Incorporating Performance Testing in Test-Driven Development

Michael J. Johnson and E. Michael Maximilien, *IBM*
Chih-Wei Ho and Laurie Williams, *North Carolina State University*

Performance testing can go hand-in-hand with the tight feedback cycles of test-driven development and ensure the performance of the system under development.

Retail customers have used IBM's point-of-sale system for more than a decade. The system software includes drivers for POS devices, such as printers and bar code scanners. The device drivers were originally implemented in C/C++. Later, developers added a Java wrapper around the C/C++ drivers to let Java programs interact with the POS devices. In 2002, IBM Retail Store Solutions reimplemented the device drivers primarily in Java to allow for cross-platform and cross-bus connectivity (that is, USB, RS232, and RS485 bus). A previous case study on the use of test-driven development in this reimplementation focused on overall quality.[1,2] It showed that TDD reduced the number of defects in the system as it entered functional verification testing.

Besides improving overall quality in this project, TDD led to the automation of performance testing.[3] Software development organizations typically don't begin performance testing until a product is complete. However, at this project's beginning, management was concerned about the potential performance implications of using Java for the device drivers. So, performance was a focal point throughout the development life cycle. The development team was compelled to devise well-specified performance requirements and to demonstrate the feasibility of using Java. We incorporated performance testing with the team's TDD approach in a technique we call *test-first performance* (TFP).

Test-first performance

We used the JUnit testing framework (www.junit.org) for both unit and performance testing. To get effective performance information from the test results, we adjusted some TDD practices.

In classic TDD, binary pass/fail testing results show an implementation's correctness. However, software performance is an emergent property of the overall system. You can't guarantee good system performance even if the individual subsystems perform satisfactorily.

Although we knew the overall system performance requirements, we couldn't specify the subsystems' performance before making some performance measurements. So, instead of using JUnit assert methods, we specified performance

0740-7459/07/$25.00 © 2007 IEEE

Team	
Size	Nine engineers
Experience (domain and programming language)	Some team members inexperienced
Collocation	Distributed
Technical leadership	Dedicated coach
Code	
Size	73.6 KLOC (9.0 base and 64.6 new)
Language	Java
Unit testing	Test-driven development

Figure 1. Project context.

scenarios in the test cases and used them to generate performance log files. Developers and the performance architect then examined the logs to identify performance problems.

We generated the logs with measurement points injected in the test cases. Whenever the execution hit a measurement point, the test case created a time stamp in the performance log file. Subsequently, we could use these time stamps to generate performance statistics such as the average and standard throughput deviation. We could enable or disable all of the measurement points via a single runtime configuration switch. We found the logs provided more information than the binary results and facilitated our investigation of performance issues.

Another difference between classic TDD and TFP is the test designer. In classic TDD, the developers implementing the software design the test cases. In our project, a performance architect who wasn't involved in coding but had close interactions with the developers specified the performance test cases. The performance architect was an expert in the domain with deep knowledge of the end users' expectations of the subsystems. The performance architect also had greater understanding of the software's performance attributes than the developers and so could specify test cases that might expose more performance problems.

For a TDD project to be successful, the test cases must provide fast feedback. However, one of our objectives was to understand the system's behavior under heavy workloads,[4] a characteristic that seemingly contradicts TDD. To incorporate performance testing into a TDD process, we designed two sets of performance test cases. One set could finish quickly and provide early warning of performance problems. The developers executed this set with other unit test cases. The other set needed to be executed in a performance testing lab that simulated a real-world situation. The performance architect ran this set to identify performance problems under heavy, simultaneous workloads or with multiple devices.

The development team consisted of nine full-time engineers, five in the US and four in Mexico. A performance architect in the US was also allocated to the team. No one had prior experience with TDD, and three were somewhat unfamiliar with Java. All but two of the nine full-time developers were new to the targeted devices. The developers' domain knowledge had to be built during the design and development phases. Figure 1 summarizes the project context.

Applying test-first performance

For ease of discussion, we divide the TFP approach into three subprocesses: *test design*, *critical test suite execution*, and *master test suite execution*.

Test design

This process involves identifying important performance scenarios and specifying performance objectives. It requires knowing the system's performance requirements and software architecture. In our case, the performance architect, with assistance from other domain experts, performed this process.

Our test design process has three steps: identify performance areas, specify performance objectives, and specify test cases.

Identify performance areas. You can specify software performance on many types of resources, with different measurement units. For example, elapsed time, transaction throughput, and transaction response time are among the most common ways to specify software performance. In our project, the important performance areas were

- the response time for typical input and output operations,
- the maximum sustainable throughput and response time for output devices, and
- the time spent in different software layers.

Specify performance objectives. We obtained the performance objectives from three sources.

The first was previous versions of similar device driver software. A domain expert identified the performance-critical device drivers. The objective for new performance-critical drivers was to exceed the current performance in the field.

The objective for non-performance-critical drivers was to roughly meet the current performance in the field.

The second source was the limitations of the underlying hardware and software components. These components' performance posed limiting factors beyond which performance improvement had little relevance. For example, figure 2 shows an execution time profile for a POS line display device. In this example, we could impact the performance for only JavaPOS (www.javapos.com), the javax.usb API (http://javax-usb.org), and the Java Native Interface (JNI) layers. The combined time spent in the underlying operating system kernel and in the USB hardware was responsible for more than half the latency from command to physical device reaction. As long as the device operation was physically possible and acceptable, we considered these latencies as forming a bound on the drivers' performance.

Finally, marketing representatives distilled some of the performance objectives from direct customer feedback. This input tended to be numerically inexact but did highlight specific areas where consumers most desired performance-related improvement from the previous release.

Specify test cases. In this final step, we designed performance test cases based on the specified objectives. This step included defining the measurement points and writing JUnit test cases to automatically execute the performance scenarios. We put the measurements points at the entry and exit points for each software layer. We put another measurement point at the lowest driver layer so that we could capture the time stamp of the event signal for the input from the Java virtual machine or native operating system. Performance test cases asserted additional start and stop measurement points.

We specified two performance test suites. The *critical test suite* consisted of a small subset of the performance test cases used to test individual performance-critical device drivers. The *master test suite* comprised the full set of all performance test cases.

Critical test suite execution

Developers ran the critical test suite on their own machines before committing their code to the repository. They executed the suite using Jacl (http://tcljava.sourceforge.net), an interactive scripting language that allows manual interactive exercising of the classes by creating objects and calling their methods. Figure 3a shows the critical-test-suite execution process.

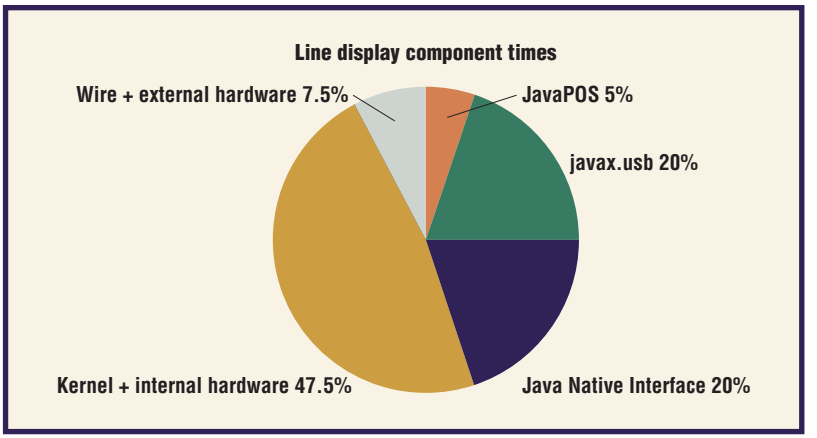

Figure 2. Some limiting factors imposed by the hardware and operating system layers.

Performance testing for a feature can't reasonably start until the feature passes the TDD unit and functional tests. (the gray boxes in figure 3a). After the TDD test cases passed, the developers ran the critical test suite to generate performance logs. This suite provides preliminary feedback on potential performance issues associated with an individual device. To provide timely feedback, test case responsiveness is important in TDD. However, obtaining proper performance measurements usually requires running the software multiple times in different operating environments, sometimes with heavy workloads. Consequently, performance test cases tend to be slower than functional test cases. To get quick feedback, the critical suite focused only on quick testing of the performance-critical drivers.

In classical TDD, developers rely on the "green bar" to indicate that an implementation has passed the tests. When a test case fails, the development team can usually find the problem in the newly implemented feature. This *delta debugging*[5] is an effective defect-removal technique for TDD. However, in performance testing, a binary pass/fail isn't always best. We wanted developers to have an early indication of any performance degradation caused by a code change beyond what a binary pass/fail for a specific performance objective could provide. Additionally, hard performance limits on the paths being measured weren't always available in advance. So, the developers had to examine the performance log files to discover performance degradation and other potential performance issues. The performance problem identification and removal process thus relied heavily on the developers' expertise. After addressing the performance issues,

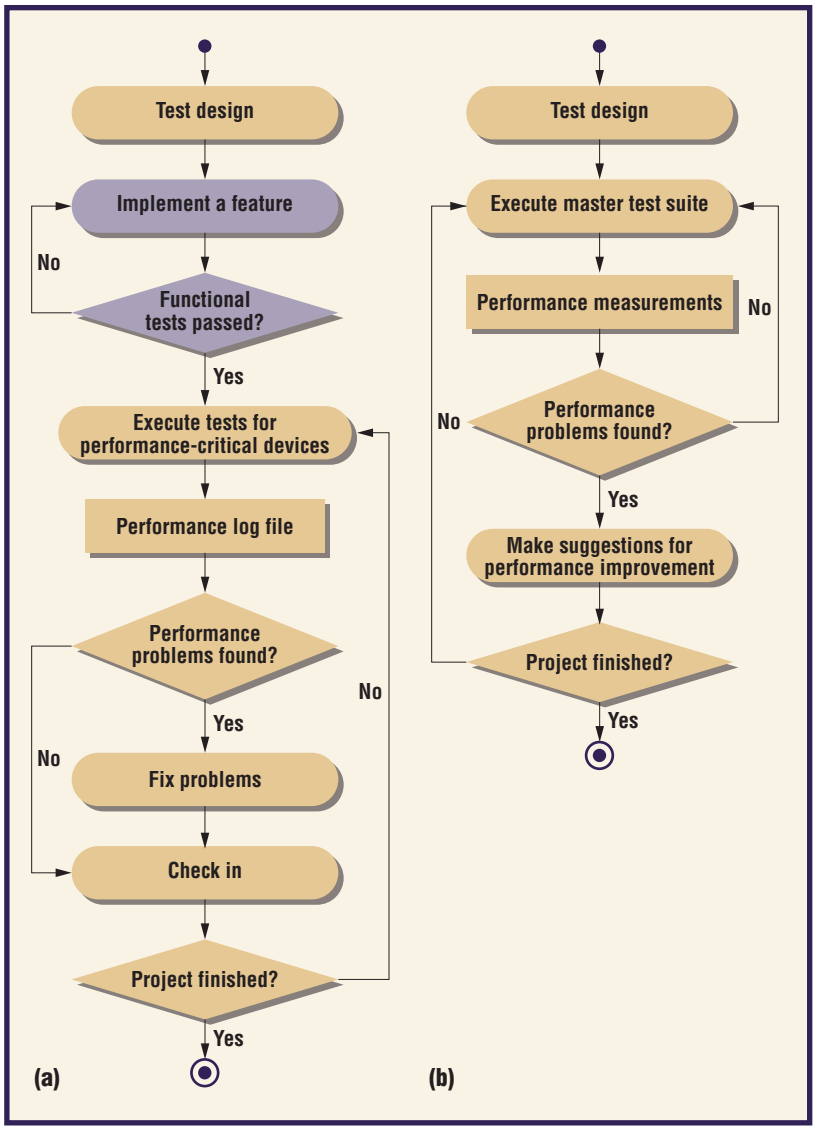

Figure 3. The process for (a) critical suite execution and (b) master suite execution.

the developers checked in the code and began implementing another new feature.

Master suite execution

The master suite execution included these performance test cases:

- *Multiple runs*. Many factors, some that aren't controllable, can contribute to software performance. So, performance measurements can differ each time the test cases are run. We designed some test cases to run the software multiple times to generate performance statistics.
- *Heavy workloads*. Software might behave differently under heavy workloads. We designed some test cases with heavy workloads to measure the software's responsiveness and throughput under such circumstances.
- *Multiple devices*. The interaction among two or more hardware devices might cause unforeseen performance degradations. We designed some test cases to exercise multiple-device scenarios.
- *Multiple platforms*. We duplicated most test cases to measure the software's performance on different platforms and in different environments.

Running these test cases could take significant time, and including them in the critical suites wasn't practical. Additionally, it was more difficult to identify the causes of the performance problems found with these test cases. We therefore included these test cases in the master test suite only.

The performance architect ran the master suite weekly in a controlled testing lab. Figure 3b shows this process.

The performance architect started running the master suite and collecting performance measurements early in the development life cycle, as soon as enough functionality had been implemented. After collecting the measurements, the performance architect analyzed the performance log file. If the performance architect found significant performance degradation or other performance problems, he would perform further analysis to identify the root cause or performance bottleneck. The architect then highlighted the problem to developers and suggested improvements when possible. The testing results were kept in the performance record, which showed the performance results' progress.

A key element of this process was close interaction between the development team and the performance architect. The performance architect participated in weekly development team meetings. In these meetings, the development team openly discussed the key changes in all aspects of the software. Additionally, because team members were geographically distributed, the meetings let everyone freely discuss any dependencies they had with other members from the remote sites and plan one-on-one follow-up. The performance architect could use the knowledge gained from the meetings to determine when a subsystem had undergone significant changes and thus pay more attention to the related performance logs.

Results

By the project's end, we had created about 2,900 JUnit tests, including more than 100 performance tests.[3] Performance-testing results showed improvement in most intermediate snapshots. We sometimes expected the opposite trend, because checking for and handling an increasing number of exception conditions made the code more robust. In practice, however, we observed that the throughput of successive code snapshots generally did increase.

Figure 4 shows an example of receipt printer throughput in various operating modes and retail environments. Three sets of results show throughput with asynchronous-mode printing:

- a short receipt typical of a department store environment (ADept),
- a long receipt typical of a supermarket (ASmkt), and
- a very long printout (MaxS).

A fourth set (SDept) shows throughput with synchronous-mode printing in a department store environment.

A code redesign to accommodate the human-interface device specification resulted in a significant throughput drop between snapshots S1 and S2. However, the development team received early notice of the performance degradation via the ongoing performance measurements, and by snapshot S3 most of the degradation had been reversed. For the long receipts where performance is most critical, performance generally trended upward with new releases.

We believe that the upward performance trend was due largely to the continuous feedback provided by the TFP process. Contrary to the information provided by postdevelopment optimization or tuning, the information provided by the in-process performance testing enabled a feedback-induced tendency to design and code for better performance.

Another indication of our early measurement approach's success is its improvement over previous implementations. We achieved the Java support for an earlier implementation simply by wrapping the existing C/C++ drivers with the JNI. This early implementation was a tactical interim solution for releasing Java-based drivers. However, customers used this tactical solution,

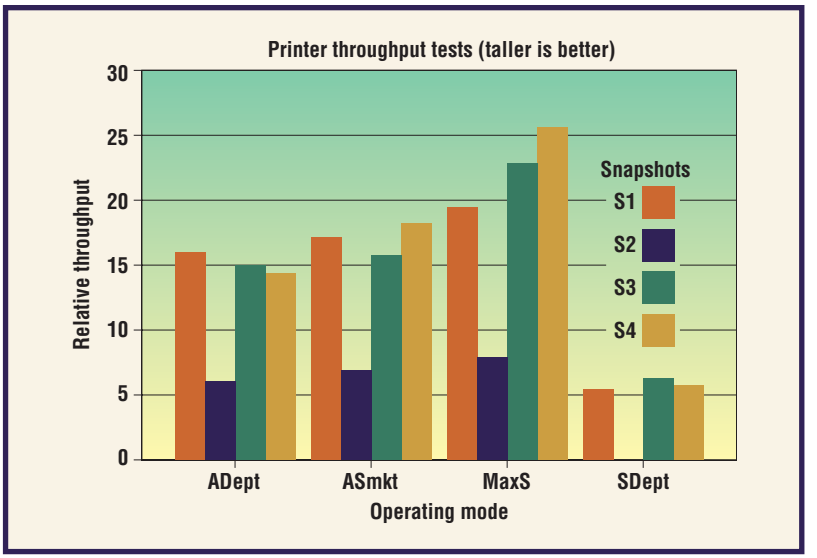

Figure 4. Performance tracking using snapshots of receipt printer throughout tests. Three sets of results show throughput with asynchronous-mode printing: a short receipt typical of a department store environment (ADept), a long receipt typical of a supermarket (ASmkt), and a very long printout (MaxS). The fourth set (SDept) shows throughput with synchronous-mode printing in a department store environment.

and we could have kept it as the mainstream offering if the new drivers didn't match its performance levels. The tactical drivers' performance was a secondary consideration and was never aggressively optimized, although the underlying C/C++ drivers were well tuned. The existence of the JNI-wrapped drivers let us run the same JUnit-based performance test cases against both the JNI-wrapped drivers and the Java drivers for comparison. The tactical drivers' performance provided a minimum acceptable threshold for performance-critical devices.

Figure 5a shows results for the set of throughput tests in figure 4, this time comparing snapshot S3 to the previous implementation. The overall results were better with the new code. Figure 5b shows the improvement in operation times of two less performance-critical device drivers. The line display was an intermediate case in that it didn't receive the intense performance design scrutiny afforded the printer, although bottom-line operation time was important. We considered the cash drawer operation time noncritical.

Lessons learned

Applying a test-first philosophy in software performance has produced some rewarding results.

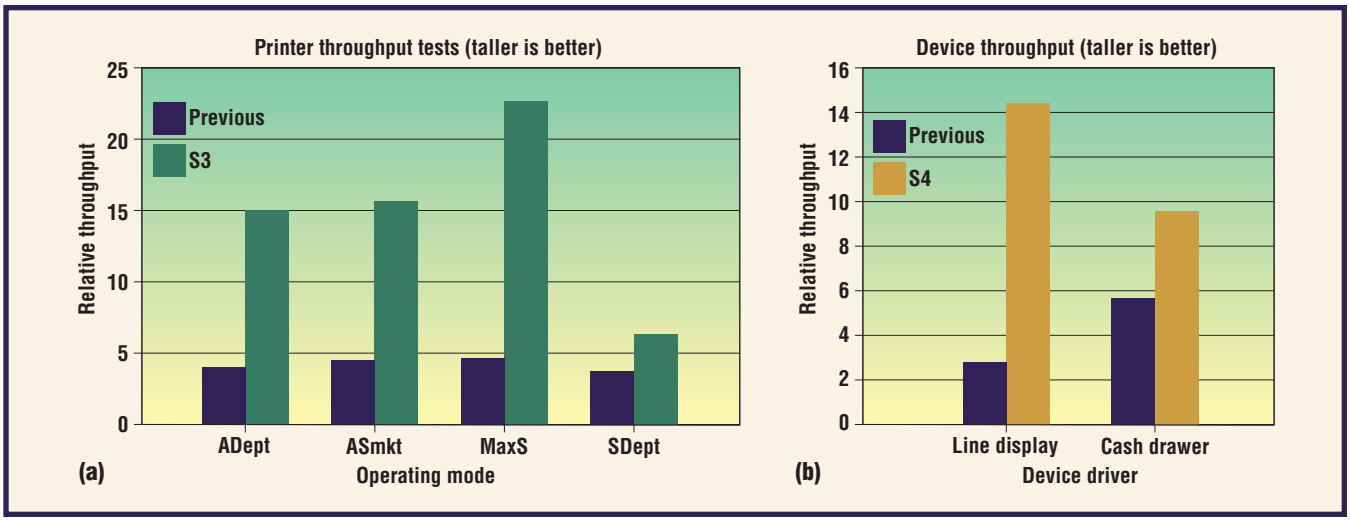

Figure 5. A comparison to the previous implementation of (a) performance-critical printer tasks and (b) less critical and noncritical device tasks.

First, when we knew performance requirements in advance, specifying and designing performance test cases before coding appears to have been advantageous. This practice let developers know, from the beginning, those operations and code paths considered performance critical. It also let them track performance progress from early in the development life cycle. Running the performance test cases regularly increased developers' awareness of potential performance issues. Continuous observation made the developers more aware of the system's performance and could explain the performance increase.[6]

We also found that having a domain expert specify the performance requirements (where known) and measurement points increased overall productivity by focusing the team on areas where a performance increase would matter. This directed focus helped the team avoid premature optimization.

The performance architect could develop and specify performance requirements and measurement points much more quickly than the software developers. Also, limiting measurement points and measured paths to what was performance sensitive in the end product let the team avoid overinstrumentation. Having a team member focused primarily on performance also kept the most performance-critical operations in front of the developers. This technique also occasionally helped avoid performance-averse practices, such as redundant creation of objects for logging events.

A third lesson is that periodic in-system measurements and tracking performance measurements' progress increased performance over time.

This practice provided timely notification of performance escapes (that is, when some other code change degraded overall performance) and helped isolate the cause. It also let developers quickly visualize their performance progress and provided a consistent way to prioritize limited development resources onto those functions having more serious performance issues. When a critical subsystem's performance was below the specified bound, the team focused on that subsystem in the middle of development. The extra focus and design led to the creation of an improved queuing algorithm that nearly matched the targeted device's theoretical limit.

Our performance-testing approach required manually inspecting the performance logs. During the project's development, JUnit-based performance testing tools, such as JUnitPerf, weren't available. Such tools provide better visibility of performance problems than manual inspection of performance logs. Although we believe manual inspection of performance trends is necessary, specifying the bottom-line performance in assert-based test cases can complement the use of performance log files, making the TFP testing results more visible to the developers. We're investigating the design of assert-based performance testing to improve the TFP process.

Another direction of future work is automatic performance test generation. In this project, we relied on the performance architect's experience to identify the execution paths and measurement points for performance testing. We can derive this crucial information for performance testing from the performance requirements and system design. We plan to find guidelines for specifications of performance requirements and system design to make the automation possible.

Acknowledgments

We thank the IBM device driver development team members, who implemented the performance tests described in this article. We also thank the RealSearch reading group at North Carolina State University for their comments. IBM is a trademark of International Business Machines Corporation in the United States or other countries or both.

References

1. E.M. Maximilien and L. Williams, "Assessing Test-Driven Development at IBM," *Proc. 25th Int'l Conf. Software Eng.* (ICSE 03), IEEE CS Press, 2003, pp. 564–569.
2. L.E. Williams, M. Maximilien, and M.A. Vouk, "Test-Driven Development as a Defect Reduction Practice," *Proc. 14th Int'l Symp. Software Reliability Eng.*, IEEE CS Press, 2003, pp. 34–35.
3. C.-W. Ho et al., "On Agile Performance Requirements Specification and Testing," *Proc. Agile 2006 Int'l Conf.*, IEEE Press, 2006, pp. 47–52.
4. E.J. Weyuker and F.I. Vokolos, "Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study," *IEEE Trans. Software Eng.*, vol. 26, no. 12, Dec. 2000, pp. 1147–1156.
5. A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, 2002, pp. 183–200.
6. G.M. Weinberg, *The Psychology of Computer Programming*, Dorset House, 1998, pp. 31.

About the Authors

Michael J. Johnson is a senior software development engineer with IBM in Research Triangle Park, North Carolina. His recent areas of focus include performance analysis, performance of embedded systems, and analysis tools for processors. He received his PhD in mathematics from Duke University. He's a member of the IEEE and the Mathematical Association of America. Contact him IBM Corp., Dept. YM5A, 3039 Cornwallis Rd., Research Triangle Park, NC 27709; mjj@us.ibm.com.

Chih-Wei Ho is a PhD candidate at North Carolina State University. His research interests are performance requirements specification and performance testing. He received his MS in computer science from NCSU. Contact him at 2335 Trellis Green, Cary, NC 27518; dright@acm.org.

E. Michael Maximilien is a research staff member in the Almaden Services Research group at the IBM Almaden Research Center. His research interests are distributed systems and software engineering, with contributions to service-oriented architecture, Web services, Web 2.0, service mashups, and agile methods and practices. He received his PhD in computer science from North Carolina State University. He's a member of the ACM and the IEEE. Contact him at the IBM Almaden Research Center, 650 Harry Rd., San Jose, CA 95120; maxim@us.ibm.com.

Laurie Williams is an associate professor at North Carolina State University. Her research interests include software testing and reliability, software engineering for security, empirical software engineering, and software process, particularly agile software development. She received her PhD in computer science from the University of Utah. She's a member of the ACM and IEEE. Contact her at North Carolina State Univ., Dept. of Computer Science, Campus Box 8206, 890 Oval Dr., Rm. 3272, Raleigh, NC 27695; williams@csc.ncsu.edu.