Distributed Performance Testing using Statistical Modeling

Alan F. Karr
National Institute for Statistical Sciences (NISS)
PO Box 14006
Research Triangle Park, NC 27709–4006
karr@niss.org

Adam A. Porter
University of Maryland
College Park, MD 27042
aporter@cs.umd.edu

ABSTRACT

This article briefly presents some of our recent research in distributed continuous performance analysis. In general this work pushes substantial parts of performance analysis out of developer laboratories and onto remote, end-user machines.

To do this effectively we have found it useful to recast performance analysis as a model-based experimental design and execution problem.

Our experience suggests that this approach has merit, but that much future work remains to be done. We therefore discuss some of the limitations of our current efforts and describe some plans for future work.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Experimentation, Measurement, Performance, Statistical Modeling

Keywords

Distributed continuous quality assurance

1. PROJECT DESCRIPTION

Emerging trends and challenges. Software developers and users spend a great deal of time measuring, modeling, and analyzing the systems the build and use. Typically these investigations are performed in-house—by a small group of experts, on their own computing platforms, using input workloads they have generated. One benefit of in-house QA is that programs can be analyzed at a fine level of detail since QA teams have extensive knowledge and unrestricted access to the software. The shortcomings of in-house efforts, however, are well-known and severe. They include (1) increased cost and schedule; and (2) misleading results when the test-cases and input workload differs from actual inputs and workloads, or when the developer systems and execution environments differ from fielded ones.

We expect these shortcomings to worsen considerably in the future as software is increasingly subjected to the following trends:

1. Demand for user-specific customization. To serve a broad and diverse user community and to amortize development costs across it, software is increasingly designed be customizable for particular run-time contexts and application requirements. General-purpose, one-size-fits-all software solutions often have unacceptable performance.

2. Severe cost and time-to-market pressures. Global competition and market deregulation are shrinking budgets for the development and analysis of software in-house. In addition, customers are demanding that software be frequently and rapidly updated to add functionality, fix bugs and patch security flaws. As a result, developers must assess the consequences of development decisions and hunt down problems in ever shortening amounts of time.

3. Distributed and evolution-oriented development processes. Today's global IT economy, multi-tier architectures and component-based development techniques often involve developers distributed across geographical locations, time zones, and business organizations. (This is true for both open source and commercial projects.) This approach reduces cycle time by letting developers work in parallel, with minimal direct inter-developer coordination. At the same time though it can increase software churn rates and obscure system-wide constraints, assumptions and interactions. All this in turn increases the need to detect, diagnose, and fix faulty changes quickly. The same situation occurs in evolution-oriented processes, where many small increments are routinely added to the base system.

These accelerating trends present many challenges to developers. One of the biggest is the *exploding system configuration space*. Software that runs on many hardware and OS platforms and supports very different feature sets and workload profiles can have up to thousands of compile- and/or run-time configuration options. For example, SQL Server 7.0 has 47 configuration options, Oracle 9 has 211 initialization parameters, the ACE+TAO CORBA implementation has over 500 configuration options, and a recent Linux kernel has nearly 5000. While not all options are independent—they often constraint one another —this nevertheless leads to a great many system configurations, each of which has its own performance characteristics.

When growing configuration spaces are coupled with shrink-

ICSE Workshop

ing software development resources, it becomes infeasible to handle all performance analysis in-house. One reason is individual developers lack access to all the *heterogeneous hardware, OS, and compiler platforms* on which their software will run. Even with access to resources, it simply takes too long to get useful feedback. The end result is that developers must often release software whose performance in many configurations is unknown.

Moreover, distributed development teams exacerbate these trends because it becomes hard to maintain a system-wide perspective. As a result, *design and optimization decisions are made without precise knowledge of their consequences in fielded systems.*

These trends and their resulting challenges create an environment in which performance analyses done in-house do not match in-the-field behavior. We contend that techniques and tools needed to bridge this gap effectively will save substantial time, effort, and resources throughout the software industry.

To address these problems we are developing tools and techniques for Distributed, Continuous Quality Assurance (DCQA). In particular we are focusing on DCQA processes that conduct performance analyses around-the-world and around-the-clock, leveraging fielded resources during local, off-peak hours. To support this effort we are devising and/or redesigning performance analyses so they are (1) *highly distributed and lightweight* from the perspective of individual participants; (2) *incremental and adaptive*, changing their behavior over time based on earlier results, time constraints and resource availability; and (3) *robust* to differences in end user resources, such as processor speeds, memory availability, transient workloads and network traffic.

To accomplish these steps effectively, we depend on models to help solve the following technical challenges:

Process management. We are developing the tool infrastructure necessary to conduct distributed analyses on production user programs. This infrastructure includes mechanisms for deploying fielded instances, executing and measuring each instance's performance, acquiring data from them, and storing and analyzing the data. It also involves placing hooks in individual systems and modifying development tools, such as configuration management and bug-tracking systems.

Distributed compositional analysis techniques. Since each user provides only a small amount of the total data, traditional analysis that is localized to an individual user machine will not work. We are therefore developing new dynamic analysis techniques that decompose the analysis into smaller steps and use state-of-the-art statistical techniques to distribute the steps among multiple users and then to fuse each user's results into an accurate solution to the original problem.

Distributed data management. Initially, our approach has distributed performance measurement tasks across a grid of computing resources. The data are returned to central collection sites and analyzed there. This approach will have problems scaling to very large number of participants. To deal with these issues, we are redeveloping the infrastructure and analysis techniques so that not only measurement, but data management and analysis are also distributed.

Scenario implementation and empirical evaluation. To help explore these ideas we are developing specific usage scenarios—applications that depend on performance measurement to achieve some developer or user goal. We are also evaluating how well our approach supports these applications to better understand the limits of our approach.

2. PRELIMINARY INVESTIGATIONS

2.1 The Skoll DCQA Environment

As explained earlier, we are exploring *distributed continuous quality assurance* processes. DCQA processes work by dividing high level QA tasks, such as running a regression test suite, into multiple subtasks, such as running a subset of the regression test suite on a particular system configuration. These subtasks are then intelligently and continuously distributed to—and executed by—heterogeneous 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.

To support this effort we have developed *Skoll*, a prototype DCQA environment described at `www.cs.umd.edu/projects/skoll`. Currently Skoll includes 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 system configuration space, which captures different control parameters, called options, and their allowable settings. Since in practice not all combinations of options make sense (for instance, 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. 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 DCQA processes can be quite large, Skoll has an *Intelligent Steering Agent* (ISA) that uses planning techniques to control DCQA processes by deciding which valid configuration to allocate to each incoming Skoll client request. When a client is available, the ISA decides which subtask to assign it by considering many factors, including (1) *the system 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*: for example, 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 (for example, developer-supplied regression or 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. The ISA can learn from the

results and adapt the process: if some configurations fail to work properly, developers may either want to pinpoint the source of the problems or, alternatively, to 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 [10, 16], for example, we used CTA to diagnose options and settings that were the likely causes of specific test failures. In more recent work we developed statistical tools to design and analyze formal experiments.

2.2 A Sample DCQA Process for Performance-Oriented Regression Testing

As software systems change, developers often run regression tests to detect unintended functional side effects. Much less attention has been directed to unintended side effects on performance. To detect performance problems, developers often run benchmarking regression tests. As described in Section 1, however, in-house QA efforts cannot deal with enormous system configuration spaces, especially 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 found that only a small number of default configurations are benchmarked routinely by the core ACE+TAO development team, who thus get an extremely limited view of their middleware's Quality of Service (QoS). Problems not readily seen in these default configurations therefore often escape detection until systems based on ACE+TAO are fielded by end users.

To address these problems, we used the Skoll environment to develop and implement a new hybrid DCQA process called *main effects screening* (See Yilmaz et al. [3] for a more detailed discussion).

2.2.1 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* [15], which are highly economical and can reveal important *low-order effects*, such as individual options and pairs or triples of options 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, in order to identify the main effects; (4) estimate overall performance whenever the software changes by evaluating all combinations of the main effects (defaulting or randomizing all other options); and (5) continuously recalibrate the main effects options by restarting the process. This last step is done because 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.

2.2.2 Feasibility Study

We evaluated this new process by applying it to a large-scale system called ACE+TAO[13]. The study focused on the following 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. Thus, there are $2^{14} = 16,384$ different configurations.

To evaluate our approach, we generated performance data for all 16,384 valid configurations, which we refer to as the "full suite" and the performance data as the "full data set." We then examined the effect of each option and judged whether they had important effects on performance. Based on this data we determined that two options were clearly important, another three were arguably important, and the remaining options were not important.

We then evaluated whether the remotely executed screening designs correctly identified the same important options discovered in the full data set. To do this, we calculated and executed three different screening designs. 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. We determined that all screening designs correctly identified the two important options. Scr_{128} also identified the three possibly important options.

These results suggest that (1) screening designs can, in some cases, detect important options at a small 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.

Next, we evaluated whether benchmarking all the combinations of the most important options can be used to estimate performance quickly across the entire configuration space. The estimates are generated by examining all combinations of the most important options, while randomizing the settings of the unimportant options. Based on this data we determined that the distributions of performance for the five most important and two most important options screening suites closely track the overall performance data. This data suggests that the screening suites computed at step 4 of the main effects screening process can be used to estimate overall performance in-house at extremely low time/effort, (running 4 benchmarks takes 40 seconds, running 32 takes 5 minutes, running 16,384 takes two days).

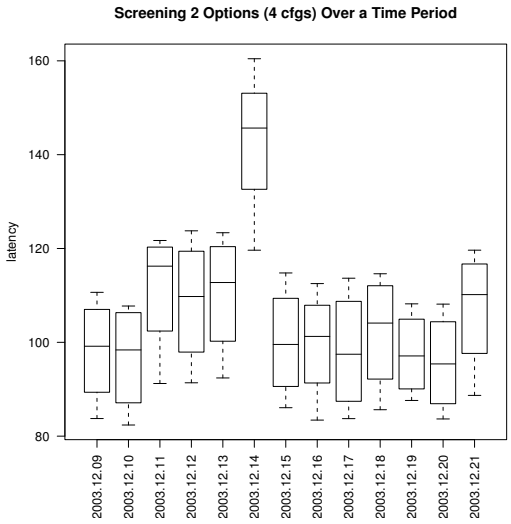

Figure 1: Performance estimates across time.

Finally, we evaluated the primary goal of the main effects screening which is to quickly detect performance degradations in evolving systems. To better understand this issue, we measured latency on the top-2 screening suite, once a day, using CVS snapshots of ACE+TAO+CIAO.

Figure 1 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 of nearly 50%. Further analysis by system developers indicated that their unsystematic testing failed to evaluate configurations where the degradation was much more pronounced.

3. SOME TECHNICAL CHALLENGES

Despite our initial successes, much work remains to be done. In particular, we must improve our modeling approaches and execution infrastructure, develop new applications, and better handle practical issues caused by platform heterogeneity and other unobservable factors.

Modeling the QA subtask space: We formally represent those aspects of the QA subtasks and underlying software that will be varied under control of the DCQA process. This includes not only process and software configuration parameters, but also the constraints among them. Currently, we use a "flat" representation in which all options are assumed to interact. We intend to explore hierarchical models and multiple-component models where we know *a priori* that groups of options don't interact. We will also explore techniques for learning whether options interact.

Modeling unobservables: Many characteristics of Skoll client machines are not part of the data transmitted to the server. Reasons range from privacy to impossibility of measurement. Examples may include processor speed, memory, library versions and what other programs are running at the same time. Statistical modeling is necessary in order to represent the effects of unobservables.

Scaling: Some unobservables may modeled as scaling factors on performance, which allows collection of data about them. See the discussion of **Calibration** below.

Space-filling experimental designs: Space-filling designs have been used successfully for analysis of large-scale computer models [12], where the high dimensionality of input spaces is a significant issue. They should be applicable as well to high-dimensional configuration spaces, but will require adaptation in order to deal with constraints.

Sequential and adaptive experimental designs: For scenarios such as those discussed below, adaptive exploration of the configuration space may be particularly effective. One strategy, used by NISS in other settings, is proceed sequentially, beginning with a space-filling design to ensure coverage, and following up with other designs targeted at regions in the configuration space "where the action is."

Borrowing strength: Since QA tasks are assigned to remote machines, volunteered by end users, it is difficult to predict the availability of resources. Moreover, some volunteer may wish to maintain some control of how their resources will be used; for example limiting which version of a system can undergo QA on their resources. In such cases, it is impossible to pre-compute QA subtask schedules. Therefore, we have to develop scheduling techniques that adapt based on a variety of factors including resource availability.

Distributed data management. Currently, all performance data are returned to central collection sites for analysis. We foresee scalability problems and are therefore investigating more distributed approaches (inspired by peer-to-peer algorithms) in which control and data storage are distributed to end users machines.

4. PLANNED APPLICATION SCENARIOS

We plan to drive our research by focusing on three scenarios. The purpose of each scenario is to reliably choose settings, or ranges of settings, for configuration options to achieve a particular software engineering goal. At a high level these goals include: (1) performance-oriented regression test selection—determining whether a recent change has caused an unintended performance degradation; (2) performance optimization and variability reduction—determining option settings that maximize performance or minimize performance variability; and (3) requirements satisfaction—determining a range of option settings over which several performance criteria can be simultaneously satisfied.

4.1 Scenario 1: Performance-oriented Regression Testing

As we mentioned earlier, when developers change software they need to ensure that performance is not degraded. The size of the configuration space precludes benchmarking all configurations. We are iteratively extending our earlier work to handle a set of practical concerns that so far have been ignored.

We will allow multiple hardware and OS platforms, not just one. We will model and account for unobservable machine-specific differences that our current approach cannot handle. In practice these unobservable characteristics would be factors such as processor speeds, memory size, other applica-

tions and network traffic.

We will modify the approach to use space-filling design techniques and later adaptive designs as well to allow for faster feedback. This is important when individual QA subtasks are computationally expensive (Downloading ACE+TAO from CVS, fully building it, compiling and executing all tests can take 6-8 hours on a typical developer's workstation.) or when developers have severe time to respond requirements (for example, when a system release date is approaching and many bugs fixes are being done.)

To formulate the approach in more detail, let

- C = configuration space (which is big, involves constraints, ...)
- O = space of observables for instances (example: OS)
- Θ = generic space of unobservables (example: baseline speed or load, memory)
- P = performance measure. Assume that larger values are better, so P could be (1 / running time).

All of C, O, Θ are high-dimensional.

The basic statistical model is for P_i = measured performance for instance i:

$$P_i = f(C_i, O_i, \theta_i) + \varepsilon_i, \tag{1}$$

where

- $C_i \in C$ = configuration for instance i
- $O_i \in O$ = observables for instance i, treated as a known constant
- $\theta_i \in \Theta$ = unobservables for instance i, treated as a random variable
- ε_i = measurement error.

This is a nonparametric *random effects* model. The goal is to estimate f, which relates performance to configuration, observables and unobservables. Statistical technology for such models is available, but will require adaptation to deal with the dimensionality and size of C, O and Θ.

As an entry point and proof of concept, we will

1. Start with a space-filling design (for example, a Latin hypercube) in C. This will be challenging because most methods for producing space-filling designs assume that C is a product space.
2. Using the space-filling design, estimate the model (1) using Bayesian techniques for random effects models. This step produces an estimator $\hat{f}$. It is challenging because of the high dimensionality of C, O and Θ and the relative paucity of data.

Other challenges include:

Selection of performance measures: How many measures, and which ones, are needed in a particular setting?

Calibration, which is one way of obtaining information about the unobservables O_i. Suppose that P is running time and one component of each θ_i is baseline speed/load. Suppose that prior to running the "real software" we run a piece of benchmark software, and record the running time of each. Then the scaled performance measure

$$P = \frac{\text{Running time of real software}}{\text{Running time of benchmark software}} \tag{2}$$

may remove the need for an unobservable θ representing speed/load. This measure changes the direction of performance, but that can be fixed. The point is that scaling could decrease the dependence of the model on unobservables.

Model validation: How do we assess how well our method works? Standard techniques, such as cross-validation, may require too much data.

A more challenging version—but necessary because of size and complexity of C—is to make the entire approach adaptive, selecting the C_i based on results of previous tests. Issues include:

- How to do the selection. NISS has worked on how to use space-filling designs to guide more detailed sequential search. These may be too complex, partly because fitting the model for every new data point may be too inefficient. A less daunting alternative is to think of dividing a big (space-filling or other) design into say 20 parts, execute one, fit the model, choose which to execute next, ... Especially if the full design is infeasible, this is interesting because it lets the data help determine which parts of the full design are actually done.
- Some C_i are now random, because they depend on previous data. This complicates the analysis.
- "Weird" configurations are actually a problem throughout. Should designs allow configurations that satisfy constraints but would never be used in reality?

4.2 Scenario 2: Optimization and Variability Control

In this scenario we envision developers who are building their system on top of one or more components. We assume that the underlying components must be customized to support the requirements of the overall application. The system developers need help in deciding how to do this.

In this scenario, component developers will provide multiple benchmarks (input drivers with different workload characteristics) which will be run as part of a DCQA process. The results of these runs will populate a large performance database.

Users will select which of the benchmarks resemble their application requirements and indicate which options are fixed by their application and which are free.

We will investigate techniques for both optimizing mean performance and for controlling variability. The goal is to find the configuration that optimizes performance for a given set O_0 of observables. If f were known, this would require solving an optimization problem, two forms of which are:

Average case performance: If performance averaged over values of the unobservables is of principal interest, then the optimization problem is

$$C^*(O_0) = \arg\max_C \int f(C, O_0, \theta) p(\theta) d\theta, \tag{3}$$

where α is some performance threshold and p some density—in particular, the posterior distribution of θ from the Bayesian analysis.

There are a numerous computational challenges here, including the numerical integration, difficulties in calculating p, and the complexity of C and O.

Worst case performance: If worst case performance over all values of the unobservables is the criterion, then the optimization problem is

$$C^*(O_0) = \arg\max_C \min_\theta f(C, O_0, \theta). \quad (4)$$

4.3 Scenario 3: Multi-Objective Constraint Satisfaction

In this scenario developers of a component-based application want to quickly find configurations that fail to satisfy a set of performance requirements or, alternatively, find a set of conforming configurations. We will assume that each component developer publishes its configuration model.

The focus here is on configurations with bad performance, which is relevant if, for example, the software must meet performance requirements such as speed or memory footprint. Let B^* be the set of (configuration, observable) pairs with bad performance. Two possible definitions (Recall that higher P is better performance.) are:

Average performance: If performance averaged over values of the unobservables is of principal interest, then

$$B^* = \left\{ (C, O) : \int f(C, O, \theta) p(\theta) d\theta \leq \alpha \right\}, \quad (5)$$

where α is some performance threshold and p some density—in particular, the posterior distribution of θ from the Bayesian analysis.

There are the same computation challenges here as for Scenario 2.

Worst case performance: If worst case performance over all values of the unobservables is the criterion, then

$$B^* = \left\{ (C, O) : \min_\theta f(C, O, \theta) \leq \alpha \right\}. \quad (6)$$

The approach will be to estimate the set B^*, and figure out what is driving its structure. The principal steps are:

1. Construct estimator $\widehat{B^*}$ of B^*. The naive way to do this is to substitute $\hat{f}$ for f in (5) or (6). Even with this, characterizing uncertainties in $\widehat{B^*}$ is a challenge. More sophisticated approaches use "better" estimators $\widehat{B^*}$.
2. Use $\hat{f}$ and $\widehat{B^*}$ to understand (configuration, observable) effects on performance. For example, run a classifier with membership in $\widehat{B^*}$ treated as a binary response.

5. RELATED WORK

Our proposed work is related to and draws on existing research in several areas including applying DOE to software engineering, feedback-based optimization techniques, and large-scale testbed environments.

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 experiments (DoE) theory within software engineering has mostly been limited to *interaction testing*, largely to compute and sometimes generate minimal test suites that cover all combinations of specified program inputs. Some examples of this work include Dalal *et al.* [5], Burr *et al.* [2], Dunietz *et al.* [6], and Kuhn *et al* [8]. Yilmaz *et al.* [16] used covering arrays as a configuration space sampling technique to support the characterization of failure-inducing option settings.

Program optimization. Off-line analysis 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.

On-line analysis, where feedback control is used to dynamically adapt QoS measures. An example of online analysis is the ControlWare middleware [17], 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 [9] to automatically adjust the rate of remote operation invocation transparently to an application.

Hybrid analysis combines aspects of off-line and on-line 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.

Large-scale benchmarking testbeds. EMULab [14] 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, by modeling the underlying OS, hardware, and communication links. This topology is then mapped [11] to ~250 physical nodes that can be accessed via the Internet. The EMULab tools can generate script files that use the Network Simulator (NS) (`http://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 single testbed, but instead can leverage the large amounts of end-user computer resources in the Skoll grid. Moreover, the BGML model interpreters can generate NS scripts to integrate our benchmarks with experiments in EMULab.

Skoll can enhance 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.

6. ACKNOWLEDGMENTS

This material is based on work supported by the US National Science Foundation under NSF grants CCR-0312859,

CCR-0205265, CCR-0098158, CCF-0205118 and CCF-0447864. as well as funding from BBN Technologies, Lockheed Martin, Raytheon, and Siemens.

7. REFERENCES

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

[2] 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.

[3] Cemal Yilmaz, Arvind Krishna, Atif Memon, Adam Porter, Douglas Schmidt, and Aniruddha Gokhale. Main effects screening: A distributed continuous quality assurance process for monitoring performance degradation in evolving software systems. In *International Conference on Software Engineering*, St. Louis, MO, May 2005. IEEE/ACM.

[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] 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.

[9] 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.

[10] 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.

[11] Robert Ricci and Chris Alfred and Jay Lepreau. A Solver for the Network Testbed Mapping Problem. *SIGCOMM Computer Communications Review*, 33(2):30–44, Apr. 2003.

[12] J. Sacks, W. J. Welch, T. J. Mitchell, and H. P. Wynn. Design and analysis of computer experiments. *Statistical Science*, 4:409–435, 1989.

[13] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, and C. Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), Feb. 2002.

[14] 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*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

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

[16] 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.

[17] 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.