

Monitoring Deployed Software Using Software Tomography

Jim Bowring
College of Computing
Georgia Institute of
Technology
Atlanta, Georgia 30332-0280
bowring@cc.gatech.edu

Alessandro Orso
College of Computing
Georgia Institute of
Technology
Atlanta, Georgia 30332-0280
orso@cc.gatech.edu

Mary Jean Harrold
College of Computing
Georgia Institute of
Technology
Atlanta, Georgia 30332-0280
harrold@cc.gatech.edu

ABSTRACT

Software products are often released with missing functionality or errors that result in failures in the field. In previous work, we presented the GAMMA technology, which facilitates remote monitoring of deployed software and allows for a prompt reaction to failures. In this paper, we investigate one of the principal technologies on which GAMMA is based: software tomography. *Software tomography* splits monitoring tasks across many instances of the software, so that partial information can be (1) collected from users by means of light-weight instrumentation and (2) merged to gather the overall monitoring information. After describing the technology, we illustrate an instance of software tomography for a specific monitoring task. We also present two case studies that we performed to evaluate the presented technique on a real program. The results of the studies show that software tomography can be successfully applied to collect accurate monitoring information using only minimal instrumentation on each deployed program instance.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—Monitors, Testing tools, Tracing

General Terms

Verification, Reliability, Experimentation, Measurement

Keywords

Software engineering, code coverage, Gamma technology, software tomography, tomographic refinement

1. INTRODUCTION

The development of reliable and safe software is difficult. Quality assurance tasks such as testing are often constrained due to time-to-market pressures and because products must function in many environments with variable configurations.

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.

PASTE'02, November 18–19, 2002, Charleston, SC, USA.
Copyright 2002 ACM 1-58113-479-7/02/0011 ...\$5.00.

Consequently, released software products may exhibit missing functionality, errors, incompatibilities with the running environment, security holes, or inferior usability.

Therefore, there is a need for techniques that can monitor software's behavior after its release, enabling software producers to effectively detect and fix problems. During development, systems can be diagnosed by performing static and dynamic analysis, using techniques such as ad-hoc runtime environments or code instrumentation. Unfortunately, these approaches cannot be used when the software is executing in the field. A suitable ad-hoc runtime environment may not be available on the target platform and, even if available, might be too inefficient to be used. Extensive instrumentation, on the other hand, produces space and execution time overhead that is generally unacceptable [3].

Other researchers have addressed this problem and investigated possible solutions. The Perpetual Testing project recognizes the need to develop “seamless, perpetual analysis and testing of software through development, deployment and evolution,” [15] and proposes one partial solution, called Residual Testing [14]. Residual Testing proposes to continuously monitor for fulfillment of test obligations that were not satisfied in the development environment—the *residue*. Residual Testing provides only a partial solution to the problem of monitoring deployed software because it does not address the issue of reducing the instrumentation required to monitor the residue.

Another proposed approach is Expectation-Driven Event Monitoring (EDEM). EDEM uses software agents deployed over the Internet to collect application-usage data to help developers improve the fit between application design and use [4, 9]. This approach addresses the problem of monitoring deployed software, but is limited in that it collects only usage information through event monitoring and cannot be used to gather different kinds of information (e.g., profiling information). Furthermore, EDEM collects the same information about each executing program, which could result in unacceptable overhead.

In previous work, we presented GAMMA technology as a way of continuously monitoring and improving software products after deployment [12]. GAMMA provides several important technologies, including software tomography and on-site code modification. *Software tomography*¹ (1) divides

¹Software Tomography is named after computed axial tomography (CAT) wherein the structure of a subject is reconstructed from a series of cross-sectional images.

the task of monitoring software and gathering dynamic analysis information into a set of subtasks that require only minimal instrumentation, (2) assigns the subtasks to different instances of the software, so that none of the instances will experience significant performance degradation due to instrumentation, and (3) integrates the information returned by the different software instances to gather the overall monitoring information. *On-site code modification* enables modifications of the code in the field and allows for dynamically reconfiguring instrumentation (e.g., to further investigate a problem or to gather different kinds of information) and for efficiently delivering solutions or new features.

In this paper, we investigate software tomography. After describing the technology, we show how it can be used to collect monitoring information from running instances of a program. In particular, we select a representative monitoring task, branch coverage, and describe a suitable set of software-tomography techniques.

To evaluate the presented techniques, we perform two case studies in which we apply the techniques to a real program and assess, through simulation, their effectiveness and efficiency in performing the selected monitoring task. Although preliminary in nature, the results of the study are promising. We find that by using software tomography we are able to collect accurate monitoring information for measuring branch coverage by instrumenting less than two percent of the branches in each instance. Monitoring conducted without software tomography would normally require full instrumentation of some, if not all, the running instances of the program.

The main contributions of the work are:

- An investigation into software-tomography techniques for the monitoring of deployed software. This includes the establishment of an initial framework for (1) the partitioning of monitoring tasks so as to provide for the distribution of light-weight instrumentation over many program instances; and (2) the process of tomographic refinement, a dynamic improvement of the tomographic process.
- The first set of studies to evaluate, by simulation, software-tomography techniques on a real example. The studies explore the application of the aforementioned framework to the task of measuring branch coverage in a subject.

2. SOFTWARE TOMOGRAPHY

Traditional monitoring tasks involve instrumenting the code by inserting a full complement of probes into each instance of the software. For most such tasks, this approach requires probes at many points in the software, significantly increasing its size and compromising its performance. Such overhead is in general unacceptable to users, and inapplicable in contexts in which resources are limited, such as embedded systems.

To achieve low-impact and minimally-intrusive monitoring, we developed a new technique, called *software tomography* [12]. Our technique divides the monitoring task into a set of subtasks, each of which involves little instrumentation, and assigns these subtasks to individual software instances for monitoring. The technique then synthesizes the information collected from each instrumented instance, and yields the monitoring information required for the original task.

In order to use software tomography and gather information for a specific monitoring task we must perform a series of steps: (1) identification of basic subtasks, (2) assignment of subtasks to instances, and (3) optimization of the number and placement of probes. These steps represent the key components of our process and next we discuss each in detail.

Identification of basic subtasks. Some simple tasks, such as statement coverage, can be easily partitioned into minimal subtasks. More complex monitoring tasks, such as dynamic data-flow analysis and profiling, will generally require that each subtask probe more than one point in the program. Yet other tasks, such as the detection of memory-access information, which involve the recording of execution details at several program points, might require complicated task-partitioning techniques, utilizing static program analysis. Tasks might also include the validation of usage and operational profiles developed during the software development process. Here, the subtasks will be determined on the basis of program operations.

Assignment of subtasks to instances. Software instances are executed by users at will and each user may use different parts of the software. Therefore, the frequency of execution and the parts of the software exercised may vary across instances and over time. To ensure that we gather enough information for each subtask, so that the overall monitoring information is meaningful, we need a technique to assign subtasks to instances in an effective and efficient way. Our technique performs subtask reassignment based on feedback from the field. For example, we may eliminate subtasks that have been completed and reassign subtasks not adequately accomplished. Other possible inputs to the process might include random selection or recommendations from a statistical model. We call our reassignment technique *tomographic refinement*.

Optimization of number and placement of probes. Optimizing the number and placement of probes can further reduce the instrumentation overhead for the software instances. For many tasks, such as data-flow coverage, static information extracted from the software can be used to reduce the number of probes needed for a specific subtask. Furthermore, static information can be used to optimize the number and placement of the probes. In particular, when more than one subtask must be assigned to one software instance, the way in which subtasks are grouped may affect the number of probes needed. For example, for path profiling, we use an existing technique by Ball and Larus [3], which instruments a program for profiling by choosing a placement of probes that minimizes run-time overhead. We have adapted this technique so that it determines the placement of probes that monitor only the desired paths instead of all acyclic paths in the procedures [1].

Software tomography can be used to perform different kinds of dynamic analysis and monitoring tasks on deployed systems. For example, software tomography might be used to monitor the behavior of software and detect possibly abnormal or unforeseen situations; in this case, software failures may be automatically identified using mechanisms that compare a program's behavior signature (e.g., [7, 16]) computed in the field with a signature computed in-house.

In this paper, we discuss how to use software tomography for a specific dynamic-analysis task: monitoring for branch coverage. We present two techniques for measuring branch

coverage using a set of deployed program instances. We demonstrate the partial instrumentation of single instances, the synthesis of the collected data, and the technique of tomographic refinement.

As stated above, the first step in the process for defining a software-tomography technique is the identification of basic subtasks. In the case of monitoring for branch coverage, the most basic subtask is that of monitoring a single branch.

After the basic subtasks have been identified, single subtasks can be assigned to software instances. We assume, for this discussion, that we have no a priori information about how the software will be used. Therefore, we evenly distribute the subtasks among the instances, one per instance.² Such a distribution of probes, although efficient, is in general ineffective. It is possible that the branches instrumented in a given instance will never be executed. This will be true for unreachable code and very likely for rarely-used code. Thus the initial effectiveness of the detection process varies inversely with the number of instances monitoring these unreachable and rarely-exercised branches.

To remedy this problem, we introduce tomographic refinement, as described above. In this paper, we restrict our attention to two possible refinement policies: round-robin and aggregation.

First we define a measure, which we call the refinement ratio. This ratio is calculated, in this case, by dividing the number of branches reported as covered in a given time interval by the number of branches yet to be covered. We can then define thresholds in terms of this ratio, as we demonstrate below. The measurement of time here is problematic, thus we choose to use program executions as a proxy for the unit of time in our simulations.

In *round-robin refinement*, every time branch b is reported as covered, (1) the instrumentation for that branch is disabled/removed in all the instances that are instrumented for b , and (2) all the instances that were instrumented for b are then re-instrumented for another branch, on a round-robin basis, from the pool of branches not yet reported as covered.

Aggregation refinement is an improvement of round-robin refinement that provides for the assignment of more than one branch per instance. Aggregation refinement is initiated when the refinement ratio falls below a specific threshold. It is worth noting that this approach increases the instrumentation overhead of each instance primarily with respect to code size, as the additional instrumentation is for branches that are likely to be rarely exercised.

As both refinement techniques are interactive and dynamic, we need to be able to control their termination based on the quality of their success. For these techniques, we define a termination policy by providing a lower limit to the refinement ratio. When ratio falls below this threshold value, we terminate the refinement process.

In the next section, we describe our empirical investigations using two case studies that we designed to evaluate the proposed software-tomography techniques.

3. EMPIRICAL INVESTIGATIONS

To investigate whether software tomography can be effectively used to perform continuous, lightweight monitoring of

²It is worth noting that, based on the number of deployed instances, we may not be able to assign all the subtask in this initial phase. If this is the case, tasks that are left unassigned are assigned by the tomographic-refinement policy.

deployed software, we performed a set of case studies considering the following research question:

Given a program P , a set of instances of P running on different machines, and a monitoring task T that, if realized by fully instrumenting program P , would produce information I , can we estimate I by partially instrumenting each instance of P , collecting (partial) monitoring data from each instance, and merging such data?

We restricted our attention to the software tomography task presented in Section 2 and we considered four techniques:

Full instrumentation. Each instance is fully instrumented.

This technique does not exploit software tomography and serves as our control technique. It lets us compare the approximated information that we gather using the tomographic techniques with the complete monitoring information. We refer to this technique as the *FULL* technique.

Single entity. The set of branches are evenly assigned among the instances so that each instance is instrumented for one single branch. We refer to this technique as the *SIMPLE* technique.

Round-robin refinement. This technique implements the round-robin approach described in section 2. We refer to this technique as the *REF-RR* technique.

Aggregation refinement. This technique implements the aggregation approach described in section 2. We refer to this technique as the *REF-AG* technique.

To investigate our research question, we simulated the application of the four techniques to a subject and collected and compared the results. In the next sections, we illustrate our experimental settings and present and discuss two case studies.

3.1 Experimental Settings

As a subject for our studies, we used a C program called **Space**. **Space** is an interpreter for an antenna array definition language written for the European Space Agency. It consists of 136 functions and 6,200 lines of code, and is available with a set of 13,585 test cases.

Because of the nature of the considered studies, performing our experiments in a real setting would involve the actual deployment of an application to a number of sites, the presence of one or more users at each site, and a complex infrastructure to collect and merge the data. Therefore, we performed our experiments using simulation. Using simulation we can explore our research question without the initial overhead of actually deploying software and collecting the data via a network or the Internet.

We implemented our simulation as follows. First, we used the ARISTOTLE analysis system [2, 6] to instrument **Space** for branch coverage, ran all the 13,585 test cases, and stored the coverage information (i.e., which branches are covered by each test case) in a relational database. Then, we built a tool on top of the database: the TOMOGRAPHIC SIMULATOR (TS).

Using TS, we can perform simulations where (1) several instances of **Space** are deployed on different sites, and (2) each instance is partially instrumented and is executed by

one or more users at each site. The tool simulates partial instrumentation by recording, for each instance, the instrumented branch(es) for that instance. The tool simulates an execution of the program at a given site s by randomly selecting a test case t from a pool. The pool can be the entire set of test cases or a subset thereof, depending on the parameters of the simulation (as described below). Based on the coverage information stored in the database for t and on the instrumentation information recorded for s , the tool can report whether an instrumented branch would be covered by the considered execution.

There are different parameters for a simulation, which can be set using a menu-driven interface provided by TS:

Number of deployed instances (N). This parameter represents the number of sites at which the monitored program is deployed and used.

Site weight (W). This parameter is used by the tool to assign to each site a random weight w between 1 and W . The weight of one site represents the frequency with which users at that site execute the software relative to other users at other sites. For example, if the weight is 2 for site $s1$ and 4 for site $s2$, then the instance of the program in $s2$ will be executed twice as often as the instance in $s1$. If W is set to 1, all sites execute the program with the same frequency.

Site input space (I). This parameter is used by the tool to assign to each site a random value i between 1 and I . This value determines how many test cases are associated with the site (i.e., how many different executions may occur at the site). A set of i test cases is assigned to the site by selecting test cases from two pools: a pool of unique test cases, which are removed from the pool once assigned, and a pool of overlapping test cases, which can be assigned to more than one site. If I is set to *all*, then each site is assigned the whole test suite.

Using the latter two parameters we can simulate individual operational profiles [11] at each site. By modifying these two parameters, we can tune the heterogeneity of the user population. In the case of technique *REF-AG*, there are two additional parameters, that is, the *threshold for refinement (TR)* and the *threshold for termination (TT)*. The value of *TR* is used to decide when to assign additional branches to the instances and the value of *TT* is used to decide when to terminate the refinement process, as described in Section 2. For both our studies, we used a value of 10% for *TR* and a value of 0.1% for *TT*.

After the parameters have been set and TS has initialized all the sites with a pool of test cases and possibly a weighting factor, the simulation starts. The tool iterates through all the sites and, for each site, simulates the execution of that instance with the input provided from the set of tests assigned to that site. TS reports whether the instrumented branches were exercised by that execution. This information is retrieved with an appropriate query to the database. When all the sites have been processed once, TS performs the refinement for *REF-RR* and *REF-AG*, by updating the instrumentation information, and then re-iterates through all the sites. The process terminates after a given number of iterations, when *TT* is reached, or when the user stops it.

3.2 Case Studies

In this section, we present the results of two case studies in which we evaluated the considered software-tomography techniques on *Space* using our simulation tool TS. In these studies, we considered two different scenarios. For the first study, we constructed a scenario in which users manifest identical behavior in terms of their potential set of inputs. For the second study, we constructed a scenario in which users have differing operational profiles.

To be able to compare the results, we assumed the initial distribution of probes to be the same for all the three tomographic techniques.

In each study, we evaluated the different tomographic techniques along three dimensions:

Effectiveness. The accuracy of the information computed by the technique with respect to the actual information (provided by the control technique).

Efficiency with respect to instrumentation. The number of probes per instance required for the application of the technique.

Efficiency with respect to interactions. The number of interactions with the deployed instances occurring during the monitoring. Interactions occur each time the instrumentation is updated.

3.2.1 Case Study 1

In this study, we examine the performance of each considered technique when all instances of the software are used in a uniform fashion. To this end, we set the input space so that each site is assigned the complete test suite.

The parameters of the simulation are set as follows:

- Number of deployed instances := 2,500
- Site weight := 1
- Site input space := *all*
- Threshold for *REF-AG* := 10%

The diagram in Figure 1 shows the results for 100 iterations of the simulation for the four techniques. As shown in the graph, technique *FULL* reports the coverage of 1,074 branches during the first iteration. This limit is a function of the test suite. Thus, no additional coverage is achieved during the following iterations.

Effectiveness. As expected, all three tomographic techniques asymptotically tend to the curve for technique *FULL*. The curve for technique *SIMPLE* reaches the curve for *FULL* after several thousands of iterations (at a point not shown in the diagram). The reason why the two curves eventually converge is that, with uniform sites and without any refinement, each branch reported as covered by *FULL* has a constant, greater than zero probability of being exercised. In other words, because of the uniformity of the sites, this result is a property of the probability distribution for branch coverage in the test suite. Nevertheless, if we express the accuracy of a technique in terms of the area between the curve for the technique and the curve computed by *FULL*, we can easily see how *REF-AG* performs better than *REF-RR* and, in turn, *REF-RR* performs better than *SIMPLE*.

Efficiency w.r.t. instrumentation. By definition, *SIMPLE* and *REF-RR* always have only one branch instrumented per instance. *REF-AG*, on the other hand, performs tomographic refinement by assigning an increasing number of probes to software instances. In this study, *REF-AG* reaches

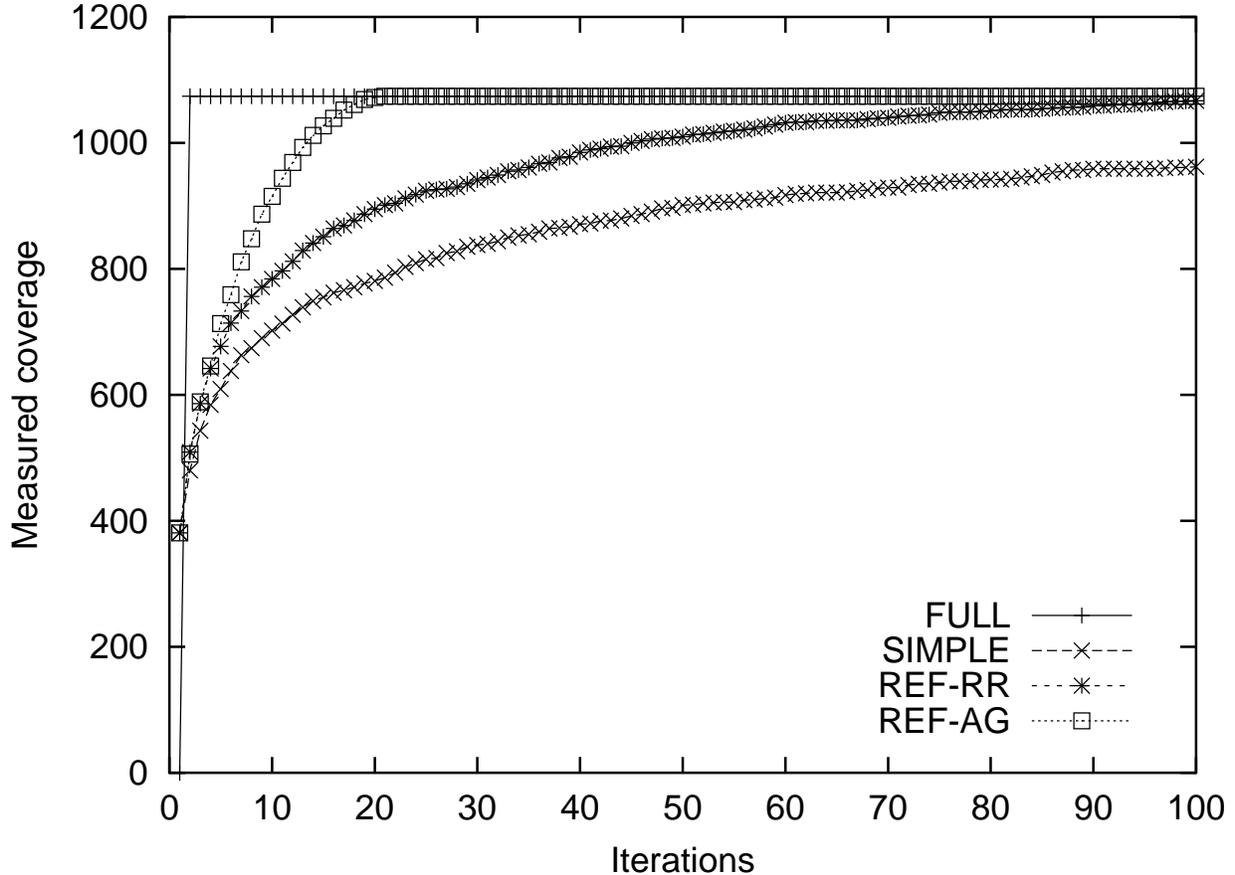


Figure 1: Results for Case Study 1.

an optimum and stops refining at the 22nd iteration, after having performed 17 refinements. Therefore, the number of branches instrumented by the technique in this case is 1.4% of the total number of branches (17 over 1,228).

Efficiency w.r.t. interactions. By definition, *SIMPLE* has no interactions with the software instances, whereas techniques using tomographic refinement do. According to the termination policy described in Section 2 and the simulation parameters, both refinement techniques keep on performing refinements until the rate of progress falls below a threshold of 0.1%. In the case of *REF-RR*, the threshold is not reached within the first 100 iterations. During this time *REF-RR* performs 4,811 interactions with the sites on which the program is deployed. *REF-AG* computes the accurate coverage information (1,074 branches covered) in just 21 iterations, but at the cost of a four-fold increase in the number of interactions over *REF-RR* (20,562). Thus, the improved performance is achieved at a performance cost. This trade-off suggests that there may be ways to tune the two parameters, depending on the requirements of the monitoring task.

3.2.2 Case Study 2

In this study, we examine the performance of each considered technique when every site is given a distinct operational profile or “personality.” To this end, we assign a range of weights and distinct input spaces to each site by suitably modifying the parameters of the simulation.

This second scenario is more representative of how real users might execute the software than is Study 1. This is because, in general, programs may be used by a variety of people in different contexts producing heterogeneous program behaviors.

The parameters of the simulation are set as follows:

- Number of deployed instances := 2,500
- Site weight := 50
- Site input space := 25
- Threshold for *REF-AG* := 10%

The diagram in Figure 2 shows the results for 50 iterations of the simulation for the four techniques. As in study one, technique *FULL* reports the coverage of 1,074 branches during the first iteration and no additional coverage is reported subsequently. It is important to note that, because of the way the simulation is performed, each iteration in study one represents 2,500 executions, whereas in study two each represents an average of 62,500 executions. Although we could normalize the results in terms of executions, this would not change the qualitative nature of the results.

Effectiveness. For this study, the presence of operational profiles changes the results with respect to the effectiveness of the three techniques. The curves for *SIMPLE* and *REF-RR* do not converge to the curve for *FULL*. *SIMPLE* does not converge because, without refinement and with different operational profiles, some of the probes are never executed in the instance to which they are assigned. *REF-RR*

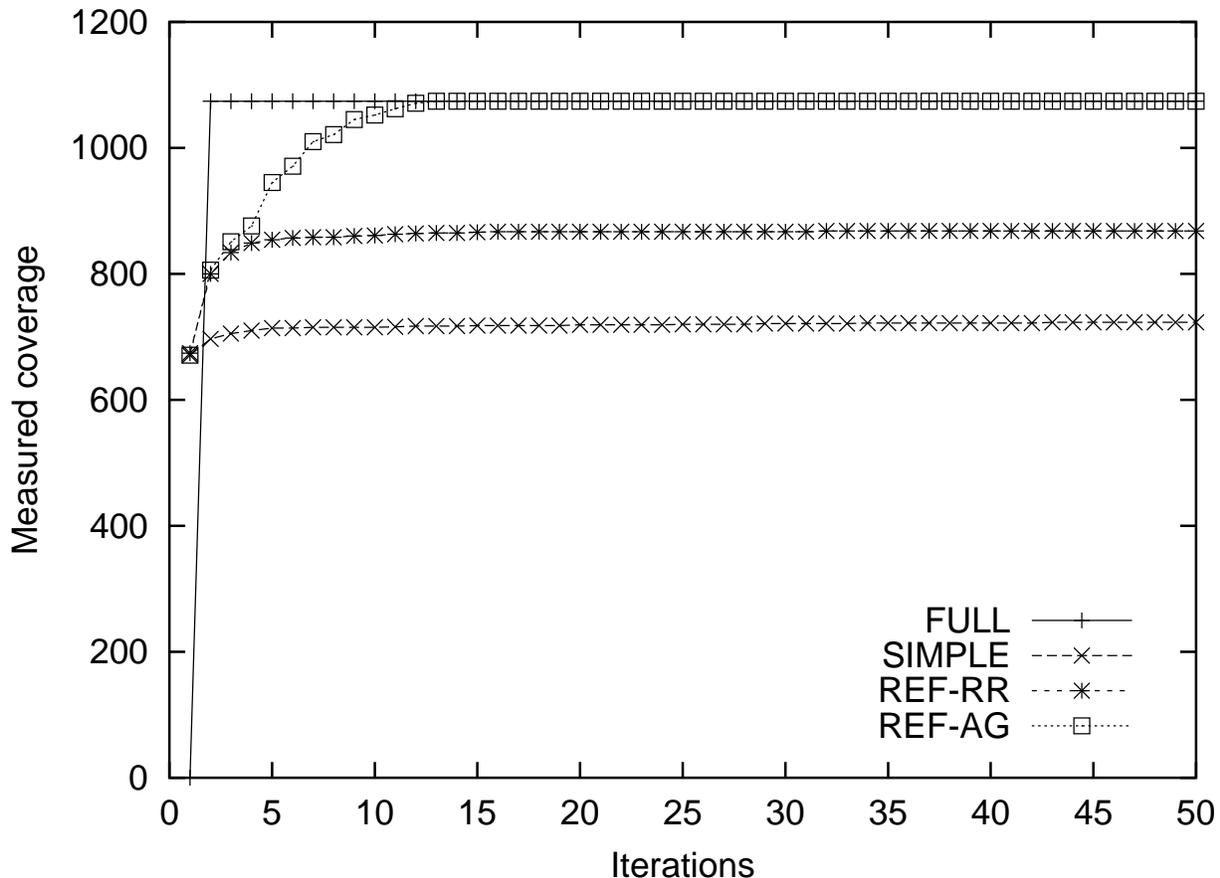


Figure 2: Results for Case Study 2.

does not converge because the rate of progress falls below the threshold that determines refinement termination. The curve for *REF-AG*, on the other hand, converges to the curve for *FULL* after 14 iterations.

Efficiency w.r.t. instrumentation. For this study, *REF-AG* reaches an optimum and terminates the refinement at the 14th iteration, after having performed 8 refinements. Therefore, the number of branches instrumented by the technique in this case is 0.7% of the total number of branches (8 over 1,228).

Efficiency w.r.t. interactions. In the case of *REF-RR*, the threshold for termination is reached after 13 iterations, when the number of branches reported as covered is 865. During the 13 iterations, the number of interactions with the deployment sites is 2,708. *REF-AG* interacts with the sites only up to the 14th iteration, when the refinement process terminates, and performs a total of 8,447 interactions. As in study 1, the improved performance of *REF-AG* is achieved at a cost of a higher number of interactions.

3.3 Limitations of the Studies

As with any empirical study, this study has limitations. We have considered the application of the tomographic techniques to a single program and we do not know whether the input space given by our test suite is representative of any actual input space that might be provided by real users. Therefore, although we used a real program, we cannot claim

generality for our results and additional studies are required to address such questions of external validity.

Other limitations involve internal and construct validity. In our studies, we assumed that dynamic updates of the instrumentation can always be performed instantaneously and with a fixed overhead. Latencies or hidden costs in the updates could affect the performances of the tomographic techniques. We also assumed that the overhead for a given probe is fixed. Depending on the location of the probe, the overhead associated with it may vary. For example, a probe in a loop that is executed a high number of times before the probe gets eliminated or deactivated comports a bigger overhead than a probe in a branch rarely executed.

3.4 Discussion

These two studies show that for the task of branch coverage detection using software tomography, where the goal is to measure coverage, the accuracy can be increased at the cost of additional interaction with the deployed instances. In fact, without a refinement strategy of some sort, the growth in the rate of detection for *SIMPLE* approaches zero before the task is complete. The cost of this incompleteness can be contrasted with that of a refinement strategy that will likely complete the task in a reasonable time. Although we have not formalized a cost model, there are two distinct cost components relative to refinement as we have presented it as a technique available to a software developer. There is

the cost to the developer of increased interaction with the deployed instances. There is the cost of increased runtime and/or space overhead for individual users. The developer will need to assess the impact of these cost components and then choose a position between no refinement and aggressive refinement.

This study of software tomography has been very productive for us in creating and validating a basic research framework for software tomography, which will guide future research. The case studies prepare us for investigations into the application of tomography to much more complex tasks for software deployed among real users.

4. CONCLUSION AND FUTURE WORK

In this paper, we have investigated the use of software tomography for the monitoring of deployed software. We have presented two techniques for collecting coverage information from the field for a set of instances of a program. We have implemented and validated the two presented techniques by performing two case studies using simulation. The result of the studies shows that, at least for the cases considered, software tomography can be successfully applied.

In the paper, we have not addressed problems related to dynamic updates of the running instances, by simply assuming the technology is available. In fact, there are many existing approaches for the dynamic update of instrumentation or code in general (e.g., [5, 8, 13, 18]). Moreover, new architectures such as Microsoft .Net [10] and Sun ONE [17] architectures may provide new opportunities to efficiently update software on a per-user basis. We are currently evaluating a prototype for the dynamic updating of instrumentation. We will use it to assess overhead and possible issues involved in tomographic refinement [13].

An important merit of our first set of studies is that they raised several issues and fueled further research. We are currently considering two main directions for future work. First, we will start collecting real operational-profile data, by distributing fully-instrumented versions of other software products. We will use the data collected to investigate the performance of software tomography using real users' operational profiles. Second, we will investigate refinement using probabilistic and data-mining models. An immediate step will be to investigate how we might improve tomographic refinement using a probabilistic model for probe placement based on the results from in-house testing.

Another area of future research is the applicability of software tomography to other monitoring tasks. We foresee that criteria related to operational profiles, such as determining feature use and detecting commonly used paths, will be good candidates. For each task, we will develop the algorithms for identifying subtasks and placing probes and design appropriate refinement techniques.

We believe that, in the long run, software tomography will provide for the collection of all sorts of dynamic information about deployed programs and the way they are used. Such information will be invaluable to software producers as an integral part of their software development processes. Interpretation of the collected data might provide feedback to inform the next product cycle, aid in diagnosing systems, yield insights for the design of new products, and inspire new development strategies.

Acknowledgments

This work was supported in part by National Science Foundation awards CCR-9988294, CCR-0096321, CCR-0205422, SBE-0123532 and EIA-0196145 to Georgia Tech, by the State of Georgia to Georgia Tech under the Yamacraw Mission, and by the Office of Naval Research through a National Defense Science and Engineering Graduate (NDSEG) Fellowship. Alberto Pasquini, Phyllis Frankl, and Filip Vokolos provided *Space* and many of its test cases. Gregg Rothermel provided many of the test cases, test suites, and coverage information used for the studies. The anonymous reviewers provided insightful comments and suggestions that helped us to improve the paper.

5. REFERENCES

- [1] T. Apiwattanapong and M. J. Harrold. Selective path profiling. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE'02)*, November 2002. (to appear).
- [2] Aristotle Research Group. ARISTOTLE: Software engineering tools, 2002. <http://www.cc.gatech.edu/aristotle/>.
- [3] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 46–57, Dec 1996.
- [4] Expectation-driven event monitoring (EDEM), 2002. <http://www.ics.uci.edu/~dhilbert/edem/>.
- [5] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, Feb 1996.
- [6] M. Harrold and G. Rothermel. Aristotle: A system for research on and development of program-analysis based tools. Technical Report Technical Report OSU-CISRC-3/97-TR17, The Ohio State University, Mar 1997.
- [7] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and Y. L. An empirical evaluation of the correlation between fault-revealing test behavior and differences in program spectra. *Journal of Software Testing, Verification, and Reliability*, 10(3), Sep 2000.
- [8] M. Hicks, J. Moore, and S. Nettles. Dynamic software updating. In C. Norris and J. J. B. Fenwick, editors, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, ACM SIGPLAN Notices, pages 13–23. ACM Press, Jun 2001.
- [9] D. M. Hilbert and D. F. Redmiles. Extracting usability information from user interface events. *ACM Computing Surveys*, 32(4):384–421, Dec 2000.
- [10] Microsoft Corporation. Microsoft .Net, 2002. <http://www.microsoft.net/net/>.
- [11] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. MacGraw-Hill (New York NY), 1987.
- [12] A. Orso, D. Liang, M. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '02)*, pages 65–69, Jul 2002.

- [13] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, Oct 2002. (to appear).
- [14] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 277–284, May 1999.
- [15] Perpetual testing, 2002.
<http://www.ics.uci.edu/~djr/edcs/>,
<http://laser.cs.umass.edu/perptest/>,
<http://www.cs.uoregon.edu/research/perpetual/edcs/Perpetual-Testing.html>.
- [16] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the Year 2000 problem. In *Proceedings of the 6th European Software Engineering Conference (ESEC'97)*, pages 432–449, Sep 1997.
- [17] Sun Microsystems. Sun[tm] Open Net Environment (Sun ONE), 2002.
<http://www.sun.com/software/sunone/>.
- [18] M. Tikir and J. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 86–96, Jul 2002.