

# Research Statement

Kristen R. Walcott-Justice

---

Recent advances in computer architecture have brought new opportunities for producing more efficient and effective software development tools. These developments include multicore processors and support for hardware event sampling across a wide range of hardware mechanisms. Most commodity computers that are built today, including smaller devices such as tablets, smart phones, and netbooks, have these features. Multicore processors can enable more efficient program analysis and modification, while hardware mechanisms enable architectural event monitoring at very low cost.

The goal of my research is to develop more efficient and effective testing techniques by taking advantage of these hardware advances. The quality of a set of structural tests and test suites is determined using test coverage metrics, which are traditionally analyzed by instrumenting every program element under consideration that is reached during program execution. Even monitoring simple structures such as branches or statements presents a number of challenges including the compile time, runtime, and code-size overheads incurred by instrumentation. When monitoring large scale programs or more complex structures for data-flow or paths, the overall cost of instrumenting can become prohibitive in time and space, especially in resource constrained environments such as handheld devices. Instrumentation also is impractical for monitoring multithreaded or time-sensitive programs, in which additional probe and payload code may perturb normal execution.

In my doctoral research, I address challenges caused by using instrumentation in software testing by exploiting hardware performance monitors and multicore technology. To evaluate the quality of test suites in an efficient and effective way, I developed a system called THEME: Testing by HardwarE Monitoring Events. In the first component of my research, I evaluated the tradeoffs of leveraging hardware monitors and multicore technology in testing. I then applied my hardware monitoring techniques in two different environments: testing on restricted memory devices and testing multithreaded programs.

## **Improving Efficiency of Test Coverage Monitoring**

Hardware mechanisms and multicore technology have been used successfully to monitor and analyze data in many areas of software development including debugging, path profiling, and race detection. However, the use of these hardware advances has not been adequately explored for testing software.

Testing depends on the monitoring of runtime events, which is traditionally performed by instrumentation. Instrumentation, however, can be costly and intrusive to insert and execute. Hardware mechanisms, on the other hand, can be used with very little overhead or disruption because they typically use in-CPU registers, which can be quickly set up and read. Because hardware monitoring can remove the need for instrumentation, hardware mechanisms have the potential to be used for testing with negligible code growth and time overhead relative to the original program.

A key challenge in performing monitoring for coverage analysis using hardware mechanisms lies in collecting all events with which we are concerned and only those events from the binary code. However, hardware mechanisms are designed for performance evaluation and for sampling from all events executed on the CPUs. Thus, when monitoring binary code for testing, hardware sampling must be performed frequently and carefully to improve the likelihood of specific events being observed.

To examine the trade offs among sampling rate, coverage completeness, and the time overhead and code growth incurred, I developed a runtime system that performs coverage analysis by monitoring a number of hardware mechanisms on single and multiple cores. My system, THEME, consists of a runtime system and static components that monitor partial runtime behavior and extend the amount of coverage derived from each sample.

I observed that monitoring program execution using hardware mechanisms was up to 11.13% faster compared to using instrumentation. The results showed that up to 90% of the actual code coverage can be determined with low time overhead and only 0.5% code growth. My paper introducing this work was published in the ICSE 2011 NIER track, and a paper describing the THeME system is in submission.

### **Testing in Restricted Memory Environments**

Because my hardware approaches to monitoring test case coverage require no alterations to the program under test, they are ideal in memory-constrained environments where testing generally must be performed by simulation. In traditional test coverage analysis, the code growth incurred by instrumentation is impacted by the size of the probe and payload and the frequency of insertion, and in branch testing, average code growth ranges from 60% to 90%. On low memory devices whose program execution may already be restrained by the lack of resources, instrumentation's high code growth and memory overhead can make testing on the device infeasible.

Hardware mechanisms, however, require little or no program modification to perform monitoring, and processors used in low memory devices generally include hardware monitors and counters that can be exploited. For example, many newer smart phones and tablets run on a TI OMAP 4430 processor, which has more than fifty hardware performance counters and mechanisms that are accessible at the user and kernel levels. By taking advantage of these mechanisms, test monitoring can be enabled even for programs that push the device's memory constraints.

I am currently adapting the THeME system to execute on a Kindle Fire, which uses the OMAP 4430 processor and has only 512 MB of RAM. As a tablet, the Kindle Fire is designed to run potentially large, memory intensive applications, making testing on the device challenging. The THeME system will enable testing of such programs with low overhead.

### **Testing Multithreaded Applications**

My hardware monitoring runtime system is also advantageous when testing multithreaded programs. Traditional instrumentation approaches involve additional probe and payload code that may perturb normal program execution. Through the use of hardware mechanisms, however, snapshots of instruction execution can be taken on each core on which the program is executing without intruding on normal program behavior. I am currently investigating how my system can be adapted to testing multithreaded programs that execute on multiple cores.

### **Future Work**

In my future work, I intend to explore ideas related to my current research in software quality, particularly in the areas of testing and debugging. To improve coverage monitoring, I would like to combine hardware sampling with dynamic instrumentation to improve the effectiveness of my monitoring techniques. My current research focuses primarily on branch testing, but I also plan to apply my hardware monitoring schemes to other coverage metrics for sequential and multithreaded programs.

Based on bug characteristic reports, numerous coverage metrics and analysis tools have been discussed as approaches to testing multithreaded programs. While these seem promising on the surface, these metrics and approaches have yet to be tested to see to what extent they reveal behavior that is related to concurrency specific bugs. My THeME system will be particularly advantageous when tracking events that are related to synchronization based coverage criteria.

Another research direction that will broaden the impact of my hardware monitoring runtime system is in the area of fault localization. Several hardware mechanisms exist that enable partial or full paths of execution to be observed. By incrementally sampling partial path information during execution based on inserted triggers, a path can be constructed that leads up to the execution of suspicious code blocks. Additionally, hardware mechanisms can be useful in debugging and fault localization by monitoring multiple events at both the user and kernel levels. Instead of analyzing only a section of program execution, hardware mechanisms can be

used to report events that occur outside the source, such as in library calls and external routines, painting a much fuller picture of program execution.

These are just a few of the many research challenges in software testing and debugging that I believe could benefit from applying hardware monitoring related techniques. I am very excited about the possibilities and the importance of addressing these challenges, and I look forward to contributing to the critical fields of software testing and debugging.