Reliable Effects Screening: A Distributed Continuous Quality Assurance Process for Monitoring Performance Degradation in Evolving Software Systems

Cemal Yilmaz$^{\star *}$, Adam Porter†, Arvind S. Krishna‡, Atif Memon†, Douglas C. Schmidt‡, Aniruddha Gokhale‡, Balachandran Natarajan‡

* IBM T. J. Watson Research Center, Hawthorne, NY, 10532

† 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 in-house performance-oriented "regression testing" to ensure that their modifications do not adversely affect their software's performance across its large configuration space. Unfortunately, time and resource constraints can limit in-house testing to a relatively small number of possible configurations, followed by unreliable extrapolation from these results to the entire configuration space. As a result, many performance bottlenecks escape detection until systems are fielded.

In our earlier work, we improved the situation outlined above by developing an initial quality assurance process called "main effects screening". This process (1) executes formally designed experiments to identify an appropriate subset of configurations on which to base their performance-oriented regression testing, (2) executes benchmarks on this subset whenever the software changes, and (3) provides tool support for executing these actions on in-the-field and

* Cemal Yilmaz was a graduate student at the University of Maryland, College Park, when this work was carried out.

in-house computing resources. Our initial process had several limitations, however, since it was manually configured (which was tedious and error-prone) and relied on strong and untested assumptions for its accuracy (which made its use unacceptably risky in practice).

This paper presents a new quality assurance process called "reliable effects screening" that provides three significant improvements to our earlier work. First, it allows developers to economically verify key assumptions during process execution. Second, it integrated several model-driven engineering tools to make process configuration and execution much easier and less error prone. Third, we evaluated this process via several feasibility studies of three large, widely-used performance-intensive software frameworks. Our results indicate that reliable effects screening can detect performance degradation in large-scale systems more reliably and with significantly less resources than conventional techniques.

Index Terms

Distributed continuous quality assurance, performance-oriented regression testing, design-of-experiments theory

I. Introduction

The quality of service (QoS) of many performance-intensive systems, such as scientific computing systems and distributed real-time and embedded (DRE) systems, depend heavily on various environmental factors. Example dependencies include the specific hardware and operating system on which systems run, installed versions of middleware and system library implementations, available language processing tools, specific software features that are enabled/disabled for a given customer, and dynamic workload characteristics. Many of these dependencies are not known until deployment and some change frequently during a system's lifetime.

To accommodate these dependencies, users often need to tune infrastructure and software applications by (re)adjusting many (*i.e.*, dozens to hundreds) of compile- and run-time configuration options that record and control variable software parameters. These options are exposed at multiple system layers, including compiler flags and operating system, middleware, and application feature sets and run-time optimization settings. For example, there are $\sim$50 configuration options for SQL Server 7.0, $\sim$200 initialization parameters for Oracle 9, and $\sim$90 core configuration options for Apache HTTP Server Version 1.3.

Although designing performance-intensive systems to include such configurations options promotes code reuse, enhnaces portability, and helps end users improve their QoS, it also yields an enormous family of "instantiated" systems, each of which might behave differently and thus may need quality assurance (QA). The size of these system families creates serious and often under-appreciated challenges for software developers, who must ensure that their decisions, additions, and modifications work across this large (and often dynamically changing) space. For example, consider that:

- Option settings that maximize performance for a particular environment may be ill-suited for different ones. Failures can and do manifest in some configurations, but not in others. Similarly, individual code changes can have different runtime effects in different configurations.
- Individual developers, especially those in smaller companies or in open-source projects, may not have access to the full range of hardware platforms, operating systems, middleware, class library versions, etc. over which the system must run. In these situations, individual QA efforts will necessarily be incomplete.
- Limited budgets, aggressive schedules, and rapidly changing code bases mean that QA efforts frequently focus on a relatively small number of system configurations, often chosen in an *ad hoc* fashion. Developers then unreliably extrapolate from this data to the entire configuration space, which allows quality problems to escape detection until systems are fielded.

In summary, resources constraints and the large number of possible system configurations make exhaustive evalution infeasible for performance-intensive systems. Developers therefore need (1) ways to identify a small core set of configurations whose QA results can be reliably generalized across all configurations and (2) support for executing QA activities across a sufficiently rich and diverse set of computing platforms. To address these needs, our research [33], [32], [20], [31] has focused on system support and algorithms for *distributed continuous quality assurance* (DCQA) processes. DCQA helps improve software quality iteratively, opportunistically, and efficiently by executing QA tasks continuously across a grid of computing resources provided by end-users and developer communities.

In prior work [20], we created a prototype DCQA support environment called *Skoll* that helps developers create, execute, and analyze their own DCQA processes, as described in Section II. To facilitate DCQA process development and validation, we also developed *model-driven engineering* tools for use with Skoll. We then used Skoll to design and execute an initial DCQA process, called "main effects screening" [33], whose goal was to estimate performance efficiently across all system configurations (hereafter called the "configuration space").

Main effects screening borrows ideas from statistical quality improvement techniques that have been applied widely in engineering and manufacturing, such as Exploratory Data Analysis [27], Robust Parameter Design [30], and Statistical Quality Control [23]. A central activity of these techniques is to identify aspects of a system or process that contribute substantially to outcome variation. We use similar ideas to identify important configuration options whose settings define the distribution of performance across all configurations by causing the majority of performance variation. Evaluating all combinations of these important options (and randomizing the other options), thus provides an inexpensive, but reliable estimate of performance across the entire configuration space.

Although our initial work on main effects screening presented in [33] showed promise, it also had several limitations. For example, the definition and execution of the process had many manual steps. To improve this, we have extended and better integrated several model-driven engineering (MDE) [25] tools, including the *Options Configuration Modeling Language (OCML)* [28], which models configuration options and inter-option constraints, and the *Benchmark Generation Modeling Language* (BGML) [13], which models the QA tasks that observe and measure QoS behavior under different configurations and workloads. These MDE tools precisely capture common and variable parts of DCQA processes and the software systems to which they are applied. They also help reduce development and QA effort by generating configuration files and many other supporting code artifacts [2] needed to manage and control process execution across heterogeneous computing resources.

Another limitation with our initial main effects screening process was its dependence on strong and untested assumptions regarding the absence of interactions among certain groups of options. If these assumptions do not hold in practice, our results could be wildly incorrect. Moreover, we had no way to assess the validity of the assumptions without resorting to exhaustive testing, whose avoidance motivated our DCQA process in the first place.

To remedy these problems, this paper describes further enhancements to our earlier work that significantly broadens its applicability with little additional operating costs. Our new DCQA process, called "reliable effects screening" is implemented using Skoll and its MDE tools, and relies on design-of-experiments (DoE) techniques called "screening designs" [30] and "D-optimal designs" [22]. Our reliable effects screening process first identifies a small subset of the most important performance-related configuration options by creating formally-designed experiments and executing them across the Skoll grid. Whenever software changes occur thereafter, reliable effects screening then uses a far smaller amount of computing resources to estimate system performance across the entire configuration space by exhaustively exploring all combinations of the important options, while randomizing the rest. This subsequent analysis can even be run completely in-house, assuming appropriate computing platforms are available, since the reduced configuration space is *much* smaller than the original, and thus more tractable using only in-house resources.

In addition to describing our new reliable effect screening DCQA process, this paper also evaluates this process empirically on ACE, TAO, and CIAO (`dre.vanderbilt.edu`), which are three widely-used, production-quality, performance-intensive software frameworks. This evaluation indicates that (1) our reliable effects screening process can correctly and reliably identify the subset of options that are most important to system performance, (2) monitoring only these selected options helps to detect performance degradation quickly with an acceptable level of effort, and (3) alternative strategies with equivalent effort yield less reliable results. These results support our contention that reliable effects screening can cheaply and quickly alert performance-intensive system developers to changes that degrade QoS, as well as provide them with much greater insight into their software's performance characteristics.

The remainder of this paper is organized as follows: Section II summarizes the Skoll DCQA environment; Section III describes how we extended Skoll to implement the new reliable effect screening DCQA process to conduct performance-oriented regression testing efficiently; Section IV presents the design and results of a feasibility study using ACE, TAO, and CIAO; Section VI presents guidelines on how to use reliable effect screening; Section VII compares our research on reliable effect screening with related work; and Section VIII evaluates threats to the validity of our experiments and outlines future directions of our DCQA process and tool research.

II. An Overview of the Skoll DCQA Environment

To improve the quality of performance-intensive systems across large configuration spaces, our work focuses on *distributed continuous quality assurance* (DCQA) processes [20] that evaluate various software qualities, such as portability, performance characteristics, and functional correctness "around-the-world and around-the-clock."[1] To support this methodology, we developed *Skoll*, which is a *model-driven engineering* (MDE)-based DCQA environment (`www.cs.umd.edu/projects/skoll`). Skoll divides QA processes into multiple *tasks*, each of which is implemented as a generic process parametrized by one of several alternative configurations expressed via MDE tools. Example tasks might include running regression tests in one of many system configurations, evaluating system response time under one of several different input workloads, or measuring code execution coverage using one of several instrumentation schemes. As shown in Figure 1, these tasks are then intelligently and continuously distributed to – and

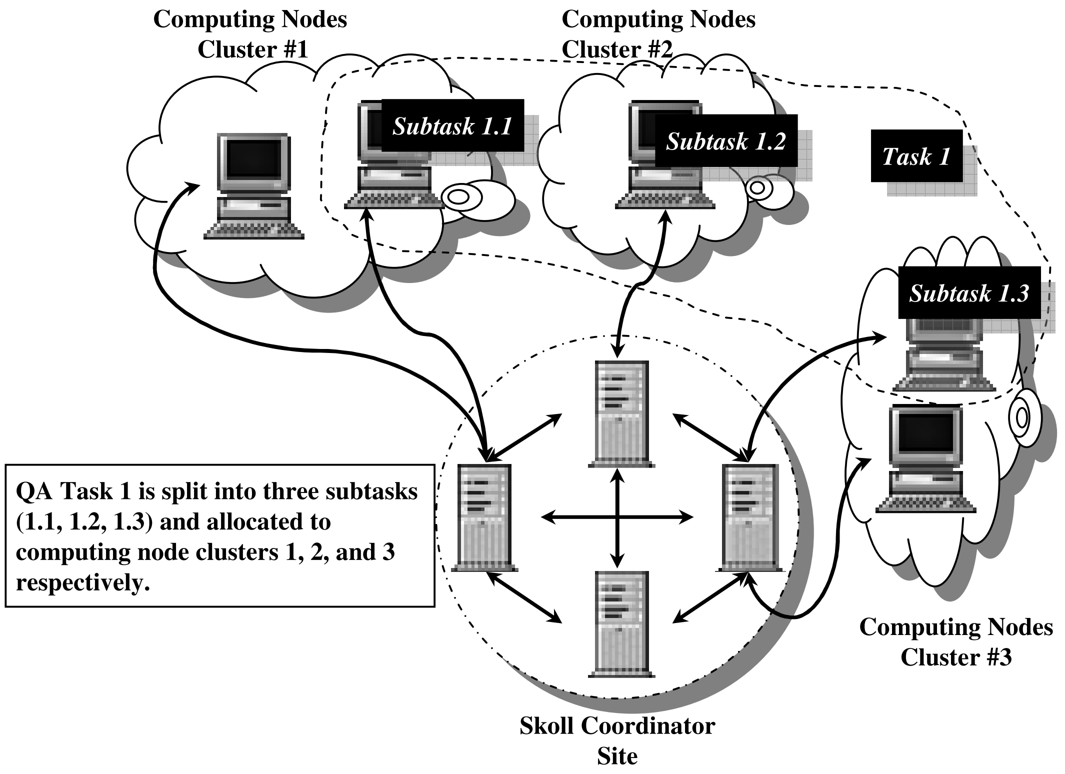

Computing Nodes Cluster #1
Computing Nodes Cluster #2
Subtask 1.1
Subtask 1.2
Task 1
Subtask 1.3
QA Task 1 is split into three subtasks (1.1, 1.2, 1.3) and allocated to computing node clusters 1, 2, and 3 respectively.
Computing Nodes Cluster #3
Skoll Coordinator Site

Fig. 1. The Skoll Distributed Continuous Quality Assurance Architecture

[1]Naturally, DCQA processes can also be executed effectively in more constrained and smaller-scale environments, such as company-wide intranets.

executed by – clients across a grid of computing resources contributed by end-user and developer communities. The results of individual tasks are returned to servers at central collection sites, where they are merged and analyzed to steer subsequent iterations and ultimately to complete the overall QA process.

This section summarizes Skoll's key components and services, which include MDE tools for modeling system configurations and their constraints, a *domain-specific modeling language* (DSML) to describe these models, algorithms for scheduling and remotely executing tasks via planning technology that analyzes task results and adapts the DCQA process in real time, a DSML to package the subtasks, and techniques to interpret and visualize the results.

QA task space. Performance-intensive systems, such as the ACE+TAO+CIAO QoS-enabled middleware, provide a range (*i.e.*, $\sim$500) of configuration options that can be used to tune its behavior.[2] To be effective, DCQA processes must keep track of these options, in addition to other environmental information, such as OS platform, build tools used, and desired version numbers. This information is used to parameterize generic QA tasks and aids in planning the global QA process, *e.g.*, by adapting the process dynamically and helping interpret the results.

In Skoll, tasks are generic processes parameterized by QA task options. These options capture information that is (1) varied under QA process control or (2) needed by the software to build and execute properly. These options are generally application-specific, including workload parameters, operating system, library implementations, compiler flags, or run-time optimization controls. Each option must take its value from a discrete number of settings. For example, in other work, our QA task model include a configuration option called `OperatingSystem` so Skoll can select appropriate binaries and build code for specific tasks [31].

QA task modeling. The QA task model underlies the DCQA process. Our experience [20], [13], [32] with the initial Skoll prototype taught us that building these models manually was tedious and error-prone. We therefore developed and integrated into Skoll the *Options Configuration Modeling Language (OCML)* [28]. OCML is an MDE tool that provides a DSML for modeling software configurations. For example, OCML defines a *numeric option* type for middleware options that can have numeric values, *e.g.*, cache, buffer, or thread pool sizes. OCML is built

[2]All the ACE+TAO+CIAO's configuration options are described at `www.cs.wustl.edu/~schmidt/ACE_wrappers/TAO/docs/Options.html`.

atop the *Generic Modeling Environment (GME)* [16], which provides a meta-programmable framework for creating DSMLs and generative tools via *metamodels* and *model interpreters*. For the feasibility study in Section IV, we used the OCML MDE tool to define the configuration model visually and generate the low-level formats used by other Skoll components.

Exploring the QA task space. Since the QA task spaces of many systems can be enormous, Skoll contains an *Intelligent Steering Agent* (ISA) [20] that uses AI planning techniques [21] to distribute QA tasks on available Skoll clients. When clients become available they send a message to the Skoll server. Skoll's ISA then decides which task to assign it by considering many factors, including (1) *the QA task model*, which characterizes the subtasks that can be assigned legally, (2) *the results of previous tasks*, 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 users can specify preferences that configurations must run with user-level – rather than superuser-level – protection modes.

After a valid configuration is chosen, the ISA packages the corresponding QA task 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. By default, the ISA simply stores these results.

In some experiments, however, we want to learn from incoming results. For example, when some configurations prove faulty, it makes no sense to retest them. Instead, we should refocus resources on other unexplored parts of the QA task space. When such dynamic behavior is desired, DCQA process designers develop customized *adaptation strategies* that Skoll uses to monitor the global process state, analyze it, and use the information to modify future task assignments in ways that improve process performance. One example of an adaptation strategy is the *nearest neighbor search strategy*, which allows a process to target failing QA task subspaces by preferentially testing the "neighbors" of a failing configuration, (*i.e.*, other similar configurations that differ in one configuration option value) to see if they also fail [20].

Packaging QA tasks. With the initial Skoll prototype, developers who wanted to evaluate QoS issues had to provide hand-written benchmark programs. For example, ACE+TAO+CIAO developers creating such benchmarks to measure latency and throughput for a particular workload had to write (1) the header files and source code that implement the measurements, (2) the configuration and script files that tune the underlying ORB and automate running tests and output generation, and (3) project build files (*e.g.*, makefiles) required to generate executable binaries from source code. Our initial feasibility study [20] revealed that this process was tedious and error-prone.

To address these problems, we developed the *Benchmark Generation Modeling Language (BGML)* [14], which is an MDE tool that automates key QoS evaluation concerns of QoS-enabled middleware and applications, such as (1) modeling how distributed system components interact with each other and (2) representing metrics that can be applied to specific configuration options and platforms. Middleware/application developers can use BGML to graphically model interaction scenarios of interest. BGML automates the task of writing repetitive source code to perform benchmark experiments and generates syntactically and semantically valid source and benchmarking code.

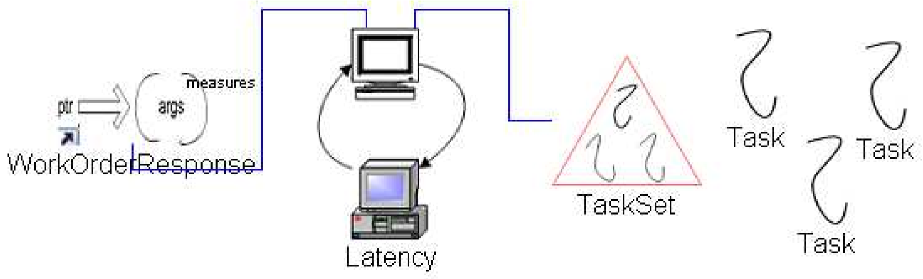


Fig. 2. Associating QoS with an Operation in BGML

Figure 2 depicts how QA engineers can visually configure an experiment that measures end-to-end latency. As shown in the figure, the latency metric was associated with an operation (`WorkOrderResponse()`) using BGML. BGML's `TaskSet` element was also used to create background tasks that invoked the `WorkOrderResponse()` operation continuously for a fixed number of iterations.

Analysis of results. Since DCQA processes can be complex, Skoll users often need help to visualize, interpret, and leverage process results. Skoll therefore supports a variety of pluggable analysis tools, such as Classification Tree Analysis (CTA) [4]. In previous work [20], [32], 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 generated by the formally-designed experiments described next.

III. Performance-Oriented Regression Testing

As software systems evolve, developers often run regression tests to detect unintended functional side effects. Developers of performance-intensive systems must also detect unintended side effects on end-to-end QoS. A common way to detect these effects is to run benchmarks when the system changes. As described in Section I, however, these efforts can be confounded for systems with many possible configurations because time and resource constraints (and often high change frequencies) severely limit the number of configurations that can be examined using only in-house resources.

For example, our earlier experiences applying Skoll to ACE+TAO [20] showed that ACE+TAO developers have a limited view of their software's QoS since they routinely benchmark only a small number of common configurations. QoS degradations not readily seen in these configurations, therefore, can and do escape detection until systems based on ACE+TAO are fielded by end-users [20], [13]. The key problem here is that the ACE+TAO developers are benchmarking a small and unrepresentative sample of system configurations, so their extrapolations from this data are bound to be unreliable.

To address this problem we have developed and evaluated the *reliable effects screening* process, which uses "design of experiments" theory [11] to determine an appropriate subset of system configurations to benchmark when the system changes. This section describes how we implemented the reliable effects screening process, applied it to ACE+TAO+CIAO, and disscusses several process choices developers must make when applying it.

A. *The Reliable Effects Screening Process*

Reliable effects screening (RES) is a process we developed to detect performance degradation rapidly across a large configuration space as a system changes. This process identifies a small subset of "important" configuration options that substantially affect variation in performance. Benchmarking a "screening suite" containing all combinations of these important option settings (with other options assigned randomly) should therefore provide a reliable estimate of performance across the entire configuration space at a fraction of the cost and effort of exhaustive benchmarking.

At a high level the process involves the following steps:

1) *Compute* a formal experimental design based on the system's QA task model.
2) *Execute* that experimental design across volunteered computing resources in the Skoll computing 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 most important options, *i.e.*, the options that affect performance most significantly.
4) Conduct *supplementary analysis* again on volunteered computing resources to check the basic assumptions underlying Step 1 and to confirm the results of Step 3.
5) *Estimate* overall performance (in-house, if possible) whenever the software changes by evaluating all combinations of the important options (while randomizing all other options).
6) Frequently *recalibrate* the important options by restarting the process since these effects can change over time, depending on how rapidly the subject system changes.

B. Screening Designs Background

The first step of reliable effects screening is to identify options accounting for the most performance variation across the system's QA task space. We do this by executing and analyzing formally-designed experiments, called *screening designs*, which are described in Kolarik [11], Wu and Hamada [30], or the NIST Engineering Statistics Handbook (`www.itl.nist.gov/div898/handbook/index.htm`). Screening designs are highly economical plans for identifying 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).

To better understand screening designs, consider a full factorial (*i.e.*, exhaustive) experimental design involving k independent binary factors. The design's run size (number of experimental observations) is therefore 2^k. Although such designs allow all 1^{st}- through k^{th}-order effects to be computed, they quickly become computationally expensive to run. Screening designs, in contrast, reduce costs by observing only a carefully selected subset of a full factorial design. The tradeoff is that they cannot compute most higher-order effects because the selection of observations aliases the effects of some lower-order interactions with some higher-order ones, *i.e.*, it conflates certain high- and low-order effects.

Which effects are conflated depends on the design's *resolution*. In resolution R designs, no effects involving i factors are aliased with effects involving less than $R-i$ factors. For instance, a resolution III design is useful to evaluate "clear" (no two aliased together) 1^{st}-order effects, where all higher effects are negligible. Here all 1^{st}-order effects will be clear, but they may be aliased with 2^{nd}- or higher-order effects. This conflation is justified only if the high-order effects are indeed negligible. If these assumptions are patently unreasonable, then a higher resolution may be needed.

In practice, statistical packages are used to compute specific screening designs. We used the SAS/QC [1] package in this work, but many other packages, such as MINITAB and SPSS, are also applicable. these packages will produce a screening design, assuming one can be found, given the following application-specific information: (1) a list of options and their settings, (2) a maximum run size, and (3) the design's resolution.

Since we already build our QA task model graphically using Skoll's MDE tools (see Section II), we can just use a translator to convert it into the list of options and settings expected by our statistical package. The second and third items are interwined and must be chosen by developers. In particular, higher resolution designs will yield more accurate estimates (assuming some higher-level effects exist), but require more observations. It is also often advisable to run more than the minimum number of observations needed for a given resolution to improve precision or to deal with noisy processes. Developers must balance these competing forces.

C. Computing a Screening Design

To demonstrate these choices, consider a hypothetical software system with 4 independent binary configuration options, A through D, each with binary settings + and −. A full factorial design for this system involves 16 (2^4) observations. We assume that our developers can only afford to gather 8 observations. With so few observations, there is no design with clear 1^{st}- and 2^{nd}-order effects. Developers must therefore either allow more observations or limit themselves to capturing only the 4 1^{st}-order effects, *i.e.*, the effect of each option by itself. We assume they choose to stay with a run size of 8 and to use a resolution IV design.

Given these choices, the developers generate one acceptable design using a statistical package.

The design (which appears in Table I) is identified uniquely as a 2^{4-1}_{IV} design, which means that the total number of options is 4, that they will observe a $1/2$ ($2^{-1} = 1/2$) fraction of the full factorial design, and that the design is a resolution IV screening design. The design tool also

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

TABLE I

2^{4-1}_{IV} DESIGN (BINARY OPTION SETTINGS ARE ENCODED AS (-) OR (+))

outputs the aliasing structure $D = ABC$. We can see this aliasing in Table I where the setting of option D is the product of the settings of options A, B and C (think of + as 1 and − as -1). This dependence explains why the effect of option D cannot be untangled from the interaction effect of options A, B, and C.

D. *Executing and Analyzing Screening Designs*

After defining the screening design, developers will execute it across the computing resources comprising the Skoll grid. In our later feasibility studies, each experimental observation involves measuring a developer-supplied benchmark program while the system runs in a particular configuration. Our QA engineers use BGML to generate workload and benchmark code. Once the data is collected we analyze it to calculate the effects. Since our screening designs are balanced and orthogonal by construction (*i.e.*, no bias in the observed data), the effect calculations are simple. For binary options (with settings − or +), the 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 required, 2^{nd}-order effects can be calculated in a similar way. The interaction effect of options 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 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 and for designs with non-binary options. See Wu and Hamada [30] for further details.

Once the effects are computed, developers will want to determine which of them are important and which are not. There are several ways to determine this, including using standard hypothesis testing. We do not use formal hypothesis tests primarily because they require strong assumptions about the standard deviation of the experimental samples. Instead, we display the effects graphically and let developers use their expert judgment to decide which effects they consider important. While this approach has some downsides (see Section VIII), even with traditional tests for statistical significance, experimenters must still judge for themselves whether a significant effect has any practical importance.

Our graphical analysis uses *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

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

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, but with means different from 0.[3] Options whose effects deviate substantially from 0 should therefore be considered important. If no effects are important, the resulting plot will show a set of points on an approximate line near $y = 0$.

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

E. Conduction Supplementary Analysis

At this point developers have a candidate set of important options. One potential problem, however, is that we arrived at these options by making the following assumptions:

1) The low-order effects identified as being important really are; while the higher-order effects they are aliased to are not.
2) Monitoring only low-order effects is sufficient to produce reliable estimates.

Since these are only assumptions, it is important to check them before proceeding since the reliability of our results will be severely compromised if they do not hold. We therefore validate these assumptions using two types of follow-up experiments:

1) We first examine additional configurations to disambiguate any effects aliased to our purported low-level important options.
2) We then examine additional configurations to look for other higher-order effects.

In these follow-up experiments we rely on another class of efficient experimental designs called *D-optimal designs* [22], which are again computer-aided designs. Given a configuration space and a model the experimenter wishes to fit, a D-optimal design uses search-based computer algorithms (*e.g.*, hill climbing or simulated annealing) to select a set of configurations that satisfy a particular optimality criterion. Unlike more common fractional factorial designs, therefore, the size of D-optimal designs need not be a perfect fraction of full factorial designs. D-optimal designs are preferable to standard classical designs when (1) the standard designs require more observations than can be tested with available time and resources and (2) the configuration space is heavily constrained (*i.e.,* when not all the configurations are valid). Both factors are frequently present in modern software systems. Full details of D-optimail designs are beyond the scope of this paper, but can be found in books and articles (See Kolarik [11] and Mitchell [22]).

Based on the results of this D-optimal design analysis, developers may modify the set of important options. At this point developers have a working set of important options that they can use to create a screening suite of configurations to benchmark whenever the system changes.

IV. Feasibility Study

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

A. Experimental Design

Hypotheses. Our feasibility study explores the following three hypotheses:

1) Our MDE-based Skoll environment cost-effectively supports the definition, implementation, and execution of our reliable effects screening process described in Section III.
2) The screening designs used in the reliable effects screening correctly identifies a small subset of options whose effect on performance is important.
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 three open-source software frameworks for performance-intensive systems: ACE v5.4 + TAO v1.4 + CIAO v0.4, which can be downloaded via `www.dre.vanderbilt.edu`. ACE provides reusable C++ wrapper facades and framework components that implements core concurrency and distribution patterns [26] for distributed real-time and embedded (DRE) systems. TAO is a highly configurable Real-time CORBA Object Request Broker (ORB) built atop ACE to meet the demanding QoS requirements of DRE systems. CIAO extends TAO to support components, which enables developers to declaratively provision QoS policies end-to-end when assembling DRE systems.

ACE+TAO+CIAO are ideal subjects for our feasibility study since they share many characteristics with other highly configurable performance-intensive systems. For example, they collectively have over 2M+ lines of source code, functional regression tests, and performance benchmarks contained in $\sim$4,500 files that average over 300 CVS commits per week by dozens of developers around the world. 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 ACE 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. The entire configuration space therefore has $2^{14} = 16,384$ different configurations. To save space, we refer to these options by their one letter indices, A-N (see Table II for more details on the mapping of letters to options).

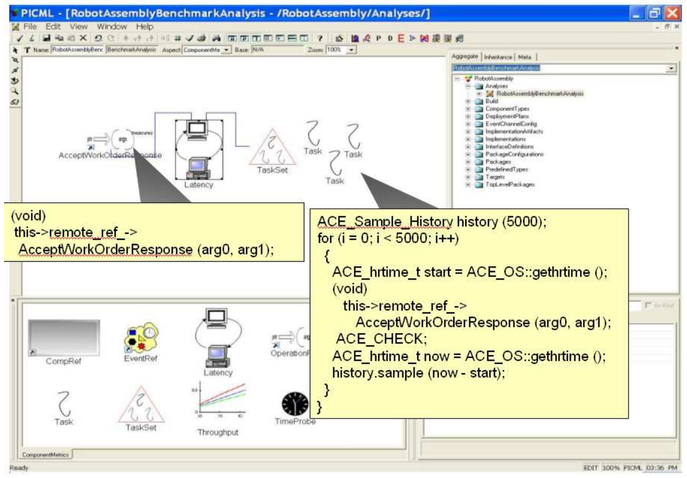
TABLE II

SOME ACE+TAO OPTIONS

Option Index	**Option Name**	**Option Settings**	**Option Description**
A	ReactorThreadQueue	{FIFO, LIFO}	Order in which incoming requests are processed in the ORB Reactor
B	ClientConnectionHandler	{RW, MT}	Client side connection handler
C	ReactorMaskSignals	{0, 1}	Enable/disable signals during request processing
D	ConnectionPurgingStrategy	{LRU, LFU}	ORB connection purging strategy
E	ConnectionCachePurgePercent	{10, 40}	% of the ORB connection cache purged
F	ConnectionCacheLock	{thread, null}	Enable/disable locking of the ORB connection cache
G	CorbaObjectLock	{thread, null}	Enable/disable locking while synchronizing object state
H	ObjectKeyTableLock	{thread, null}	Type of lock to be used within ORB for retrieving object keys
I	InputCDRAllocator	{thread, null}	Enable/disble locking during creating CDR streams
J	Concurrency	{reactive, thread-per-connect}	ORB concurrency reactive or thread per connection
K	ActiveObjectMapSize	{32, 128}	Map size for holding objects
L	UseridPolicyDemuxStrategy	{linear, dynamic}	Demultiplexing strategy when user-id policy is used
M	SystemPolicyDemuxStrategy	{linear, dynamic}	Demultiplexing strategy when system id policy is used
N	UniquePolicyRevDemuxStrategy	{linear, dynamic}	Specify the reverse demultiplexing lookup strategy to be used with the unique id policy

Using MDE tools to model experiment scenario. To ease experiment creation and generation, we used the Skoll MDE tools to compose the experiment visually. In particular, we used BGML to generate the platform-specific benchmarking code needed to evaluate the QoS of the ACE+-

TAO+CIAO software framework configurations. Figure 3 shows how we used BGML to model the benchmark. For the experiment, we modeled the operation exchanged between the client and



```
(void)
 this->remote_ref_->
  AcceptWorkOrderResponse (arg0, arg1);
```

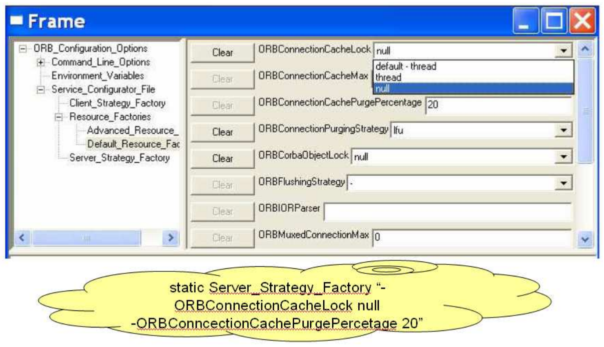
```
ACE_Sample_History history (5000);
for (i = 0; i < 5000; i++)
 {
   ACE_hrtime_t start = ACE_OS::gethrtime ();
   (void)
     this->remote_ref_->
       AcceptWorkOrderResponse (arg0, arg1);
    ACE_CHECK;
   ACE_hrtime_t now = ACE_OS::gethrtime ();
   history.sample (now - start);
 }
}
```

Fig. 3. Using BGML to Generate Benchmarking Code

server using BGML's operation element. We then associated this element with BGML's latency metric to compute the end-to-end measurements for our experiments. The number of warm up iterations and the data type exchanged between client and server were all set as attributes to the operation and latency elements provided by BGML. As shown in the figure, BGML code generators generated the benchmarking code to measure and capture the latency for our experiment.

Another step in designing our experiment involved modeling the ACE+TAO+CIAO framework configurations. We used the OCML MDE tool to ensure that the configuration were both syntactically and semantically valid, as shown in Figure 4. As shown in the figure, OCML was used to enter the ACE+TAO+CIAO configurations we wanted to measure. OCML's constraint checker first validated the configurations we modeled, while the code generator produced the framework configuration files.

Experimental process. Our experimental process used Skoll's MDE tools to implement the reliable effects screening process and evaluate our three hypotheses above. We executed the reliable 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 tasks involved running a benchmark application in a particular system configuration, which



ORBConnectionCacheLock null
ORBConnectionCacheMax
ORBConnectionCachePurgePercentage 20
ORBConnectionPurgingStrategy lfu
ORBCorbaObjectLock null
ORBFlushingStrategy -
ORBIORParser
ORBMuxedConnectionMax 0

```
static Server_Strategy_Factory "-
ORBConnectionCacheLock null
-ORBConncectionCachePurgePercetage 20"
```

Fig. 4. Using OCML to Capture Experiment Configuration

evaluated performance for the application scenario outlined above. The benchmark created an ACE+TAO+CIAO client and server and then measured message latency and overall throughput between the client and the server.

In our experiment, the client sent 300K requests to the server. After each request the client waited for a response from the server and recorded the latency measure in microseconds. At the end of 300K requests, the client computed the throughput in terms of number of requests served per second. We then analyzed the resulting data to evaluate our hypotheses. Section VIII describes the limitations with our current experimental process.

B. The Full Data Set

To provide a baseline for evaluating our approach, we first generated and analyzed performance data for all 16,000+ valid configurations of ACE+TAO+CIAO.[4] We refer to these configurations as the "full suite" and the performance data as the "full data set."

We examined the effect of each option and judged whether they had important effects on performance. Figure 5 plots the effect of each of the 14 ACE+TAO+CIAO options on latency and throughput across the full data set. We see that options B and J are clearly important, whereas options I, C, and F are arguably important. The remaining options are not important.

[4]We would not do this step in practice since it required about two days of CPU time, which would be prohibitively expensive in most production software development environments with scores of such experiments running daily.

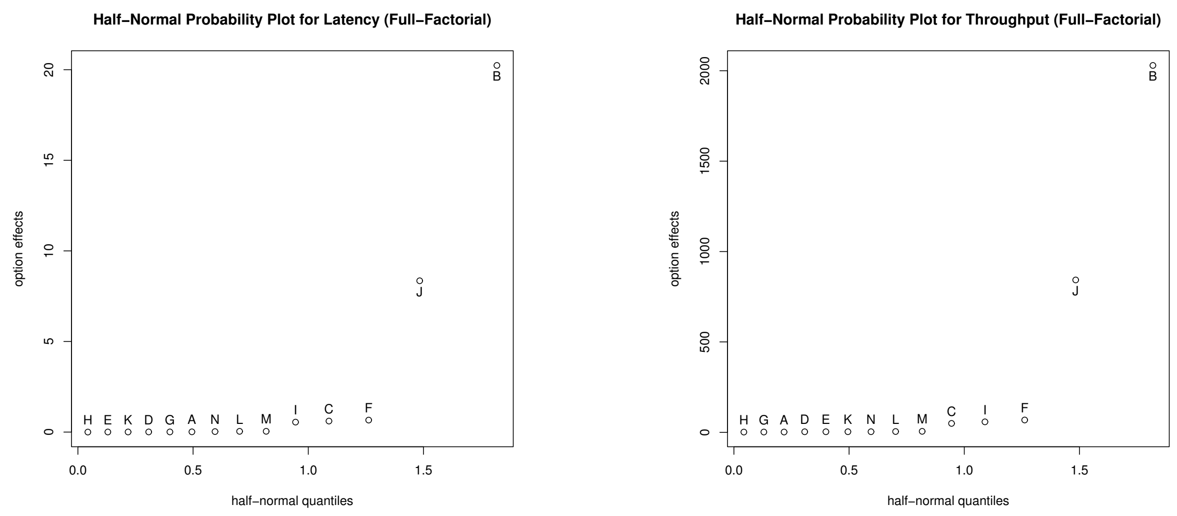


Fig. 5. Option Effects Based on Full Data

C. Evaluating Screening Designs

We now walk through the steps involved in conducting the reliable effects screening process, to see whether the remotely executed screening designs can correctly identify the same important options discovered in the full data set.

To perform these steps, we calculated and executed several different resolution IV screening designs of differing run sizes. The specifications for these designs appear in the Appendix. The first set of designs examined all 14 options using increasingly larger run sizes (32, 64, or 128 observations) to identify only important 1^{st}-order effects. We refer to these screening designs as Scr_{32}, Scr_{64} and Scr_{128}, respectively. We also calculated and executed a second set of designs that attempted to capture important 1^{st}- and 2^{nd}-order effects.

Figure 6 shows the half-normal probability plots obtained from our first set of screening designs. The figures show that all screening designs correctly identify options B and J as being important. 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 showed identical results unless otherwise stated).

D. Higher-Order Effects

The Resolution IV design only calculates clear 1^{st}-order effects, which appears to work well for our subject application and scenario, but might not be sufficient for other situations. Figure 7

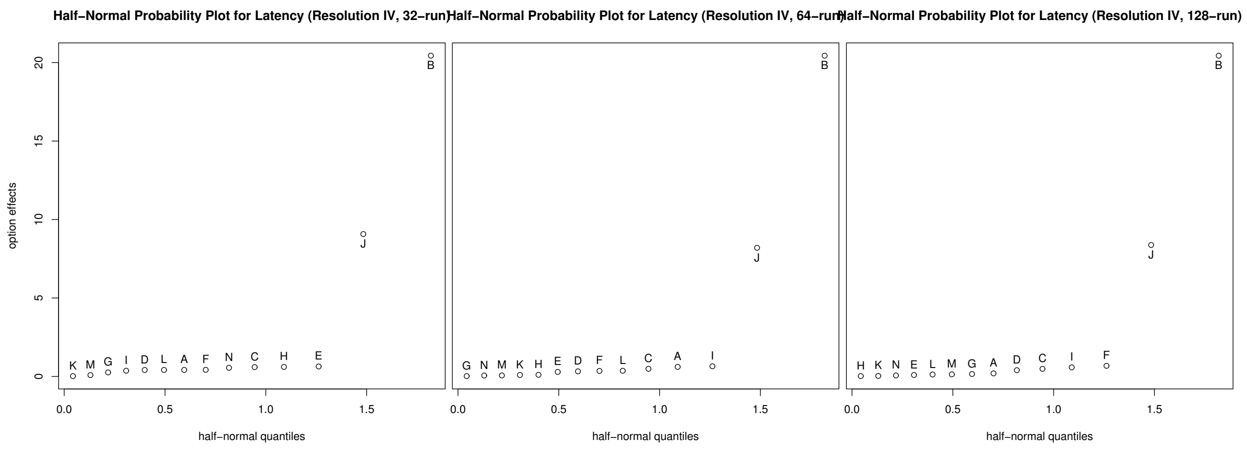


Fig. 6. Option Effects Based on Screening Designs

shows the effects of all pairs of options based on the full data set and the same effects captured via a Resolution VI screening design using 2,048 observations. From the figure we see several things: (1) the screening design correctly identifies the 5 most important pairwise interactions at $1/8^{th}$ the cost of exhaustive testing and (2) the most important interaction effects involve only options that are already considered important by themselves, which supports the belief that monitoring only first-order effects will be sufficient for our subject frameworks.

E. Validating Basic Assumptions

To compute the important options, we used a resolution IV screening design, which according to the definition given in Section III, means that (1) we aliased some 1^{st}-order effects with some 3^{rd}-order or higher effects and some 2^{nd}-order effects with other 2^{nd}-order or higher effects, and (2) we assume that 3^{rd}-order or higher effects are negligible. If these assumptions do not hold however performance estimations will be incorrect. Therefore, at this stage we perform two further analyses to validate these assumptions.

In the remainder of this section we analyze the data from our 2^{14-7}_{IV}, 128-run experiment. We did this experiment because it was the only one where we identified five important options rather than two. The analysis presented in this section is readily applicable to any screening experiment.

1) Breaking Aliases: In the 2^{14-7}_{IV}, 128-run experiment, we identified options B and J as being clearly important and options C, F, and I as being arguably important (see Table II for

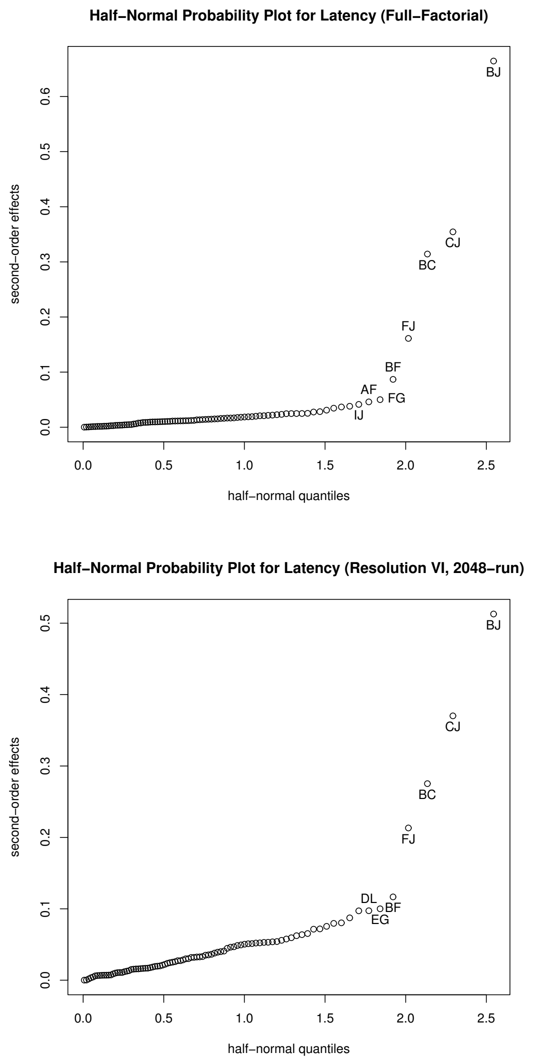


Fig. 7. Pairwise Effects Based on Full and Screening Suite

the list of options). In that design, some 1^{st}-order effects were aliased with some 3^{rd}-order or higher effects. Table III shows the aliasing structure for the important options up to 4^{th}-order effects. Note that the complete aliasing structure for the important options contains a total of 640 aliases.

Consider the alias $B = ACH$ for the important option B; the effect of option B is aliased with the interaction effect of options A, C, and H. Since the experimental analysis cannot distinguish B from ACH, there is an ambiguity whether B or ACH is really important. If ACH, not B, is the important effect then our performance estimations would obviously suffer, as would the rest

Partial Aliasing Structure
B = ACH = CGKM = CKLN = DGHL = DHMN = EFHK = HIJK = ADEI = ADFJ
J = EFI = ACIK = BHIK = CDFH = DEGM = DELN = FGKN = FKLM = ABDF
C = ABH = ADMN = AIJK = BGKM = BKLN = DEHI = DFHJ = AEFK = ADGL
F = EIJ = BEHK = CDHJ = DGIM = DILN = GJKN = JKLM = ABDJ = ACEK
I = EFJ = ACJK = BHJK = CDEH = DFGM = DFLN = EGKN = EKLM = ABDE

TABLE III

PARTIAL ALIASING STRUCTURE FOR THE IMPORTANT OPTION IN THE 2_{IV}^{14-7}, 128-RUN DESIGN

of the aliases. Ambiguities involving aliased effects for the important options should therefore be resolved to ensure the reliability of the performance estimations.

To resolve the ambiguity in this experiment, we first formally modeled the important effects and their 640 aliases by the following linear model:

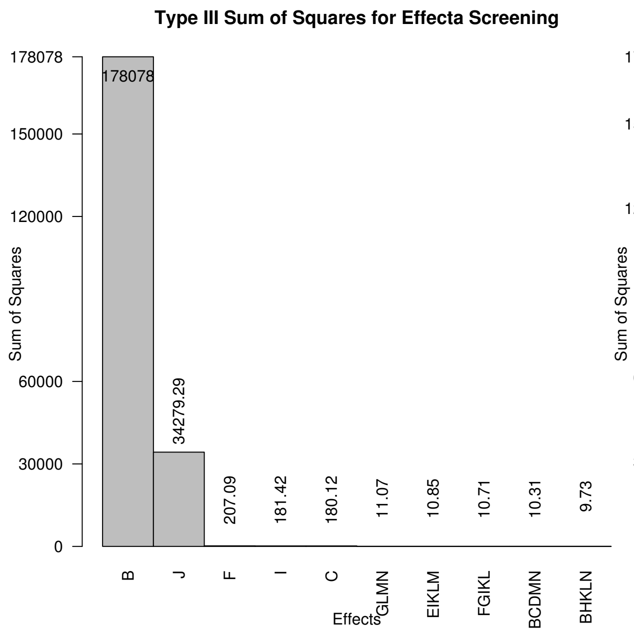
$$\begin{aligned} y &= \beta_0 + \beta_B x_B + \beta_J x_J + \beta_C x_C + \beta_F x_F + \beta_I x_I \\ &+ \beta_{ACH} x_A x_C x_H + \beta_i x_i, \;\; \forall i \in S - \{ACH\} \end{aligned} \tag{5}$$

where S is the set of all 640 aliased effects, β_0 is the intercept term, $x_B = -1, 1$ according to the level of option B (the definitions for the other x's are similar), β's are the model coefficients, etc.

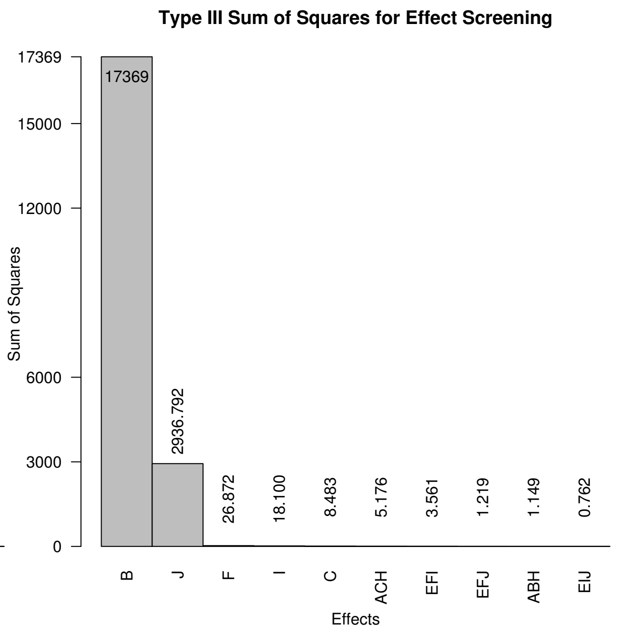
Note that since the 1^{st}-order effects other than B, J, C, F, and I are negligible (Section IV-C), we excluded them from the model. We then augmented our 128-run screening experiment using the D-optimal design approach. This search-based technique determined an appropriate design which required benchmarking an additional 2,328 configurations. We then collected the results and analyzed them together with the results of the original screening experiments.

Figure 8(a) plots the Type III sum of squares for the factorial effects. In this figure, the effects are ordered in descending order. Due to space limitations, only the top 10 effects are given. The higher the sum of squares, the more important the effects are. As can be seen from this figure, options B, J, C, F, and I are important, while the higher-order interactions to which they are aliased are not, *i.e.*, the low-order effect explains roughly 10 times the variance of its higher-order alias. Further statistical analysis also showed that these options are statistically significant at 99.99% confidence level or better.

Although this analysis confirmed our earlier results, it required benchmarking an additional



(a)



(b)

Fig. 8. (a) Complete Dealiasing Experiment and (b) Up to and Including 3^{rd}-order Effects Dealiasing Experiment.

2,000+ configuration, which in large-scale systems might too expensive. To further reduce costs, one might forego a complete dealiasing of important options, dealiasing them up to a certain level of order (*e.g.,* up to 3^{rd}-order or 4^{th}-order effects), or dealiasing only suspected aliases based on developer's domain knowledge.

To see how much savings partial dealising might yield, we repeated the evaluation process, dealiasing the important options only up to and including 3^{rd}-order effects, which required only 64 additional configurations. As can be seen in Figure 8(b), in this particular example, we were able to reach the same conclusions as the complete dealiasing experiment.

2) Checking for Higher-Order Effects: As described earlier in this Section, our second assumption was that monitoring only 1^{st}-order effects was sufficient for our subject frameworks. Below, we investigate the validity of that assumption. Without loss of generality, we will look for important effects up to and including 3^{rd}-order effects. Developers can choose up to which level to examine, keeping in mind that as the level increases, the number of observations needed converges to exhaustive testing.

Just as in Section IV-E.1, we augmented the 2_{IV}^{14-7}, 128-run experiment using the D-optimality criterion. this time, however, our generalized linear model consisted of all the 1^{st}-, 2^{nd}-, and

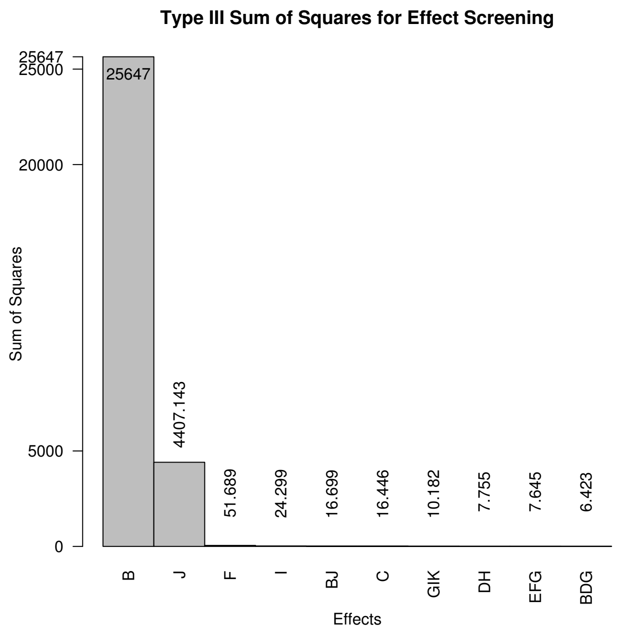


Fig. 9. Looking for Higher-order Effects Upto and Including 3^{rd}-order Effects.

3^{rd}-order effects. This design required only an additional 381 configurations. Figure 9 shows the results of this study. We identified the top 6 effects: B, J, F, I, BJ, and C, as statistically significant at 99.9% confidence level or better. Among these important options we have only one interaction effect: BJ. Since this interaction involves only options that are already considered important by themselves, we conclude that monitoring only first-order effects was sufficient for our subject frameworks.

The results presented above suggest that screening designs can detect important options at a small fraction of the cost of exhaustive testing and that techniques exist to economically check key assumptions underlying the technique. The smaller the effect, however, the larger the run size needed to identify it. Developers should therefore 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).

F. Estimating Performance with Screening Suites

Our experiments thus far have identified a small set of important options. We now evaluate whether benchmarking all combinations of these 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 randomizing the settings of the unimportant options.

In the Section IV-C, we determined that options B and J were clearly important and that options C, I, and F were arguably important. Developers therefore made 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 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 10 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. The distributions of the top-5 and top-2

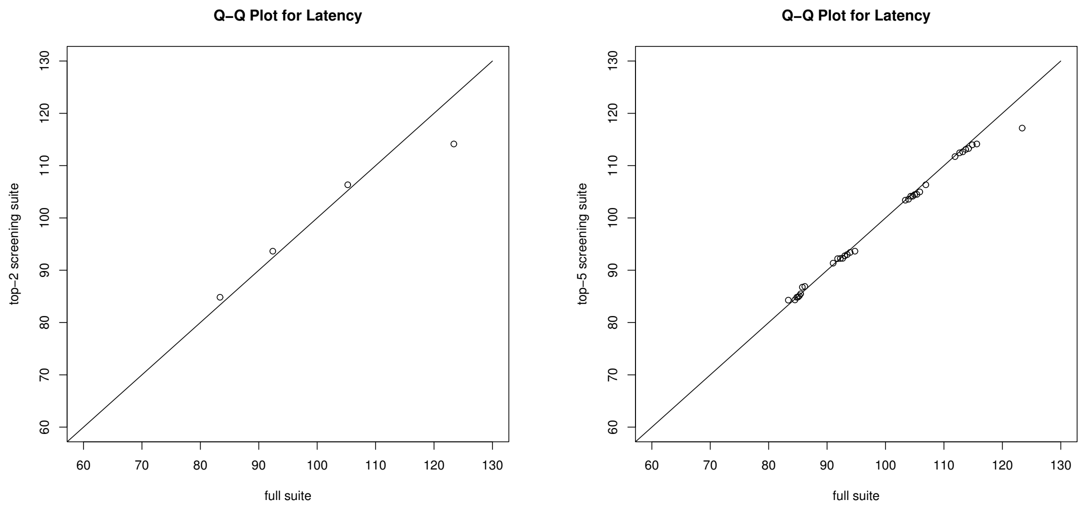


Fig. 10. Q-Q plots for the Top-2 and Top-5 Screening Suites

screening suites closely track the overall performance data. Such plots, called quantile-quantile (Q-Q) plots, are used to see how well two data distributions correlate by plotting 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 the $x = y$ line.

We also performed Mann-Whitney non-parametric tests [3] to determine whether each set of screening data (top-2 and top-5 suites) appears to come from the same distribution as the full data. In both cases we were unable to reject the null hypothesis that the top-2 and top-5 screening suite data come from the same distribution as the full suite data. These results suggest that the screening suites computed at Step 4 of the reliable effects screening process (Section III) 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 two days of CPU time.

G. Screening Suites vs. Random Sampling

Another question we addressed is whether our reliable effects screening process was better than other low-cost estimation processes. In particular, we compared the 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 11. These box plots show the

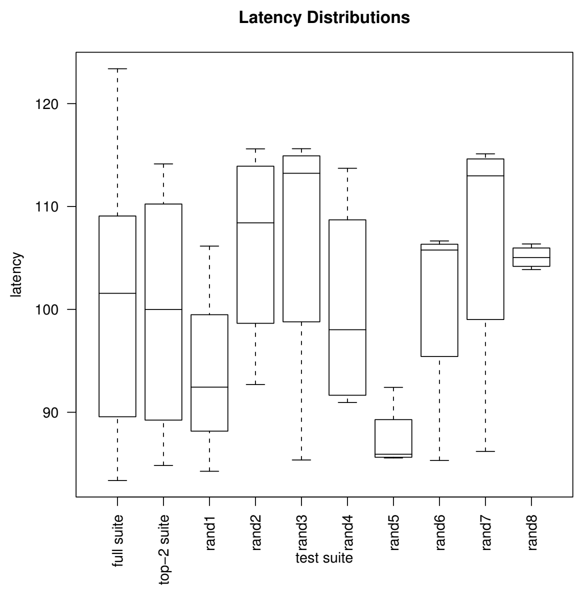


Fig. 11. Latency Distribution from Full, Top-2, and Random Suites

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.

H. Dealing with Evolving Systems

The primary goal of reliable effects screening is to detect performance degradations in evolving systems *quickly*. So far, we have not addressed 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 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 assess the system's performance without having to test all configurations for each system change exhaustively.

Figure 12 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,

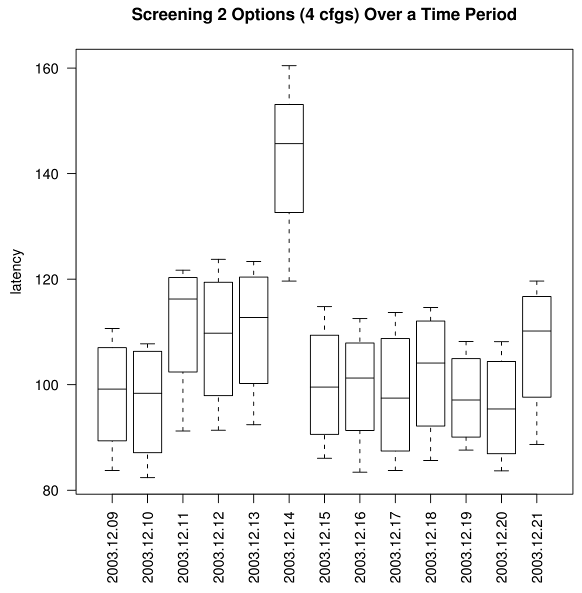


Fig. 12. Performance Estimates Across Time

crept up somewhat for days 3 through 5, and then shot up the 6^{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 not noticed until 12/14/03.

Another interesting finding was that the limited in-house testing conducted by the 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 ACE+-TAO+CIAO developers showed that their unsystematic testing failed to evaluate configurations where the degradation was much more pronounced.

Taken together, our results suggest that benchmarking only all combinations of the important options identified in steps 1–4 of the reliable effects screning process gives much the same information as benchmarking the entire configuration space, but at a substantially reduced cost.

V. Discussion of Feasibility Study

A. Threats to Validity

All empirical studies suffer from threats to their internal and external validity. For the experiments described in Section IV, we were primarily concerned with threats to external validity since they limit our ability to generalize the results of our experiment to industrial practice. One potential threat is that several steps in our process require human decision making and input. For example, developers must provide reasonable benchmarking applications and must also decide which effects they consider important.

Another possible threat to external validity concerns the representativeness of the ACE+TAO+-CIAO subject applications, which are an integrated suite of software (albeit a very large suite with over 2M+ lines of code). A related issue is that we have focused on the 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 (screening designs can be created for much more complex situations), there is clearly a need to apply reliable effects screening to larger and richer configuration spaces in future work to understand how well the process scales.

Another potential threat is that for the time period we studied, the ACE+TAO+CIAO subject frameworks were in a fairly stable phase. In particular, changes were made mostly to fix bugs and reduce memory footprint, but the software's core functionality was relatively stable. For time periods where the core software functionality is in greater flux, *e.g.*, in response to new requirements from sponsors or efforts to port the software to new and different platforms, it may be harder to distinguish significant performance degradation from normal variation.

B. Hypotheses

Despite the limitations described in Section V-A, we believe the study presented in Section IV supports our basic hypotheses presented in Section IV-A. We reached this conclusion by noting that our study suggests that the (1) MDE-based Skoll system allows QA engineers to quickly construct complex DCQA processes, (2) reliable effects screening processes provides developers with fast, cheap and reliable estimates of performance across a system's entire configuration space; and (3) developers of ACE+TAO+CIAO believe the technique provides them with important development information.

Benefits from applying our MDE tools. Our MDE tools helped improve the productivity of QA engineers by allowing them to create QA task models and to compose benchmarking experiments *visually* rather than wrestling with low-level formats and source code. These tools thus resolve tedious and error-prone accidental complexities associated with writing correct code by auto-generating them from higher level models. For example, Table IV summarizes the BGML code generation metrics for a particular configuration.

TABLE IV

GENERATED CODE SUMMARY FOR BGML

Files	Number	Lines of Code	Generated (%)
IDL	3	81	100
Source (.cpp)	2	310	100
Header (.h)	1	108	100
Script (.pl)	1	115	100

This table shows how BGML automatically generates 8 of 10 required files that account for 88% of the code required for the experiment. Since these files must capture specific information for each configuration, these tools imply large improvements in productivity for performing benchmarking QA tasks. Similarly, OCML enabled us to generate both syntactically and semantically correct middleware configurations, thereby eliminating accidental complexity in generating middleware configurations.

Reliable effects screening. Our experiments showed that the reliable effects screening process was fast, cheap, and effective. We came to this conclusion by noting that:

1) Screening designs can correctly identify important options (Sections IV-C – IV-E).

2) These options can be used to produce reliable estimates of performance quickly across the entire configuration space at a fraction of the cost of exhaustive testing (Section IV-F).
3) The alternative approach of random or *ad hoc* sampling can give highly unreliable results (Section IV-G).
4) The reliable effects screening process detected performance degradation on a large and evolving software system (Section IV-H).
5) The screening suite estimates were significantly more precise than the *ad hoc* process currently used by the developers of ACE+TAO+CIAO (Section IV-H).

User acceptance. Informally, we found that ACE+TAO+CIAO developers have been quite happy with the results of our experiments described in Section IV. As we move towards fully integrating reliable effects screening into their development processes they continue to find new ways in which this information can help them improve their development processes, including:

Using option importance to prioritize work. Our quantitative results showed that options L, M, and N did not have a strong effect on latency and throughput. These findings surprised some ACE+TAO+CIAO developers, who had spent considerable time optimizing code affected by these options. Further investigation showed that the options can have a somewhat larger effect, but only in very specific circumstances. The developers now see reliable effects screening as a means to better understand how widespread the effects of different pending changes may be.

Using changes in option importance to detect bugs. Prior to the release of an ACE+TAO+-CIAO beta, developers noted a significant ($\sim$25%) drop in performance. Since reliable effects screening had not yet been fully integrated into their development processes, the ACE+TAO+-CIAO developers fell back on traditional *ad hoc* QA and debugging techniques. When they failed to identify the problem's cause, they froze their CVS repository and used an exhaustive QA approach to painstakingly narrow down the change that had degraded performance.

Ultimately they found the change that caused the degradation was a feature addition that enabled TAO+CIAO to support the IPv6 protocol. Specifically, they observed that a modification to TAO's connection handler had degraded performance. Interestingly, this code is controlled by option B. In retrospect, a reliable effects screening would have shown a dramatic change in the importance of option B, thus enabling the ACE+TAO+CIAO developers to localize the problem quickly, *i.e.*, because one setting of option B triggered the buggy code, option B's effect would have been 2.5 times greater after the change than before it. Based on this experience, the developers now see added value in frequently recalibrating the important options to alert them to changes in option importance.

VI. Usage Guidelines

This section focuses on providing some guidelines on how to use the Skoll-based reliable effects screening process. We examine how to select an appropriate resolution and size for the screening designs, how to identify the important options, and how to validate the basic assumptions in identifying the important options. We also summarize on our experience to date applying this process on the ACE+TAO+CIAO software frameworks.

Step 1: Choose the resolution. Leverage *a priori* knowledge of the software being tested, if it is available, to decide the resolution of the screening experiment, *i.e.*, which high-order effects are considered important vs. negligible. If no or limited *a priori* information is available, use screening experiments in an iterative manner (*e.g.,* going from lower resolutions to higher ones) to obtain this information. Section IV-A illustrated how we did this for the ACE+TAO+CIAO software frameworks, where we selected 14 specific options to explore based prior knowledge of a recent change.

Step 2: Choose the size. Depending on available resources and how fast the underlying software changes, determine the maximum number of observations allowed in the screening experiment. Note that the resolution of the experiment chosen in step (1) may dictate the minimum size of the experiment. If this size is not feasible, consider lowering the resolution of the experiment by carefully choosing the aliasing structure so that no potentially important higher-order effects are aliased with lower-order ones. Sections IV-C and IV-D illustrate how we did this for the ACE+TAO+CIAO software frameworks, where we created three resolution IV designs with run sizes of 32, 64, and 128, and where we created one resolution VI design with a run size of 2,048.

Step 3: Identify the important options. After the screening design is computed, conducted, and analyzed, use the half-normal probability plots described in Section IV-C to identify important options. If no effects are important, these plots will show a set of points on a rough line near $y = 0$. Any substantial deviations from this line indicate important options. Depending on the benchmarking test and the desired precision of the performance estimates, decide how large

effects must be to warrant attention. Section IV-C shows how we did this for the ACE+TAO+-CIAO software frameworks, where we identified 2 important and 3 arguably important options.
Step 4: Validate the basic assumptions. If needed, validate the assumptions imposed by the choice of the resolution. Use D-optimal designs described in Section IV-E to augment the screening experiment to (1) dealias the important options and (2) identify remaining higher-order effects. Section IV-E illustrated how we did this for the ACE+TAO+CIAO software frameworks and showed that our basic assumptions helped and that our initial analysis was therefore reliable.
Step 5: Estimate performance after changing software. Focusing on important options allows developers to reduce the effective configuration space significantly by evaluating all combinations of the important options, while randomizing the rest. Section IV-F illustrated how we did this for the ACE+TAO+CIAO software frameworks and showed that our approach gave reliable estimates of performance across the entire configuration space using only 40 seconds (for top-2 suite) or 2 minutes (for top-5 suite) of CPU time.
Step 6: Frequently recalibrate important options. The importance of different options may change as software in a system changes. We therefore recommend frequent recalibration of the important effects. Although our feasibility study in Section IV does not show the need for recalibration, our experience applying reliable effects screening to ACE+TAO+CIAO over time indicates that recalibration is essential.

VII. Related Work

This section compares our work on reliable effects screening and performance evaluation techniques in Skoll with other related research efforts, including (1) applying *design-of-experiments (DOE) testing* to software engineering, (2) large-scale testbed environments for conducting experiments using heterogeneous hardware, OS, and compiler platforms, (3) evaluating the performance of layered software systems, and (4) feedback-based optimization techniques that use empirical data and mathematical models to identify performance bottlenecks.
Applying DOE to software engineering. As far as we know, we are the first to use screening designs to assess software performance. The use of DoE theory within software engineering has focused mostly on *interaction testing*, largely to compute and sometimes generate minimal test suites that cover all combinations of specified program inputs. Mandl [19] first used orthogonal arrays, a special type of covering array in which all t-sets occur *exactly* once, to test enumerated

types in ADA compiler software. This idea was extended by Brownlie *et al.* [5] who developed the orthogonal array testing system (OATS). They provided empirical results to suggest that the use of orthogonal arrays is effective in fault detection and provides good code coverage.

Dalal *et al.* [8] argue that the testing of all pairwise interactions in a software system finds a large percentage of the existing faults. In further work, Burr *et al.* [6], Dunietz *et al.* [9] and Kuhn *et al.* [15] provide more empirical results to show that this type of test coverage is effective. These studies focus on finding unknown faults in already tested systems and equate covering arrays with code coverage metrics. Yilmaz *et al.* [32] used covering arrays as a configuration space sampling technique to support the characterization of failure-inducing option settings.

Large-scale benchmarking testbeds. EMULab [29] is a testbed at the University of Utah that provides an environment for experimental evaluation of networked systems. EMULab provides tools that researchers can use to configure the topology of their experiments, *e.g.*, by modeling the underlying OS, hardware, and communication links. This topology is then mapped to ∼250 physical nodes that can be accessed via the Internet [24] . The EMULab tools can generate script files that use the Network Simulator (NS) (`www.isi.edu/nsnam/ns/`) syntax and semantics to run the experiment.

The Skoll infrastructure provides a superset of EMULab that is not limited by resources of a particular testbed, but instead can leverage the vast end-user computer resources in the Skoll grid. Moreover, the Skoll's MDE-based tools described in Section II can generate NS scripts to integrate our benchmarks with experiments in EMULab.

Feedback-driven optimization techniques. Traditional feedback-driven optimization techniques can be divided into *online*, *offline*, and *hybrid* analysis. Offline analysis has commonly been applied to program analysis to improve compiler-generated code. For example, the ATLAS [34] 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.

Like ATLAS, Skoll's MDE tools use an optimization engine to configure/customize software parameters in accordance to available OS platform characteristics (such as the type of threading, synchronization, and demultiplexing mechanisms) and characteristics of the underlying hardware (such as the type of CPU, amount of main memory, and size of cache). This information can be used to select optimal configurations ahead of time that maximize QoS behavior.

Online analysis is commonly used for feedback control to adapt QoS behaviors based on dynamic measures. An example of online analysis is the ControlWare middleware [35], which uses 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 [18] to automatically adjust the rate of remote operation invocation transparently to an application.

Though online analysis enables systems to adapt at run-time, the optimal set of QoS features are not determined at system initialization. Using the MDE tools, QoS behavior and performance bottlenecks on various hardware and software configurations can be determined offline and then fed into the models to generate optimal QoS characteristics at model construction time. Moreover, dynamic adaptation can incur considerable overhead from system monitoring and adaptation, which may be unacceptable for performance-intensive DRE systems.

Hybrid analysis combines aspects of offline and online analysis. For example, the continuous compilation strategy [7] 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.

Skoll's MDE-based strategy enhances conventional hybrid analysis by tabulating platform-specific and platform-independent information separately using the Skoll framework. In particular, Skoll does not incur the overhead of system monitoring since behavior does not change at run-time. New platform-specific information obtained can be fed back into the models to optimize QoS measures.

Generative Benchmarking Techniques. There have been a several initiatives that use generative techniques similar to our approach for generating test-cases and benchmarking for performance evaluation. The ForeSight [17] tool uses empirical benchmarking engine to capture QoS information for COTS based component middleware system. The results are used to build mathematical models to predict performance. This is achieved using a three pronged approach of (1) create a performance profile of how components in a middleware affect performance, (2) construct a reasoning framework to understand architectural trade-offs, *i.e.*, know how different QoS attributes interact with one another and (3) feed this configuration information into generic performance models to predict the configuration settings required to maximize performance.

The SoftArch/MTE [10] tool provides a framework for system architects to provide higher level abstraction of the system specifying system characteristics such as middleware, database technology, and client requests. The tool then generates an implementation of the system along with the performance tests that measure system characteristics. These results are then displayed back, *i.e.*, annotated in the high level diagrams, using tools such as Microsoft Excel, which allows architects to refine the design for system deployment.

Our MDE approach closely relates to the aforementioned approaches. Both the ForeSight and SoftArch tools, however, lack DCQA environments to help capture QoS variations accurately on a range of varied hardware, OS, and compiler platforms. Rather than using generic mathematical models to predict performance, MDD tools use a feedback-driven approach [13], wherein the DCQA environment is used to empirically evaluate the QoS characteristics offline. This information can then be used to provide QA engineers with accurate system information. Moreover, platform- and application-specific optimization techniques [12] can then be applied to maximize QoS characteristics of the system.

VIII. Concluding Remarks

This paper presents a new *distributed continuous quality assurance (DCQA)* process called "reliable effects screening" that uses in-house and in-the-field resources to efficiently and reliably detect performance degradation in performance-intensive systems that have large configuration spaces. The novelty of our approach stems from its application of statistical quality control techniques to efficiently identify a subset of system configurations that accurately represents the performance of the entire configuration space. Benchmarking this subset after a change provides a reliable estimate of the distribution of performance across all configurations. Unexpected changes in this distribution signal unintended side effects that must be examined further.

To evaluate the reliable effects screening process, we integrated it with the Skoll DCQA environment and applied it to three large software projects (ACE, TAO, and CIAO) using a grid of computing resources managed by the Skoll DCQA environment. Skoll's reliable effects screening process is also supported by model-driven engineeering (MDE) tools that control and manage its execution across a grid of in-house and in-the-field computing resources. The results of our experiments in this environment indicated that:

- The reliable effects screening process helped developers detect and understand performance bottlenecks across large configuration spaces.
- The ACE+TAO+CIAO developers used the information provided by this process as a scalable defect detection aid, *i.e.*, when the important options change unexpectedly (at recalibration time), developers re-examined the frameworks to rapidly identify possible problems with software updates.
- ACE+TAO+CIAO developers also used information provided by the reliable effects screening process to understand and arbitrate disputes about subtle changes in framework and application performance characteristics.

In conclusion, we believe that this line of research is novel and fruitful, though much R&D remains to be done. We are therefore continuing to develop enhanced MDE-based Skoll capabilities and use them to create and validate more sophisticated DCQA processes that overcome the limitations and threats to external validity described above. In particular, we are exploring the connection between design-of-experiments theory and the QA of other software systems with large software configuration spaces. We are also incorporating the tools in the Skoll environment and the reliable effect screening process into open-source software repositories, such as ESCHER (`www.escherinstitute.org`). Finally, we are conducting a much larger case study using Skoll and reliable effect screening to orchestrate the ACE+TAO+CIAO daily build and regression test process with 200+ machines contributed by users and developers worldwide, as shown on our online DCQA scoreboard at `www.dre.vanderbilt.edu/scoreboard`.

IX. Acknowledgments

This material is based on work supported by the US National Science Foundation under NSF grants ITR CCR-0312859, CCF-0447864, CCR-0205265, and CCR-0098158, as well as funding from BBN, Cisco, DARPA, Lockheed Martin Advanced Technology Lab and Advanced Technology Center, ONR, Qualcomm, Raytheon, and Siemens. We thank the anonymous reviewers for their comments that helped improve this paper.

REFERENCES

[1] “SAS Institute,” www.sas.com/.

[2] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema, “Developing applications using model-driven design environments,” *IEEE Computer*, vol. 39, no. 2, pp. 33–40, 2006.

[3] G. E. P. Box, W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters*. New York: John Wiley & Sons, 1978.

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

[5] R. Brownlie, J. Prowse, and M. S. Padke, “Robust testing of AT&T PMX/StarMAIL using OATS,” AT&T Technical Journal, vol. 71, no. 3, pp. 41–7, 1992.

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

[7] 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.

[8] 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 *Proceedings of the Intl. Conf. on Software Engineering, (ICSE)*, 1999, pp. 285–294.

[9] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino, “Applying design of experiments to software testing,” in *Proceedings of the Intl. Conf. on Software Engineering, (ICSE ’97)*, 1997, pp. 205–215.

[10] J. Grundy, Y. Cai, and A. Liu, “Generation of Distributed System Test-beds from High-level Software Architecture Description,” in *16 th International Conference on Automated Software Engineering, Linz Austria*. IEEE, Sept. 2001.

[11] W. Kolarik, *Creating Quality: Systems, Concepts, Strategies and Tools*. McGraw-Hill Education, 1995.

[12] A. S. Krishna, A. Gokhale, D. C. Schmidt, V. P. Ranganath, J. H. White, and D. C. Schmidt, “Model-driven Middleware Specialization Techniques for Software Product-line Architectures in Distributed Real-time and Embedded Systems,” in *Proceedings of the MODELS 2005 workshop on MDD for Software Product-lines*, Half Moon Bay, Jamaica, Oct. 2005.

[13] 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: ACM/IEEE, July 2004.

[14] 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: IEEE, May 2004.

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

[16] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai, “Composing Domain-Specific Design Environments,” *IEEE Computer*, pp. 44–51, Nov. 2001.

[17] Y. Liu, I. Gorton, A. Liu, N. Jiang, and S. Chen, “Designing a Test Suite for Empirically-based Middleware Performance Prediction,” in *40 th International Conference on Technology of Object-Oriented Languages and Systems, Sydney Australia*. Australian Computer Society, Aug. 2002.

[18] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son, “Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms,” *Real-Time Systems Journal*, vol. 23, no. 1/2, pp. 85–126, July 2002.

[19] R. Mandl, “Orthogonal Latin squares: an application of experiment design to compiler testing,” *Communications of the ACM*, vol. 28, no. 10, pp. 1054–1058, 1985.

[20] 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: IEEE/ACM, May 2004.

[21] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Hierarchical gui test case generation using automated planning," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 144–155, February 2001.

[22] T. Mitchell, "An algorithm for the construction of the 'd-optimal' experimental designs," *Technometrics*, vol. 16, no. 2, pp. 203–210, 1974.

[23] D. C. Montgomery, *Introduction to Statistical Quality Control*, 3rd ed. New York: John Wiley & Sons, 1996.

[24] Robert Ricci and Chris Alfred and Jay Lepreau, "A Solver for the Network Testbed Mapping Problem," *SIGCOMM Computer Communications Review*, vol. 33, no. 2, pp. 30–44, Apr. 2003.

[25] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

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

[27] J. W. Tukey, *Exploratory Data Analysis*. Reading, Massachusetts: Addison-Wesley, 1977.

[28] 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: ACM, Apr. 2004.

[29] B. White and J. L. et al, "An Integrated Experimental Environment for Distributed Systems and Networks," in *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*. Boston, MA: USENIX Association, Dec. 2002, pp. 255–270.

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

[31] C. Yilmaz, M. B. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Transactions on Software Engineering*, vol. 31, no. 1, pp. 20–34, Jan. 2006.

[32] ——, "Covering arrays for efficient fault characterization in complex configuration spaces." in *ISSTA*, 2004, pp. 45–54.

[33] C. Yilmaz, A. S. Krishna, A. Memon, A. Porter, D. C. Schmidt, A. Gokhale, and B. Natarajan, "Main effects screening: a distributed continuous quality assurance process for monitoring performance degradation in evolving software systems," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM Press, 2005, pp. 293–302.

[34] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu, "A Comparison of Empirical and Model-driven Optimization," in *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation*, June 2003.

[35] 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.

Appendix

The screening designs used in Section IV-C 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_{IV}^{14-8} with design generators $G = ABC$, $H = ABD$, $I = ABE$, $J = ACDE$, $K = ABF$, $L = ACDF$, $M = ACEF$, $N = ADEF$.
Scr_{128} is a 2_{IV}^{14-7} with design generators $H = ABC$, $I = ABDE$, $J = ABDF$, $K = ACEF$, $L = ACDG$, $M = ABEFG$, $N = BCDEFG$.

The screening designs used in Section IV-D were calculated using the SAS statistical package. (`www.sas.com`).
It is a 2_{VI}^{14-3} (2048-run) design with design generators $L = ABCDEFGHIJK$, $M = EFGHIJK$, and $N = CDGHIJK$.