An Empirical Study of Profiling Strategies for Released Software and their Impact on Testing Activities

Sebastian Elbaum and Madeline Hardojo
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE
(elbaum,mhardojo)@cse.unl.edu

ABSTRACT

An understanding of how software is employed in the field can yield many opportunities for quality improvements. Profiling released software can provide such an understanding. However, profiling released software is difficult due to the potentially large number of deployed sites that must be profiled, the extreme transparency expectations, and the remote data collection and deployment management process. Researchers have recently proposed various approaches to tap into the opportunities and overcome those challenges. Initial studies have illustrated the application of these approaches and have shown their feasibility. Still, the promising proposed approaches, and the tradeoffs between overhead, accuracy, and potential benefits for the testing activity have been barely quantified. This paper aims to overcome those limitations. Our analysis of 1200 user sessions on a 155 KLOC system substantiates the ability of field data to support test suite improvements, quantifies different approaches previously introduced in isolation, and assesses the efficiency of profiling techniques for released software and the effectiveness of their associated testing efforts.

Categories and Subject Descriptors: D.2.5: Testing tools; Tracing.

General Terms: Experimentation, Reliability, Verification.

Keywords: Profiling, instrumentation, software deployment, testing, empirical studies.

1. INTRODUCTION

Software test engineers cannot predict, much less exercise, the overwhelming number of potential scenarios faced by their software. Instead, they allocate their limited resources based on assumptions about how the software will be employed after release. Yet, the lack of connection between in-house activities and how the software is employed in the field can lead to inaccurate assumptions, resulting in decreased software quality and reliability over the system's lifetime. Even if estimations are initially accurate, isolation from what happens in the field leaves engineers unaware of future shifts in user behavior or variations due to new environments until too late.

Approaches integrating in-house activities with field data appear capable of overcoming such limitations. These approaches must profile field data to continually assess and adapt quality assurance activities, considering each deployed software instance as a source of information. The increasing software pervasiveness and connectivity levels[1] of a constantly-growing pool of users coupled with these approaches offers a unique opportunity to gain a better understanding of the software's potential behavior.

Early commercial efforts have attempted to harness this opportunity by including built-in reporting capabilities in deployed applications that are activated in the presence of certain failures (e.g, Software Quality Agent from Netscape [18], Traceback [13], Microsoft Windows Error Reporting API). More recent approaches, however, are designed not only to leverage the deployed software instances throughout their execution, but also to consider the levels of transparency required when profiling users' sites, the management of instrumentation across the deployed instances, and issues that arise from the large scale data collection. For example:

- The Perpetual Testing project produced the residual testing technique to reduce the instrumentation based on previous coverage results [23, 25].
- The EDEM prototype provided a semi-automated way to collect user-interface feedback from remote sites when it does not meet an expected criterion [11].
- The Gamma project introduced an architecture to distribute and manipulate instrumentation across deployed software instances [10].
- The Skoll project presented an architecture and a set of tools to distribute different job configurations across users [27].

Although these efforts present reasonable conjectures, we have barely begun to quantify their potential benefits and costs. Most publications have illustrated the application of isolated approaches [11], introduced supporting infrastructure [27], or explored a technique's feasibility under particular scenarios (e.g., [9, 14]). (Previous empirical studies

[1]35 million Americans had broadband Internet access in 2003 [19].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA'04, July 11–14, 2004, Boston, Massachusetts, USA.
Copyright 2004 ACM 1-58113-820-2/04/0007 ...$5.00.

are summarized in Section 2.5.) Given that feasibility has been shown, we must now quantify the observed tendencies, investigate the tradeoffs, and explore whether the previous findings are valid at a larger scale. This paper presents a family of three empirical studies that quantify the efficiency and effectiveness of profiling strategies for released software. The studies also assess several techniques that employ field data to drive test suite improvements and identify factors that can affect the potential gains.

In the following section, we abstract the essential attributes of existing profiling techniques for released software to organize them in strategies and summarize the results of previous empirical studies. Section 3 describes the research questions, object preparation, design and implementation, metrics, and potential threats to validity. Section 4 presents results and analysis. Section 5 provides additional discussion and conclusions.

2. PROFILING STRATEGIES FOR RELEASED SOFTWARE

Researchers have enhanced the efficiency of profiling techniques through several mechanisms: (1) performing up-front analysis to optimize the amount of instrumentation required [3], (2) sacrificing accuracy by monitoring entities of higher granularity or by sampling program behavior [7, 8], (3) encoding the information to minimize memory and storage requirements [24], and (4) repeatedly targeting the entities that need to be profiled [1]. The enumerated mechanisms gain efficiency by reducing the amount of instrumentation inserted in a program through the analysis of the properties it exhibited in a controlled in-house environment.

Profiling techniques for released software [10, 11, 23, 27], however, must consider the efficiency challenges and the new opportunities introduced by a remote and potentially large user pool. This paper investigates three profiling strategies designed to work on released software. The strategies abstract the essential elements of existing techniques helping to provide an integrated background, and also facilitating formalization, analysis of tradeoffs, and comparison of existing techniques and their implementation.

The next section describes the *full* strategy which constitutes our baseline, the following three sections describe the profiling strategies for released software, and Section 2.5 summarizes the related empirical studies.

2.1 Full Profiling

Given a program P and a class of events to monitor C, this approach generates P' by incorporating instrumentation code into P to enable the capture of ALL events in C and a transfer mechanism T to transmit the collected data to the organization at the end of each execution cycle (e.g., after each operation, when a fatal error occurs, or when the user terminates a session).

Capturing all the events at all user sites demands extensive instrumentation, increasing program size and execution overhead (reported to range between 10% to 390% [2, 16]). However, this approach also provides the maximum amount of data for a given set C serving as a baseline for the analysis of the other three techniques specific to release software. We investigate the raw potential of field data through the *full* technique in Section 4.1.

2.2 Targeted Profiling

Before release, software engineers develop an understanding about the program behavior. Factors such as software complexity and schedule pressure limit the levels of understanding they can achieve. This situation leads to different levels of certainty about the behavior of program components. For example, when testers validate a program with multiple configurations, they may be able to gain certainty about a subset of the most used configurations. Some configurations, however, might not be (fully) validated.

To increase profiling transparency in deployed instances, software engineers may just target the components for which the behavior is not sufficiently understood. Following with our example about a software with multiple configurations, engineers aware of the overhead associated with profiling deployed software could aim to profile just those least understood or most risky configurations.

More formally, given program P, a class of events to profile C, a list of events observed (and sufficiently understood) in-house $C_{observed}$ where $C_{observed} \subset C$, this technique generates a program P' with additional instrumentation to profile all events in $C_{targeted} = C - C_{observed}$. Observe that $|C_{targeted}|$ determines the efficiency of this technique by reducing the necessary instrumentation, but also bounding what can be learned from the field instances. As the certainty about the behavior of the system or its components diminishes, $|C_{observed}| \rightarrow 0$, this strategy's performance approximates *full*.

As defined, this strategy includes the residual testing technique [23] where $C_{observed}$ corresponds to statements observed in-house, and it also includes the distribution scheme proposed by Skoll [27] where $C_{targeted}$ corresponds to target software configurations that require field testing.

2.3 Profiling with Sampling

Statistical sampling is the process of selecting a suitable part of a population for determining the characteristics of the whole population. Profiling techniques have adapted the sampling concept to reduce execution costs by repeatedly sampling across space and time. Sampling across space consists of profiling a subset of the events following a certain criterion (e.g., hot paths). Sampling across time consists of obtaining a sample from the population of events at certain time intervals [4]. Common profiling utilities like *gprof* follow this approach, stopping execution at fixed intervals to determine where the cycles are being spent [8].

When considering released software, we find an additional sampling dimension: the instances of the program running in the field. We could, for example, sample across the population of deployed instances, profiling the behavior of a group of users. This is advantageous because it would only add the profiling overhead to a subset of the population. Still, the overhead on this subset of instances could substantially affect user's activities, biasing the collected information.

An alternative sampling mechanism could consider multiple dimensions. For example, we could stratify the population of events to be profiled following a given criterion (e.g., events from the same functionality) and then sample across the subgroups of events, generating a version of the program with enough instrumentation to capture just those sampled events. Then, by repeating the sampling process, different versions of P' can be generated for distribution. Potentially,

each user could obtain a slightly different version that aims to capture a particular sample of events at each site.[2]

More formally, given program P, a class of events to monitor C, the stratified sampling strategy identifies s strata $C_1, C_2, ..., C_s$, selects a total of N events from C by randomly picking n_i events per strata, where i varies from 1 to s and n_i is proportional to the stratum size, and it generates P' to capture the selected events. As N gets smaller, P' contains less instrumentation, enhancing transparency but possibly sacrificing accuracy. By repeating the sampling and generation process, $P'', P''', ..., P^m$ are generated, resulting in versions with various suitable instrumentation patterns available for deployment.[3]

There is an important tradeoff between the event sample size N and the number of deployed instances. Maintaining N constant while the number of instances increases results in constant profiling transparency across sites. As the number of deployed instances increases, however, the level of overlap in the collected data across deployed sites is also likely to increase. We could then leverage this overlap to reduce N, gaining transparency at each deployed site by collecting less data while compensating by profiling more deployed instances. Section 4.2 investigates this alternative.

Also note that, as defined, our sampling strategy provides a statistical perspective for various algorithms implemented by the Gamma research effort [14, 22], incorporating and formalizing the procedure to jointly sample deployed sites and events, but not considering the redistribution of versions across sites.

2.4 Trigger Data Transfers on Anomalies

Transferring data from deployed sites to the organization can be costly for both parties. For the user, data transfer implies at least additional computation cycles to marshal and package data, bandwidth to actually perform the transfer, and a likely decrease in transparency. For an organization with thousands of deployed software instances, data collection might become a bottleneck. Even if this obstacle could be overcome with additional collection devices (e.g., a cluster of collection servers), processing and maintaining such a data set could prove expensive. Triggering data transfers in the presence of anomalies can help to reduce these costs.

Employing anomaly detection implies the existence of a baseline behavior considered nominal or normal. When targeting released software, the nominal behavior is defined by what the engineers know or understand. For example, engineers could define an operational profile based on the execution probability exhibited by a set of beta testers [17]. A copy of this operational profile could be embedded into the released product so that deviations from its values trigger a data transfer. Sessions that fit within the operational profile would only send a confirmation to the organization, increasing the confidence in the estimated profile; sessions that fall outside an specified operational profile range are completely transferred to the organization for further analysis (e.g., determine whether the anomaly indicates a potential problem, update the profile if the anomaly was due to an incomplete in-house assessment).

We now define the approach more formally. Given program P, a class of events to monitor C, an in-house characterization of those events C_{house}, a tolerance to deviations from the in-house characterization $C_{houseTolerance}$, this technique generates a program P' with additional instrumentation to monitor events in C, and a detection algorithm to identify when field behavior C_{field} deviates from $[C_{house} \pm C_{houseTolerance}]$. When such deviation is detected, session data is transferred to the organization. Note that this definition of trigger by anomaly includes the type of behavioral "mismatch" trigger mechanism used by EDEM [11] by making $C_{houseTolerance} = 0$.

There are many interesting tradeoffs in defining and detecting deviations from C_{house}. For example, there is a tradeoff between the level of investment on the in-house software characterization and the number of false negatives reported from the field. Also, there are myriad of algorithms to detect anomalies, trading detection sensitivity and effectiveness with execution overhead. We investigate some of these tradeoffs in Section 4.3.

2.5 Previous Empirical Studies

The efforts to develop profiling techniques for released software have been supported by different mechanisms.

Hilbert and Redmiles [11, 12] utilized scenarios to demonstrate the concept of internet mediated feedback. The scenarios illustrated how software engineers could improve user interfaces through the collected information. The scenarios reflected the authors' experiences in a real context, but they did not constitute empirical studies.

The Skoll group has also started to perform studies to study the feasibility of their infrastructure on two large open source projects (ACE and TAO) [27]. The feasibility studies reported on the infrastructure's capability in detecting failures in several configuration settings. Again, no empirical evaluation has yet been made available.

Pavlopoulou and Young empirically evaluated the efficiency gains of the residual testing technique on programs of up to 4KLOC [23]. The approach considered instrumentation probe removal, where a probe is the snippet of code incorporated into P to profile a single event. Executed probes were removed after each test was executed, showing that instrumentation overhead to capture coverage can be greatly reduced under certain conditions (e.g., similar coverage patterns across test cases, non-linear program structure). Although incorporating field data into this process was discussed, this aspect was not empirically evaluated.

The Gamma group performed at least two empirical studies to validate the efficiency of their distributed instrumentation mechanisms. Bowring et al. [14] studied the variation in the number of instrumentation probes and interactions, and the coverage accuracy for two deployment scenarios. For these scenarios, they employed a 6KLOC program and simulated users with synthetic profiles. A second study by the same group of researchers lead by Orso [21] employed the created infrastructure to deploy a 60KLOC system and gathered profile information on 11 users (7 from the research team) to collect 1100 sessions. The field data was then used for impact analysis and regression testing improvement. The findings indicate that field data can provide smaller impact

[2]We could also perform the complement by stratifying the user population and proceeding as specified. However, finding subgroups of users might be difficult in the presence of new or shifting user populations.

[3]In this work we focused on two particular dimensions: space and deployed instances. However, note that sampling techniques on time could be applied on the released instances utilizing the same mechanism.

sets than slicing and truly reflect the system utilization, while sacrificing precision. The study also highlighted the potential lack of accuracy of in-house estimates, which can also lead to a more costly regression testing process.

Overall, when revisiting the previous studies in terms of the profiling strategies we find that: 1) the transfer on anomaly strategy has not been validated through empirical studies, 2) the targeted profiling strategy has not been validated with deployment data and its efficiency analysis included just four small programs, 3) the strategy involving sampling has been validated more extensively in terms of efficiency but the effectiveness measures were limited to coverage, and 4) each assessment was performed in isolation.

Our studies address those weaknesses by improving on the following items:

- Target object and subjects. Our empirical studies are performed on a 155KLOC program utilized by 30 users, providing the most comprehensive setting yet to study this topic.
- Comparison and integration of techniques with the same context. We analyze the benefits and costs of field data obtained with full instrumentation and compared it against techniques utilizing targeting profiling, sampling profiling, a combination of targeting and sampling, and anomaly driven transfers.
- Assessment of effectiveness. We measure coverage obtained by field instances but we also use field data to generate test cases and measure their coverage. Furthermore, we utilize the generated test suites on later versions of the program to quantify faults detection effectiveness.
- Assessment of efficiency. In addition to counting the number of instrumentation probes, we measure the number of necessary data transfers (a problem highlighted but not quantified in [11]).

3. EMPIRICAL STUDY

This section introduces the research questions that serve to scope this investigation. The metrics, object of study, and the design and implementation follows. Last, we identify the threats to validity.

3.1 Research Questions

We are interested in the following research questions.

RQ1: What is the potential benefit of profiling deployed software instances? In particular, we investigate the coverage and fault detection effectiveness gained through the generation of a test suite based on field data.

RQ2: How effective and efficient are profiling techniques designed to reduce overhead at each deployed site? We investigate the tradeoffs between efficiency gains (as measured by the number of probes required to profile the target software), coverage gains, and data loss.

RQ3: Can anomaly based triggers reduce the number of data transfers? What is the impact on the potential gains? We investigate the effects of triggering transfers when a departure from an operational profile is detected.

3.2 Metrics

Throughout the studies we mainly employ functional coverage to measure the potential benefit of field data obtained through the implemented profiling techniques for released software. We made a decision to capture functional level data in the field because of its relative low overhead (see Section 3.3). In two of the studies we also indirectly quantify the effectiveness of these profiling techniques by measuring the coverage (function and block level) and fault detection capabilities of test suites generated with field data.

We utilize two measures to to quantify the efficiency of profiling techniques. First, we count the number of instrumentation probes per deployed instance. Second, we count the number of transfers necessary to collect the field data.

3.3 Object

We selected the popular[4] program *Pine* (Program for Internet News and Email) as the object of the experiment. *Pine* is one of the numerous programs to perform mail management tasks. It has several advanced features such as support for automatic incorporation of signatures, internet newsgroups, transparent access to remote folders, message filters, secure authentication through SSL, and multiple roles per user. In addition, it supports tens of platforms and offers flexibility for a user to personalize the program by customizing configuration files.

Several versions of *Pine* source code are publicly available. For our study we primarily use the Unix build, version 4.03, which contains 1373 functions and 155,037 lines of code including comments.

Test Suite. To evaluate the potential of field data to improve the in-house testing activity we required an initial test suite on which improvements could be made. Since *Pine* does not come with a test suite, two graduate students, who were not involved in the current study, developed a suite by deriving requirements from *Pine*'s man pages and user's manual, and then generating a set of test cases that exercised the program's functionality. Each test case was composed of three sections: 1) a setup section to set folders and configurations files, 2) a set of Expect commands [15] that allows to test interactive functionality, and 3) a cleanup section to remove all the test specific settings. The test suite consisted of 288 automated test cases, containing an average of 34 Expect commands. The test suite took an average of 101 minutes to execute on a PC with an Athlon 1.3 processor, 512MB of memory, running Redhat version 7.2. When function and block level instrumentation was inserted, the test suite execution required 103 and 117 minutes respectively (2% and 14% overhead). Overall, the test suite covered 835 functions.

Faults. To quantify potential gains in fault detection effectiveness we required the existence of faults. We leveraged a parallel research effort by our team [5] that had resulted in 43 seeded faults in four posterior versions of *Pine*: 4.04, 4.05, 4.10, and 4.20. (The seeding procedure follows the one detailed in [6].) We then utilized these faults to quantify the fault detection effectiveness of the test suites that leveraged field data.

[4] *Pine* had 23 million users worldwide as of March 2003 [20].

3.4 Study Design and Implementation

The overall empirical approach was driven by the research questions and constrained by the costs of collecting data for multiple deployed instances. Throughout the study design and implementation process, we strived to achieve a balance between the reliability and representativeness of the data collected from deployed instances under a relatively controlled environment, and the costs associated with obtaining such a data set. As we shall see, combining a controlled deployment and collection process with aposteriori simulation of different scenarios and techniques helped to reach such a balance (potential limitations of this approach are presented under threats to validity in Section 3.5).

We performed the study in three major phases: (1) object preparation, (2) deployment and data collection, and (3) processing and simulation.

The first phase consisted of instrumenting *Pine* to enable a broad range of data collection. The instrumentation is meant to capture functional coverage information, operational traces, accesses to environmental variables, and changes in the configuration file occurring in a single session (a session is initiated when the program starts and finishes when the user exits). In addition, to enable further validation activities (e.g., test generation based on user's session data), the instrumentation also enables the capture of various session attributes associated with user operations (e.g., folders, number of emails in folders, number and type of attachments, errors reported on input fields). At the end of each session, the collected data is time-stamped, marshaled, and transferred to the central repository. For anonimization purposes, the collected session data is packaged and labeled with the encrypted sender's name at the deployed site and the process of receiving sessions is conducted automatically at the server to reduce the likelihood of associating a data package with its sender.

We conducted the second phase of the study in two steps. First, we deployed the instrumented version of *Pine* at five "friendly" sites. For two weeks we used this preliminary deployment to verify the correctness of the installation scripts, data capture process and content, magnitude and frequency of data transfer, and the transparency of the de-installation process. After this initial refinement period, we proceeded to expand the sample of users. The target population corresponded to the approximately 60 students in our Department's largest research lab. After promoting the study for a period of two weeks, 30 subjects volunteered to participate in the study (members from our group were not allowed to participate). The study's goal, setting, and duration (45 days) was explained to each one of the subjects, and the same fully-instrumented version of the *Pine*'s package was made available for them to install. At the termination date, 1193 user sessions had been collected, an average of 1 session per user per day (no sessions were received during 6 days due to data collection problems).

The last phase consisted of employing the collected data to support different studies and simulating different scenarios that could help us answer the research questions. The particular simulation details such as the manipulated variables, the nuisance variables and the assumptions, vary depending on the research question, so we address them individually within each study.

3.5 Threats to Validity

This study, like any other, has some limitations that could have influenced the results. Some of these limitations are unavoidable consequences of the decision to combine an observational study with simulation. However, given the cost of collecting field data and the fundamental exploratory questions we are pursuing, these two approaches offered us a good balance between data representativeness and power to manipulate some of the independent variables.

By collecting data from 30 deployed instances of *Pine* during a period of 45 days, we believe to have performed the most comprehensive study of this type. Our program of study is representative of many programs in the market, limiting threats to external validity. Although our subjects are students in our Department, we did not exercise any control during the period of study, which gives us confidence that they behaved as any other user would under similar circumstances. Still, more studies with other programs and subjects are necessary to confirm the results we have obtained. For example, we must include subjects that exercise the configurable features of *Pine* and we need to distribute versions of the program for various configurations.

Our simplifying assumptions about the deployment management is another threat to external validity. Although some of the assumptions are reasonable or could be restated in the simulation, empirical studies specifically including various deployment strategies are required [26]. Furthermore, our studies assume that the incorporation of instrumentation probes and anomaly based triggers do not result in additional program faults and that repeated deployments are technically and economically feasible.

We are also aware of the potential impact of observational studies on a subject's behavior. Although we clearly stated our goals and procedures, subjects could have been afraid to send certain type of messages through our version of *Pine*. This was a risk we were willing to take to increase the chances of exploring different scenarios through simulation. Overall, gaining users trust and willingness to be profiled is a key issue for the proposed approaches to succeed and should be the focus of future studies.

The in-house validation process helped to set a baseline for the assessment of the potential of field data (RQ1) and the anomaly based triggers for data transfers (RQ3). As such, the quality of the in-house validation process is a threat to internal validity which could have affected our assessments. We partially studied this factor by carving weaker test suites from an existing suite (Section 4.1) and by considering different number of users to characterize the initial operational profiles (Section 4.3). Still, the scope of our findings are affected by the quality of the initial suite.

To address RQ2, we employ the number of probes to estimate the potential performance overhead caused by different instrumentation strategies. This metric is limited in that it does not consider whether the probes were executed in the field, which equates to assigning the same execution likelihood to all probes. Still, counting the number of probes required by an instrumentation technique is advantageous because it can be computed statically, providing an evaluation that is independent of how the application is exercised. Future studies could further mitigate this threat to construct validity by considering complementary performance measures (e.g., execution time, ratio of probes over instructions executed).

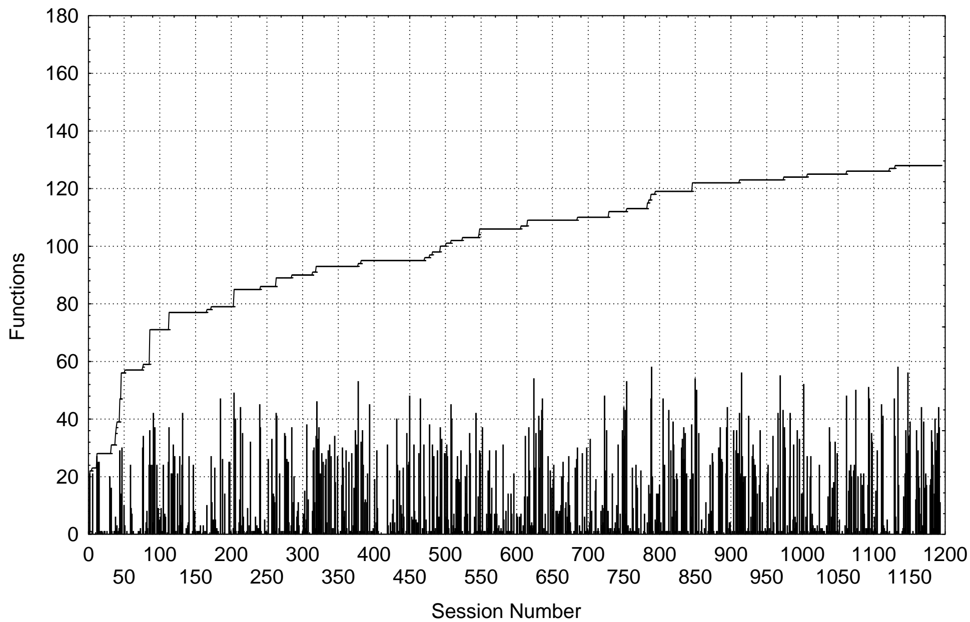

Figure 1: Coverage gain per session and cumulative

4. RESULTS AND ANALYSIS

4.1 Study 1: Field Data Driven Improvement

This first study addresses RQ1, aiming to provide a better understanding of the potential of field data to improve a test suite. This study assesses three different mechanisms to harness the potential of field data for test suite improvement as measured by coverage and fault detection gains.

Simulation Setting. For this study, we assume a *full* profiling strategy was employed to collect coverage data on each deployed instance. We also assume that the potential benefits are assessed at the end of the collection process without loss of generality (the same process could be conducted more often).

We consider three test case generation procedures. First, we defined a procedure that can generate test cases to reach all the entities executed by the users, including the ones missed by the in-house test suite. This hypothetical procedure sets an upper bound on the performance of test suites generated based on field data.

Second, we consider a simple but fully automated procedure that translates each user session into a test case. Each test case consists of two items: 1) an initial setup where the configuration file and mailbox content is matched as closely as possible to the one in the session, and 2) a sequence of commands replaying the ones employed by the user and dummy values to complete necessary fields (e.g., email's destination, subject, content). As per its definition, this simple translation mechanism resulted in 1193 test cases.

Third, we enhanced the previous procedure by allowing a tester to modify the test cases generated by analyzing trace data from field sessions that provided coverage gains that were not captured by the simple translation procedure. This enhancement resulted in 572 additional test cases.

Note that the latest two procedures are not expected to generate test cases that exactly reproduce the behavior observed in the field; these procedures simply attempt to leverage field data to enhance an existing test suite with cases that approximate field behavior.

Results. Figure 1 shows how individual sessions contribute coverage. During the 1193 collected user sessions, 128 functions (9.3% of the total) that were not exercised by the original test suite were executed in the field. Intuitively, as more sessions are gathered we would expect it would become harder to cover new functions. This is corroborated by the growth of the cumulative coverage gain line. Initially, field behavior exposes many originally uncovered functions. This benefit, however, diminishes over time suggesting perhaps that field data can benefit testing the earliest after deployment.

Table 1 presents the gains for the three test generation procedures. The first row presents the potential gain of field data if it is fully leveraged. The second row presents the simple translation approach to exploit field data, which can generate a test suite that provides 3.2% coverage gains. The more elaborated test case generation approach that requires the tester's participation provides 5.9% gains in terms of additional code covered. Although we did not capture block coverage information at the deployed instances (*na* cell in Table 1), we used the test cases generated to measure the block coverage gain which provided additional evidence (12.5% of 10664 blocks covered) about the value of data collected at deployed instances.

Table 1: Functional and block coverage gains

Mechanism / Granularity	Function	Block
Potential gain	128 (9.3%)	*na*
Simple translation	44 (3.2%)	1068 (10.0%)
+Manual enhancement	81 (5.9%)	1328 (12.5%)

Still, as the in-house suite becomes more powerful and just the extremely uncommon execution patterns remained uncovered, we expect that the realization of the potential gains to the factual gains will become harder because a more accurate reproduction of the user session may be required. For example, a user session might cover a new entity when special characters are used in an email address. Reproducing that class of scenario would require capturing the content of the email, which we refrained from doing for privacy and performance concerns. This type of situation may limit the potential gains that can be actually exploited.

The test suites developed utilizing field data also generated gains over the in-house test suite in terms of fault detection capabilities. Table 2 reports the fault seeded in each version, the faults detected through the in-house test suite, and the additional faults found through the test cases generated based on field data. The test suite generated through simple translation of field data discovered a new fault in v4.20 and the mechanism requiring the tester's participation discovered 5 additional faults (12% of seeded faults), raising the overall testing effectiveness on future versions, as measured by fault detection, from 65% to 77%.

Table 2: Fault detection effectiveness

	Fault Seeded	Detected In-house	Additional faults detected	
			Simple Translation	Manually Enhanced
v4.04	10	5 (50%)	0 (0%)	2 (20%)
v4.05	3	3 (100%)	0 (0%)	0 (0%)
v4.10	15	11 (73%)	0 (0%)	1 (7%)
v4.20	15	9 (60%)	1 (7%)	2 (13%)
	43	28 (65%)	1 (2%)	5 (12%)

Impact of initial test suite coverage. Intuitively, initial test suite coverage would affect the potential effectiveness of the field data to facilitate any improvement. For example, a weaker initial test suite is likely to benefit sooner from field data. To confirm that intuition, we carved several test suites from the initial suite with varying levels of coverage. Given the initial test suite T_0 of size n_0, we randomly choose n_1 test cases from T_0 to generate T_1, where $n_1 = n_0 * potencyRatio$ and $potencyRatio$ values range from 0 to 1. We arbitrarily picked a fixed $potencyRatio$ of 0.96 (equivalent to decreasing T_i by 10 test cases to create T_{i+1}), repeating this process using test suite T_i to generate T_{i+1}, where $T_i \supset T_{i+1}$. This process resulted in 29 test suites with decreasing coverage levels.

We then consider the impact of test suite coverage on potential gains. Figure 2 shows how T_0 initial coverage is 60%, potential gain is 9.3%, and overall coverage of 69%. On the other hand, T_{28} provides an initial coverage of 28%, potential gains of 35%, resulting in an overall coverage of 63%. Suites with lower initial coverage are likely to benefit more from field data. Note, however, that the overall coverage (initial plus gain) achieved by a weaker test suite is consistently inferior to that of a more powerful test suite. This indicates that, even if the full potential of field data is exploited to generate test cases, more sessions with specific (and perhaps unlikely) coverage patterns may be required to match the results from a stronger in-house suite.

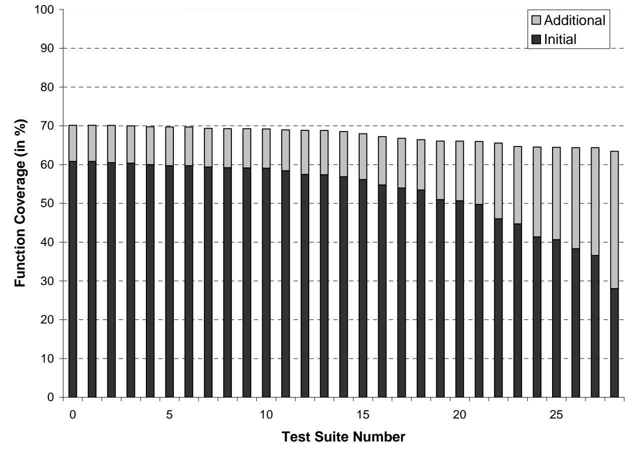

Figure 2: Initial and gained coverage per test suite

4.2 Study 2: Targeted and Sampled Profile

This study addresses RQ2, providing insights on the reduction of profiling overhead achieved by instrumenting a subset of the entities or sampling across the space of events and deployed instances.

Simulation Setting. We investigate the performance of two profiling techniques: targeted profiling (*tp*) and sampling profiling (*sp*). The results from the *full* profiling technique serve as the baseline.

For *tp*, we assume the software was deployed with enough instrumentation to profile all functions that were not executed during the testing process (considering our most powerful test suite T_0). Since we stored the coverage information for each session, we can simulate the conditions in which only a subset of those elements are instrumented. We also assume the same instrumented version is deployed at all sites during the duration of the study.

For *sp*, we have to consider two parameters: the number of strata defined for the event population and the number of versions to deploy. For our study, we defined as many strata as program components (each of the 27 files constitutes a component). Regarding the versions for P, we generate multiple versions by incorporating different sets of instrumentation probes. If we assume each user gets a different instrumented version of the program then the maximum number is 30. The minimum number of versions is two. We also simulated having 5, 10, 15, 20, and 25 deployed versions to observe trends. We denote these *sp* variations as *sp*2, *sp*5, *sp*10, *sp*15, *sp*20, *sp*25, and *sp*30. To get a better understanding of the variation due to sampling, we performed this process on each *sp* variation five times. For example, for *sp*2 we generated five pairs of two versions, where each version in a pair had half of the functions instrumented, and the selection of functions was performed by randomly sampling functions across the identified components.

The simulation results for these techniques were evaluated through two measures. The number of instrumentation probes required by each technique constitutes our efficiency measure. The potential coverage gains generated by each technique constitute our effectiveness measure.

Results. The scatterplot in Figure 3 depicts the number of instrumentation probes and the potential coverage gain for *full*, *tp*, and a family of *sp* techniques. There is one observation for the *full* and *tp* techniques, and the five observations for the *sp* (corresponding to the five samples that were simulated) have been averaged.

The *full* technique needs almost 1400 probes to provide 9.3% gain. By utilizing the *tp* technique, and given the in-house test suite we had developed, we were able to reduce the number of probes to 32% or 475 probes from the *full* technique without loss in potential gain, confirming previous conjectures about its potential. The *sp* techniques provided efficiency gains with a wide range of effectiveness. As more instrumented versions are deployed, the profiling efficiency as measured by the number of instrumentation probes increases. With the most aggressive implementation of the technique, *sp*30, the number of probes per deployed instance is reduced to 3% of the *full* technique. By the same token, a more extensive distribution of the instrumentation probes across a population of deployed sites diminished the potential coverage gains to approximately a third (when each user gets a uniquely instrumented version) compared with the *full* technique. It also appears that as the number of strata gets higher, there is a larger coverage gain variation. This suggest that when number of probes assigned to a deployed site is small, it becomes more important how the subgroups are chosen.

We then proceeded to combine *tp* and *sp* to create a hybrid technique, *hyb*, that performs sampling only on functions that were not covered by the initial test suite. Applying *tp* on *sp*2, *sp*5, *sp*10, *sp*15, *sp*20, *sp*25, and *sp*30 yielded *hyb*2, *hyb*5, *hyb*10, *hyb*15, *hyb*20, *hyb*25, and *hyb*30, respectively. Hybrid techniques (depicted in Figure 3 as triangles) were as effective as *sp* techniques in terms of coverage gains but even more efficient. For example, *hyb*2 cut the number of probes by approximately 70% compared to *sp*2. However, when a larger number of instrumented versions are deployed, the differences between the *hyb* and the *sp* techniques become less obvious as the chances of obtaining gains in an individual deployed site diminishes.

Sampling and data loss. To get a better understanding of how much information is lost when we sample across deployed instances, we identified the top 5% (68) most executed functions under *full* and compared it against the same list generated when sampling schemes are used. We executed each *sp* technique five times to avoid potential sampling bias. The results are summarized in Table 3.

Utilizing *sp*2 reduces the number of probes in each deployed instance by half, but it can still identify 65 of the top 68 executed functions. When the sampling became more aggressive, data loss increased. The magnitude of the losses based on the sampling aggressiveness is clearly exemplified when *sp*30 is used and only 61% of the top 5% executed functions are correctly identified. Interestingly, the number of probe reductions occurred at a higher rate than the data loss. For *sp*30, the number of probes was reduced to 3% when compared to *full* profiling. It was also noticeable that more aggressive sampling did not necessarily translate into higher deviations due to sampling. In other words, the results are quite independent from the functions that end up being sampled on each deployed version.

Overall, even the most extreme sampling across deployed instances did better than the in-house test suite in identifying the top executing functions. The in-house test suite, however, was not designed with this goal in mind. Still, it is interesting to realize that *sp*30, with just 45 probes per deployed version in a program with 1373 functions, can identify 61% of the most executed functions.

Table 3: *Sp* data loss versus *Full*

Technique	Probes	Top 5% Functions		
		Common Functions	%	Std.Dev
sp2	686	65	96	0.9
sp5	274	55	81	2.2
sp10	137	49	72	1.5
sp15	91	45	66	2.2
sp20	68	44	65	1.7
sp25	54	43	64	2.6
sp30	45	41	61	2.5
House	1373	35	51	-

4.3 Study 3: Anomaly Based Triggers for Transfers

This study addresses RQ3, assessing the effect of detecting anomalies in operational profiles to trigger data transfers. We empirically evaluate various techniques and assess their impact in the coverage and fault detection capabilities of test suites generated based on field data.

Simulation Setting. An operational profile consists of a set of operations and their associated probabilities of occurrences [17]. Operational profiles can serve to guide the test suite building process or the allocation of testing resources. In this study, we aim at detecting operational profile departures from an in-house operation profile. The objective is then to trigger a field data transfer when the released software is operating in ways we did not anticipate.

To develop the in-house operational profile baseline for *Pine*, we followed the methodology describe by Musa [17]: (1) identify operations initiators; in our case these were the regular users and system controller; (2) create a list of operations by traversing the menus just as regular users would initiate actions; and (3) perform several pruning iterations, trimming operations that are not fit (e.g., operations that are not able to exist without others). The final list contained 34 operations.

To assign probabilities to those operations and create the operational profile baseline, we randomly selected a percentage of users as beta-testers and employed their sessions to characterize the execution probability of each operation. We explore three levels of characterization for the baseline utilizing 3(10%), 6(20%), and 15(50%) users. We also assume the operational profile baseline and acceptable deviation is embedded within each deployed instance as a simple vector of values. (Note that under Musa's approach only one operational profile is used to characterize the system behavior independent of the variation among users.)

For this study, seven anomaly detection techniques were implemented following the strategy defined in Section 2.4. Although there are many advanced techniques for anomaly detection, the chosen techniques for this stage were selected

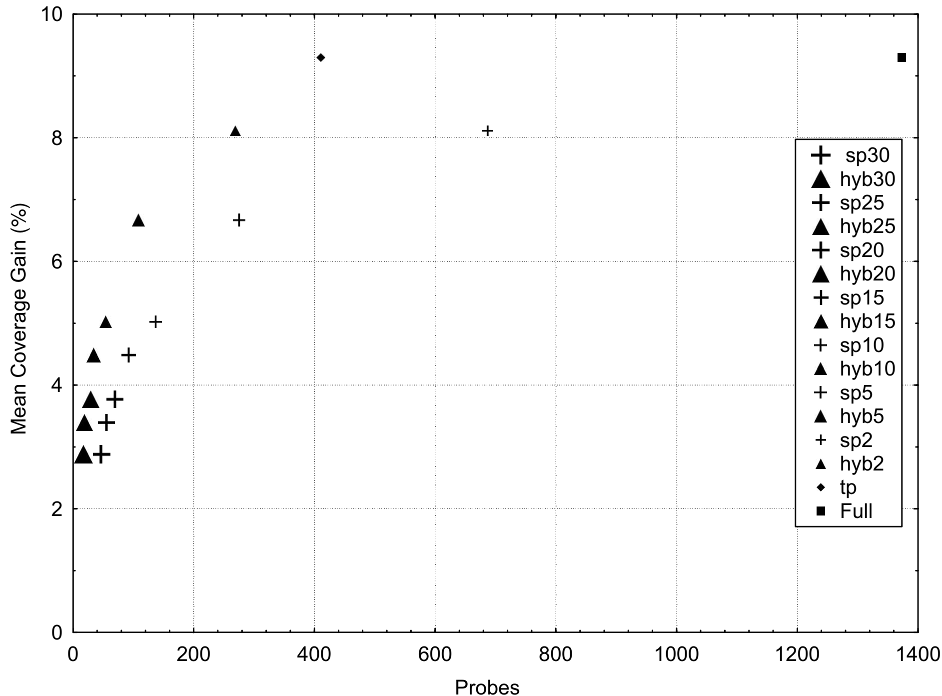

Figure 3: Coverage Gains and Probes for *full*, *tp*, *sp*, and *hyb*

because of their simplicity for implementation and automation (we mention other options in Section 5). The first technique corresponds to the utilization of the *full* strategy which performs no reduction in the number of transfers.

The second technique, *meandev*, utilizes the mean and the standard deviation of each operation to define the boundaries of normal behavior. Given $HouseVal(Op_i)$ which serves as the baseline occurrence probability for the ith operation, $HouseDev(Op_i)$ which constitutes its standard deviation, and $FieldVal(Op_i)$ which represents the field occurrence rate for that operation, the algorithm indicates an anomaly has occurred when $FieldVal(Op_i)$ falls outside $[HouseVal(Op_i) \pm HouseDev(Op_i)]$. If no anomaly is detected, then an acknowledgement is sent to the server.

Setting the tolerance to one standard deviation is a conservative approach to reduce the number of transfers (approximately a third of all sessions would be anomalies under a normal distribution). The third and fourth technique, *double_meandev* and *minmax* respectively, relax these constraints. *Double_meandev* also utilizes the mean, but doubles the range of standard deviation for its tolerance value, $HouseDev(Op_i)$. *Minmax*, defines a pair of $HouseDev(Op_i)$ (maximum and minimum) for each operation. Every time an operation has a $FieldVal(Op_i)$ above the established maximum or below the minimum, it triggers an anomaly.

When an anomaly is detected, the operational profile and all the captured field behavior is sent to the server for the software engineers to determine how this anomaly should be treated (e.g., discard, reassess baseline, reproduce scenario). This study explores two alternatives: 1) software engineers analyze the data but do not change the existing operational profile, 2) software engineers analyze the data and refine the existing operational profile and the tolerance values with the collected anomalous profile, using it as a feedback mechanism. The presence of feedback can assist *meandev* and *double_meandev* to increase the confidence in the current baseline and further constrain the levels of acceptable deviation (reduce $HouseDev(Op_i)$) as more normal observations are collected. *Minmax* with feedback can help to increase the size of the acceptable bounds, but it cannot constrain it by definition, resulting always in larger $HouseDev(Op_i)$.

Given the variable session duration, we decided to control this source of variation so that the number of operations in each session did not impact $FieldVal(Op_i)$ (e.g., sessions containing just one operation would assign 100% execution probability to that operation and most likely generate an outlier). The simulation considers partitions of 10 operations (mean number of operations across all sessions) for each user so that a session with less than 10 operations is clustered with the following session from the same deployed site until 10 operations are gathered, while partitioning longer sessions to obtain partitions of the same size.

Results. Table 4 summarizes the findings of this study. The first column identifies the triggering technique we employed. Column two contains the number of beta testers (and the percentage over all users) considered to define the baseline operational profile and what is considered normal. Column three reports the number of transfers required under each combination of technique and number of beta testers. Columns four and five correspond to the number of functions covered and the new faults found when we employed the test suite generated with field data. (For this study we considered jointly the test cases generated through the two procedures described in Section 4.1.)

Table 4: The effects of anomaly based triggers

Technique	Beta Testers #	Beta Testers %	Transfers #	Transfers %	Funct. Cov.	Faults Found
Full	3	10	452	100	921	5
Meandev			452	100	921	5
Meandev+f			363	80	839	2
Dbl. Meandev			278	61	851	1
Dbl. Meandev+f			128	28	847	1
Minmax			126	28	850	1
Minmax+f			30	7	834	0
Full	6	20	393	100	921	5
Meandev			393	100	921	5
Meandev+f			322	82	837	3
Dbl. Meandev			168	43	841	3
Dbl. Meandev+f			116	29	830	3
Minmax			49	12	855	1
Minmax+f			17	4	843	1
Full	15	50	307	100	921	5
Meandev			307	100	921	5
Meandev+f			236	77	858	4
Dbl. Meandev			86	28	853	4
Dbl. Meandev+f			76	24	853	3
Minmax			7	2	850	2
Minmax+f			4	1	850	2

These results provide evidence that a better characterization of the baseline behavior reduces the number of transfers (the only exceptions were *meandev+f* and *double_meandev+f* from 3 to 6 beta testers which could have been caused by the random choice of users). This reduction is noticeable in the smaller percentages of transfers per technique when compared with *full*, which became smaller as the number of beta-testers increased. Overall, techniques utilizing feedback to adjust what constitutes an anomaly are more effective at reducing the number of transfers. Even the conservative *meandev+f* reduced the number of transfers up to 77% when compared to *full* with 15 beta testers. *Minmax+f* provided the greatest reduction in the number of transfers, requiring just 7% of the number required by the *full* technique with just 3 beta testers. Such a dramatic reduction not only helps to improve the transparency at the deployed site, but it also acts like a filter to reduce the amount of data to be later analyzed in-house.

Given the same beta testers, we conjectured that a technique saving data transfers would sacrifice test cases, potentially forfeiting coverage and fault detection. That was true as we added feedback to *meandev*, *double_meandev*, and *minmax*. However, *minmax* and *minmax+f* had fewer transfers than *meandev+f*, *double_meandev*, and *double_meandev+f* but often covered more functions (*minmax* defines a maximum probability of execution per operation which implies that when a function is covered for the first time, it will constitute an anomaly and generate a transfer). Nevertheless, the data gathered through *meandev+f*, *double_meandev*, and *double_meandev+f* enabled us to detect more faults than the *minmax* techniques. Clearly, in addition to the technique implementation complexity, practitioners must consider these tradeoffs when choosing an anomaly anomaly detection technique.

5. CONCLUSIONS AND FUTURE WORK

We have presented a family of empirical studies to investigate several implementations of the profiling strategies and their impact on testing activities. *Our findings confirm the large potential of field data obtained through these profiling strategies to drive test suite improvements.* However, our studies also show that the gains depend on many factors including the quality of the in-house validation activities, the process to transform potential gains into factual gains, and the particular application of the profiling techniques. Furthermore, the potential gains tended to stabilize over time pointing to the need for adaptable techniques that re-adjust the type and number of profiled events.

Our results also indicate that *targeted and sampling profiling techniques can reduce the number of instrumentation probes by up to one order of magnitude, which greatly increases the viability of profiling released software.* When coverage is used as the selection criteria, *tp* provides a safe approach (it does not lose data) to reduce the overhead. However, practitioners cannot expect to notice an important overhead reduction with *tp* in the presence of a poor in-house test suite. On the other hand, the sampling profiling techniques were efficient independent of the in-house test suite attributes but at the cost of some effectiveness. Although practitioners can expect to overcome some of the *sp* effectiveness loss by profiling additional sites, the representativeness achieved by the sampling process is crucial to guide the insertion of probes across events and sites. It was also noticeable that the simple sequential aggregation of *tp* and *sp* techniques to create *hyb* provided further overhead reduction.

We also found that anomaly based triggers can be effective at reducing the number of data transfers, a potential trouble spot as the number of profiled sites increases. It is reasonable for practitioners utilizing *some of the simple algorithms we implemented to reduce the number of transfers by an order of magnitude and still retain 92% of the coverage achieved by full profiling.* Still, further efforts are necessary to characterize the appropriateness of the techniques under different scenarios and to study the adaptation of more accurate, even if expensive, anomaly detection techniques.

Profiling released software techniques still have to tackle many difficult problems. First, one of the most significant challenges we found through our studies was to define how to evaluate the applicability, cost, and benefit of profiling techniques for released software. Our empirical methodology provides a viable model combining observation with simulation to investigate these techniques. Nevertheless, we still need to explore how to evaluate these strategies at a larger scale and include additional assessment measures such as deployment costs and bandwidth requirements. Second, we have yet to exploit the full potential of field data. For example, our studies quantified simple mechanisms to transform field data into test cases. Given the appropriate data, these mechanisms could be refined to better approximate the behavior at deployed sites. Furthermore, in the presence of thousands of deployed instances, this process must happen continually, which introduces new challenges to filter, interpret, and aggregate a potentially overwhelming amount of data. Last, profiling deployed software raises issues not only about acceptable overhead but also about user's confidentiality and privacy. Future techniques and studies must take these factors into consideration.

Acknowledgments

This work was supported in part by a NSF-ITR program under award 0080898 and a CAREER Award 0347518 to University of Nebraska, Lincoln. We especially thank the users who volunteered for the study and the participants of the First Workshop on Remote Analysis and Measurement of Software Systems for their feedback on earlier stages of this work. We also thank Gregg Rothermel and Michael Ernst for providing feedback on earlier versions of this paper.

6. REFERENCES

[1] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[2] T. Ball and J. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, 1994.

[3] T. Ball and J. Laurus. Optimally profiling and tracing programs. In *Annual Symposium on Principles of Programming*, pages 59–70, Aug. 1992.

[4] B. Calder, P. Feller, and A. Eustace. Value profiling. In *International Symposium on Microarchitecture*, pages 259–269, Dec. 1997.

[5] S. Elbaum, S. Kanduri, and A.Andrews. Anomalies as precursors of field failures. In *International Symposium of Software Reliability Engineering*, pages 108–118, 2003.

[6] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, Feb. 2002.

[7] A. Glenn, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices*, 32(5):85–96, 1997.

[8] S. Graham and M. McKusick. Gprof: a call graph execution profiler. *ACM SIGPLAN*, 17(6):120–126, June 1982.

[9] K. Gross, S. McMaster, A. Porter, A. Urmanov, and L. Votta. Proactive system maintenance using software telemetry. In *Workshop on Remote Analysis and Monitoring Software Systems*, pages 24–26, 2003.

[10] M. Harrold, R. Lipton, and A. Orso. Gamma: Continuous evolution of software after deployment. cc.gatech.edu/aristotle/Research/Projects/gamma.html.

[11] D. Hilbert and D. Redmiles. An approach to large-scale collection of application usage data over the Internet. In *International Conference on Software Engineering*, pages 136–145, 1998.

[12] D. Hilbert and D. Redmiles. Separating the wheat from the chaff in internet-mediated user feedback, 1998.

[13] InCert. Rapid failure recovery to eliminate application downtime. www.incert.com, June 2001.

[14] J.Bowring, A. Orso, and M. Harrold. Monitoring deployed software using software tomography. In *Workshop on Program analysis for software tools and engineering*, pages 2–9, 2002.

[15] D. Libes. *Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs*. O'Reilly & Associates, Inc., Sebastopol, CA, Nov. 1996.

[16] B. Liblit, A. Aiken, Z. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *Conference on Programming Language Design and Implementation*, pages 141–154. ACM, June 2003.

[17] J. Musa. *Software Reliability Engineering*. McGraw-Hill, New York, NY, 1999.

[18] Netscape. Netscape quality feedback system. home.netscape.com/communicator/ navigator/v4.5/qfs1.html.

[19] Nielsen. Nielsen net ratings: Nearly 40 million Internet users connect via broadband. www.nielsen-netratings.com, 2003.

[20] U. of Washington. Pine information center. http://www.washington.edu/pine/.

[21] A. Orso, T. Apiwattanapong, and M.J.Harrold. Leveraging field data for impact analysis and regression testing. In *PFoundations of Software Engineering*, pages 128–137. ACM, September 2003.

[22] A. Orso, D. Liang, M. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *International Symposium on Software Testing and Analysis*, pages 65–69, 2002.

[23] C. Pavlopoulou and M. Young. Residual Test Coverage Monitoring. In *International Conference of Software Engineering*, pages 277–284, May 1999.

[24] S. Reiss and M. Renieris. Encoding Program Executions. In *International Conference of Software Engineering*, pages 221–230, May 2001.

[25] D. Richardson, L. Clarke, L. Osterweil, and M. Young. Perpetual testing project. http://www.ics.uci.edu/ djr/edcs/PerpTest.html.

[26] A. van der Hoek, R. Hall, D. Heimbigner, and A. Wolf. Software release management. In M. Jazayeri and H. Schauer, editors, *European Software Engineering Conference*, pages 159–175. Springer–Verlag, 1997.

[27] C. Yelmaz, A. Porter, and A. Schmidt. Distributed continuous quality assurance: The Skoll project. In *Workshop on Remote Analysis and Monitoring Software Systems*, pages 16–19, 2003.