the ACE+TAO+CIAO subject applications, which though large are still just one suite of software systems. A related issue is that we have focused on a relatively simple and small subset of the entire configuration space of ACE+TAO+CIAO that only has binary options and has no inter-option constraints. While these issues pose no theoretical problems (since screening designs can be created for much more complex situations, as discussed in Section 3), we need to apply our approach to larger, more realistic configuration spaces in future work to understand how well it scales.

Another potential threat is that for the time period we studied, the ACE+TAO+CIAO subject application was in a fairly stable phase, *i.e.*, changes were made mostly to fix bugs and reduce memory footprint, but the system's functionality was relatively stable. For situations where a system's basic functionality is in greater flux, it may be harder to distinguish significant performance degradation from normal variation. Likewise, we conducted the study on a homogeneous grid of computers, which may limit the amount of noise in our experimental measurements, making it easier to distinguish performance deviations than it might be in another, more heterogeneous context.

Even with various threats to validity, however, we believe our feasibility study supports the basic hypotheses underlying our research. We reached this conclusion by noting that our studies showed that: (1) screening designs can correctly identify important options, (2) these options can be used to quickly produce reliable estimates of performance across the entire configuration space at a fraction of the cost of exhaustive testing, (3) the alternative approach of random or *ad hoc* sampling can give highly unreliable results, (4) the main effects process detected performance degradation on a large and evolving software system, and (5) the screening suite estimates were more precise than the ad hoc process currently used by the developers of the subject system. Main effects screening process can also provide a valuable defect detection aid, *e.g.*, if the screened options change unexpectedly when recalibrated, developers can reexamine the software to identify possible problems with software updates.

We believe that this line of research is novel and interesting, but much work remains to be done. We are therefore continuing to develop enhanced model-based Skoll capabilities and using them to create and validate new more sophisticated DCQA processes that overcome the limitations and threats to external validity with our current approach. In particular, we are exploring the connection between design of experiments theory and the quality assurance of systems with large configuration spaces. We are also working to incorporate Skoll services into software repositories, such as ESCHER (`www.escherinstitute.org`). Finally, we are conducting a much larger case study using Skoll to conduct the ACE+TAO+CIAO daily build and regression test process with 100+ machines contributed by users and developers worldwide.

References

[1] L. Breiman, J. Freidman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth, Monterey, CA, 1984.

[2] R. Brownlie, J. Prowse, and M. S. Padke. Robust testing of AT&T PMX/StarMAIL using OATS. AT&T Technical Journal, 71(3):41–7, 1992.

[3] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proc. of the Intl. Conf. on Software Testing Analysis & Review*, 1998.

[4] B. Childers, J. Davidson, and M. Soffa. Continuous Compilation: A New Approach to Aggressive and Adaptive Code Transformation. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Apr. 2003.

[5] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proc. of the Intl. Conf. on Software Engineering, (ICSE)*, pages 285–294, 1999.

[6] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. In *Proc. of the Intl. Conf. on Software Engineering, (ICSE '97)*, pages 205–215, 1997.

[7] Kamen Yotov and Xiaoming Li and Gan Ren et.al. A Comparison of Empirical and Model-driven Optimization. In *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation*, June 2003.

[8] 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.

[9] 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.

[10] A. S. Krishna, D. C. Schmidt, A. Porter, A. Memon, and D. Sevilla-Ruiz. Improving the Quality of Performance-intensive Software via Model-integrated Distributed Continuous Quality Assurance. In *Proceedings of the 8th International Conference on Software Reuse*, Madrid, Spain, July 2004. ACM/IEEE.

[11] 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.

[12] D. Kuhn and M. Reilly. An investigation of the applicability of design of experiments to software testing. *Proc. 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, pages 91–95, 2002.

[13] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms. *Real-Time Systems Journal*, 23(1/2):85–126, July 2002.

[14] R. Mandl. Orthogonal Latin squares: an application of experiment design to compiler testing. *Communications of the ACM*, 28(10):1054–1058, 1985.

[15] 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.

[16] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[17] E. Turkay, 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.

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

[19] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterizations in complex configuration space. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[20] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic. Controlware: A Middleware Architecture for Feedback Control of Software Performance. In *Proceedings of the International Conference on Distributed Systems 2002*, July 2002.

A. Actual Screening Designs

The screening designs used in Section 4.3 were calculated using the SAS statistical package. (`www.sas.com`).

Scr_32 is a 2^{14-9}_{IV} with design generators $F = ABC$, $G = ABD$, $H = ACD$, $I = BCD$, $J = ABE$, $K = ACE$, $L = BCE$, $M = ADE$, $N = BDE$.

Scr_64 is a 2^{14-8}_{IV} with design generators $G = ABC$, $H = ABD$, $I = ABE$, $J = ACDE$, $K = ABF$, $L = ACDF$, $M = ACEF$, $N = ADEF$.

Scr_128 is a 2^{14-7}_{IV} with design generators $H = ABC$, $I = ABDE$, $J = ABDF$, $K = ACEF$, $L = ACDG$, $M = ABEFG$, $N = BCDEFG$.