THeME: A System for Testing by Hardware Monitoring Events

Kristen Walcott-Justice
University of Virginia
Charlottesville, VA USA
walcott@cs.virginia.edu

Jason Mars
University of Virginia
Charlottesville, VA USA
jom5x@cs.virginia.edu

Mary Lou Soffa
University of Virginia
Charlottesville, VA USA
soffa@cs.virginia.edu

ABSTRACT

The overhead of test coverage analysis is dominated by monitoring the application, which is traditionally performed using instrumentation. However, instrumentation can prohibitively increase the time and especially the memory overhead of an application. As an alternative to instrumentation, we explore how recent hardware advances can be leveraged to improve the overheads of test coverage analysis. These hardware advances include hardware performance monitors and multicore technology.

In this work, we present our system, THeME, a testing framework that replaces instrumentation with hardware monitoring. THeME consists of a runtime system that takes advantage of hardware mechanisms and multiple cores and a static component to further extend the coverage derived from hardware event sampling. The results show that up to 90% of the actual coverage can be determined with less time overhead and negligible code growth compared to instrumentation.

Categories and Subject Descriptors

D.2.5 [**Software**]: Software Engineering—*Testing and Debugging - Monitors, Testing Tools*

General Terms

Performance, Measurement, Algorithms, Experimentation

Keywords

Software testing, hardware, performance monitoring

1. INTRODUCTION

Structural testing is one of the most commonly used classes of program testing strategies. The quality of a set of structural tests and test suites is determined using test coverage metrics, which are analyzed by monitoring selected program elements reached during program execution. Unfortunately, even monitoring simple structures, such as branches, presents a number of challenges. These challenges include the compile time, runtime, and code-size overheads incurred by monitoring.

The overhead of test coverage analysis is dominated by the cost of monitoring program execution, which is generally enabled using code instrumentation. To instrument code, the program is analyzed, either statically or dynamically, to determine code points of interest. Each point is marked by a probe, which is usually a jump or call to payload code that analyzes the monitored information. Usually the code inserted into the executable unnecessarily remains throughout execution, further increasing its expense. The time overhead and code growth from instrumentation can be high, even when monitoring simple structures. For example, the time overhead of using instrumentation for branch testing has been reported to be, on average, between 10% to 30%, with code growth ranging from 60% to 90% [11, 20, 23].

In many cases, these overheads are restrictive. As an example, it is not uncommon for regression test suites to require days or weeks to run to completion, with expensive equipment and engineering costs associated [8, 21, 27]. If a test suite requires 24 hours to execute, it would likely necessitate an extra 2.4 to 7.2 hours to evaluate the quality of the test suite based on branch coverage. When monitoring large scale programs or more complex structures, such as data-flow or paths, the overall cost of monitoring grows and can become prohibitive in time and space, especially in resource constrained environments. Instrumentation also is impractical for monitoring multithreaded or time-sensitive programs, in which additional probe and payload code may perturb normal execution.

Other software development tasks such as path profiling, trace selection, race detection, and dynamic optimization have also been riddled by these challenges, as they also rely on monitoring application behavior. However, in these areas, there is an emerging trend to leverage hardware performance monitoring mechanisms and multicore technology to mitigate and eliminate these challenges [3, 5, 6, 7, 18, 22, 24]. For example, research by Chen et al. [6] shows that profile information can be constructed efficiently and effectively by sampling hardware events. In their work, event monitoring incurred runtime overhead of only 2% and no code growth compared to compiler-based instrumentation, which suffered 10x time overhead over native execution.

Despite the success in these areas, advances in hardware monitoring and multicore technology has not been fully exploited in software testing. Nearly all commodity desktop, laptop, tablet, and mobile devices now available contain processors that support hardware monitoring. Many of these processors also include advanced hardware monitoring components that can provide large amounts of event information. Multicore technology is additionally now common on such devices. Through the combination of hardware performance monitors and multiple cores, program execution can potentially be monitored and recorded more efficiently for testing than when using traditional instrumentation techniques.

Compared to instrumentation, the use of hardware mechanisms is attractive as they can perform monitoring with very little overhead, and their use can remove the need for instrumentation. When monitoring using hardware mechanisms, a counter and mechanism need only be set up once per core during test execution, and reading of the mechanism is inexpensive. For example, Dey et al. [9] report that the initial setup for a counter takes approximately $318\mu s$, and reading a counter value takes only $3.5\mu s$ on average. In addition, using hardware performance monitors in lieu of instrumentation incurs no code growth and does not require recompilation.

Hardware counters can be configured on each processor core to increment when certain hardware events occur, providing count information, or they can be used for sampling. When a sample is taken, performance monitoring software records the system state including the current instruction information, register contents, etc. Such sampled information is extremely useful in areas such as profiling or dynamic optimizations because the samples can be used to estimate profiles or partial program behavior [6].

Software testing, however, relies on more exact execution information. For example, in branch testing, instrumentation is used to monitor *all* source code level branches with which the tester is concerned and monitor *only* those branches. While hardware mechanisms tracking a particular event will observe all events of that type during program execution, sampled data may miss certain events such as infrequently executed branches. Also, the use of hardware mechanisms implies that samples are likely to include branches that are not associated with the test program such as those in setup, teardown, or library code. Although recording hardware events is essentially free, there is a cost associated with reading the values from the hardware. Therefore, a balance must be found between the amount of information collected and the total overhead of sampling.

To evaluate the balance between efficiency and effectiveness of testing using hardware performance monitors and multicore technology, we designed a system called THeME: Testing by Hardware Monitoring Events, which consists of a runtime system and static components. In this work, we explore the use of hardware mechanisms and multicore technology in branch testing, and we thoroughly evaluate the tradeoffs of leveraging these technologies for branch monitoring. We first evaluate a pure hardware approach to branch testing. In this exploration, we investigate two ways of accessing and reading hardware mechanisms, namely through OS polling and OS interrupts. Analysis of our techniques demonstrates the efficiency achieved when calculating coverage information by sampling hardware. Additionally, we evaluate how performing branch testing using hardware sampling affects the completeness of coverage monitoring. Next, we analyze the effects of integrating hardware monitoring information with the compiler infrastructure, which improves the completeness of coverage monitoring through the use of static analysis techniques. Finally, we explore how multiple cores can be used in conjunction with hardware monitoring to improve the time overhead of structural testing.

We present empirical evidence that hardware monitoring can be adapted for more efficient branch coverage analysis compared to using instrumentation. Although hardware mechanism sampling leads to lossy test coverage information, it provides a low-overhead alternative to program instrumentation and can be used along with static analyses to attain upwards of 90% of the actual code coverage information. Hardware monitoring also requires only minor or no alterations to the program under test, making hardware approaches ideal in memory constrained environments, such as tablets or mobile devices. Our techniques also enable the testing of multithreaded and time-sensitive code.

In summary, the important contributions of this paper are as follows:

- The design and development of THeME, a runtime system to perform testing using hardware monitoring mechanisms.
- Techniques to monitor branch information using hardware mechanisms that incur lower overhead than instrumentation with negligible code growth.
- An empirical evaluation demonstrating the tradeoffs associated with monitoring using hardware mechanisms compared to full software-level instrumentation.
- An analysis revealing the benefits of testing using hardware mechanisms on multiple cores.
- A demonstration of how the compiler infrastructure can be used along with hardware mechanism monitoring for improved test coverage.
- A discussion of the key advantages of leveraging hardware advances in testing and of system software and hardware advances that can help to better leverage hardware mechanisms for structural testing purposes.

The remainder of this paper is organized as follows. In Section 2, we discuss the background and details of the hardware advances used in this work. We then describe the THeME system and its design in Section 3. Next, in Section 4, we explain the implementation details of the system and demonstrate the trade-offs between efficiency and effectiveness when monitoring using hardware sampling and static analyses. In Sections 5, we go over the key advantages that we observe when leveraging hardware advances in testing, and in Section 6, we discuss several system software and hardware advances that could benefit structural testing when exploiting hardware advances. Related work is discussed in Section 7. Finally, in Section 8, we conclude and suggest directions for future research in this area.

2. BACKGROUND

In this section, we discuss the recent hardware advances leveraged in this work. These include hardware counters and multicore technology, as well as a hardware feature known as the Last Branch Record.

Most microprocessors used in computers, tablets, and mobile devices now support hardware event sampling through

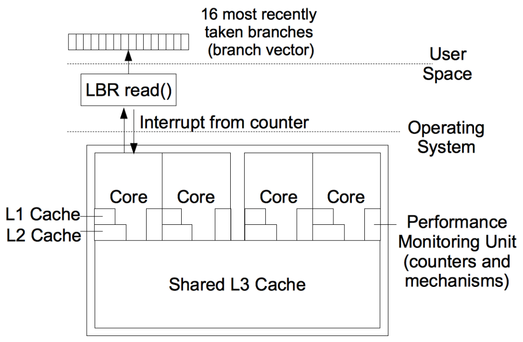

Figure 1: Recording and sampling using the LBR on a 4 core processor.

hardware counters. For example, the Intel Nehalem processor provides the capability to track more than 2000 different performance counter events, and recent Linux kernel patches provide user-level support for nearly 200 of these counters [10]. Hardware counters can be configured to increment on events such as each clock cycle, each time an instruction retires, for every L2 cache miss, etc. When sampling, the performance monitoring unit is configured to generate an interrupt whenever a hardware counter overflows, based on a user-defined value. When the interrupt is triggered, the Instruction Pointer and other register contents are recorded. This information can identify the instruction that caused the sample to be recorded.

Multicore technology is also now ubiquitous on commodity machines. Most commercial computers ship with two or four cores per chip, and recent mobile devices and tablets have two cores per chip. State-of-the-art computers and servers have twelve or more cores per chip. Often, these cores are left idle. However, each core per processor chip has its own set of hardware counters, performance monitors, and mechanisms that can be leveraged during program execution. In current hardware implementations, each core can access only the hardware mechanisms associated with that core. However, as can be seen in Figure 1, cores on a chip have both private caches and share a cache, typically the L2 or L3 cache, and thus can communicate with low overhead for improved monitoring and analysis.

2.1 Branch Vector Recording

On nearly all processors, both single and multicore, there are many hardware counters such as *Cache Misses* or *Branch Instructions Retired* that can be taken advantage of. In addition to these traditional hardware counters, more advanced hardware mechanisms have been introduced in recent processors to enable debugging and precise event reporting. In this paper, we focus on the Last Branch Record (LBR), which is a hardware feature of the performance monitoring unit on many modern microprocessors. The LBR was intended as a profiling tool for sampling partial branch paths in the operating system.

When the LBR is activated, the processor records a running trace of the most recent branches, interrupts, and exceptions taken by the processor. Each branch edge is represented by source and destination addresses and stored into a pair of LBR registers. The LBR is a circular set of registers that contains the last n taken branches, where n is dependent on the processor being used [12]. In Intel Nehalem processors, for example, the LBR records the last 16 branches [12]. The n branches contained in the LBR at any point define a *branch vector*.

2.2 Branch Vector Sampling

The LBR is used in sampling mode in conjunction with a hardware counter. As seen in Figure 1, when the hardware counter generates an interrupt based on a defined interrupt threshold, the n branches in the LBR can be accessed. Traditional hardware counters such as *Instructions Retired* or *CPU Cycles* can only record one instruction, the single instruction that caused the interrupt to take place. However, when the LBR is also enabled, a full branch vector of information (e.g. 16 branches on recent processors) is reported per interrupt. The branch vector recorded in the LBR registers represents a partial path of program execution through the branches.

Figure 1 shows how a branch vector can be recorded from a single processor into user space. The operating system first throws an interrupt saying that a counter has overflowed and that the LBR is ready to be read. When the user-level program then requests the data, the operating system reads the n LBR registers, returns the array to user space, and continues program execution until the next counter overflow.

3. THE DESIGN OF THEME

In this research, we developed a system called THeME: Testing by Hardware Monitoring Events, which enables us to evaluate the potential of using a hardware approach for structural testing. While there are many hardware mechanisms that can be used to monitor program behavior for different test metrics, in this paper, we focus on mechanisms that are appropriate for use in branch testing. Specifically, we exploit the sample data available from the LBR.

An overview of our THeME system is shown in Figure 2. THeME has three main components: the first is a static program modification and analysis tool. This tool is used to enable fall-through branch edge visibility for branch-based hardware mechanisms. Then, once the program under test has been modified, a simple static analysis is used to identify the branch edges in the program's source code, as in traditional testing techniques. The branch edges are stored in a hash table along with information pertaining to the associated source code lines. This branch table is used as a checklist of branches with which we are concerned and is later used to calculate overall branch coverage. The second component performs runtime hardware monitoring of the program under test by sampling the LBR. The final component performs an optional static analysis and calculates the sampled branch coverage. The branch coverage is based on the number of source code level branches observed compared to the total number of source code level branches in the program.

We now discuss the design of each component in more detail.

3.1 Enabling Fall-through Visibility

Independent of sampling technique, branch-based hardware mechanisms alone cannot observe when fall-through branches have occurred, which would lead to low coverage monitoring effectiveness. In branch testing, a tester wants to ensure that both edges are taken through a branch. For

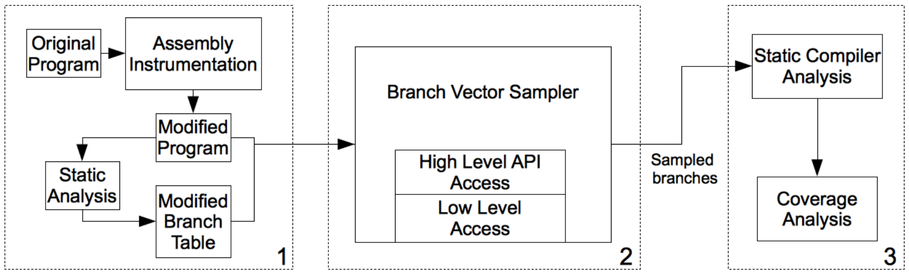

Figure 2: The THeME System

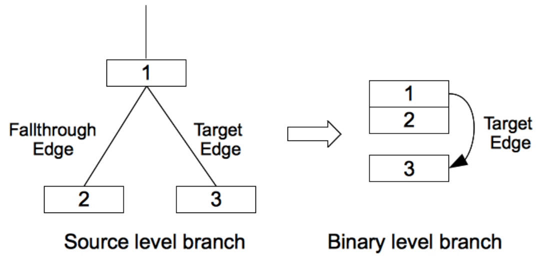

Figure 3: The LBR is incapable of detecting the fall-through branch edge from 1 to 2.

example, in Figure 3, monitoring should be able to detect both the execution of the fall-through path from 1 to 2 and the target path from 1 to 3. While this is obvious when looking at a flow graph, in the binary code, a branch is made up of some kind of jump to a target followed by another instruction. The LBR will report the jump from 1 to 3 but not the fall-through from 1 to 2. Therefore, the LBR by itself is only capable of monitoring 50% of the source level branches.

Fall-through branch observation is possible in several ways. One technique is to supplement the information from branch-based monitoring with other event data. For example, the `INST_RETIRED` event could be polled in addition to the LBR to look for fall-through instruction execution. Another technique to detect fall-through branches includes a static post mortem analysis of the program and observed information. These techniques would require no code modification, re-compilation, or code growth. However, because we want to evaluate the capabilities of using the hardware mechanisms for monitoring, we instead give the branch-based mechanism the potential to observe the fall-through path by inserting harmless unconditional branches along every fall-through edge in the binary, as pictured in Figure 2 Box 1. This is different from instrumentation, which is heavy weight and includes both probe and payload code. Our fall-through enabling technique adds only a single instruction along fall-through branch edges. Using this technique, negligible code growth is incurred.

3.2 User-level Branch Vector Access

Once the program has been modified and analyzed, it is executed, as shown in Figure 2 Box 2. LBR monitoring begins when the test program enters its main method, and branch recording continues until the last instruction before the program ends. This prevents observation of the setup and teardown instructions executed as the program is loaded into and taken out of memory. Samples are taken based on the number of CPU Cycles observed during execution. When the sample rate of cycles is reached, the branches in the LBR are read and compared against the items in the branch table, and observed branches are marked as taken.

There are a number of ways to access branch vector data contained in the LBR. Many techniques in profiling, debugging, and other software tasks use some form of user-level performance monitoring API. Alternatively, a lower lever approach using interrupts can be used.

3.2.1 Access via Polling

The simplest technique to access and read the LBR is through a performance monitoring API and Linux's `poll` event. The test program is first spawned and executed using `ptrace`. Once the program has started execution successfully, LBR reading is enabled through a high level call to the operating system, as is the hardware counter that is to be used to trigger sampling. The monitoring program then repeatedly calls `poll`, which waits for the file descriptor associated with the performance counter to contain data that can be read, as shown below.

```
for(;;) {
        ret = poll(pollfds, 1, -1);
        if (ret < 0 && errno == EINTR)
                break;
        process_smpl_buf(file_descriptor);
}
```

While `poll` is an effective technique to report sets of LBR and performance counter data, repeatedly calling `poll` when no data is available causes unnecessary overhead. Thus, we created an alternative technique that takes advantage of interrupts.

3.2.2 Interrupt Driven Access

In our second technique, we replace the repetitious call to `poll` with a lower level, more efficient hardware access approach. The hardware counters and LBR are enabled in the same way as described in Section 3.2.1. The `poll` calls are replaced by an I/O signal handler associated with our desired hardware mechanisms. The signal handler is immediately triggered upon the associated performance counter's overflow. After performing several checks, the signal handler reads the LBR branch vector, and each branch is processed. The associated hardware counter then is reset and the program is resumed. By handling the performance counter notification and refreshing the counter directly from within the monitoring tool, we expect to significantly reduce the overheard associated with accessing and gathering data.

3.3 Branch Coverage

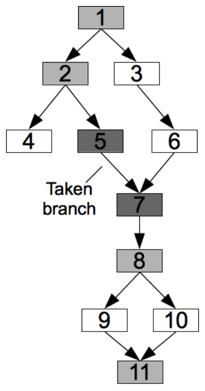

Figure 4: Dominator analyses based on an observed branch.

When accessing user-level branch vectors, sampling is used. Thus, it is possible that some executed branch data will not be recorded. To improve the branch coverage observed using a pure hardware approach to branch testing, monitored coverage details may be extended using compiler-based analyses, depicted in Figure 2 Box 3. These analyses can be performed offline or on a separate core during program execution, giving priority to the application being monitored. We first associate the branches observed by the LBR with branches in the control flow graph representation of the program. Dominator and post-dominator analyses are then executed on the control flow graph to build a dominator tree.

Within a dominator tree, a basic block b dominates basic block c if every path from the entry of the control flow graph to basic block c contains basic block b. A basic block b post-dominates basic block c if every path from c to the exit of the CFG contains basic block b. For example, Figure 4 shows a control flow graph of a function in which the LBR has observed branch 5-7. Because basic blocks 5 and 7 were executed, blocks 1 and 2 must also have executed based on the dominator analysis. Blocks 8 and 11 also necessarily executed based on the post-dominator analysis. Based on these two analyses, it is inferred that the conditional branches 1-2 and 2-5 must have executed, as well as the unconditional branch 7-8. Note that our branch testing technique only monitors conditional branches. However, when full branch vectors are observed, more branch vectors may be implied.

4. EMPIRICAL EVALUATION

The primary goal of this paper's empirical study is to evaluate the use of a hardware approach for structural testing for branch coverage calculation. We implemented THeME as described in Section 3 to measure its efficiency and effectiveness in comparison to using instrumentation. The goals of the experiments are as follows:

- Analyze the time overhead and code growth incurred by the program modification tool.
- Identify the differences between two methods of taking hardware samples in terms of efficiency.
- Analyze the trade-offs between efficiency and effectiveness of calculating coverage information using a hardware approach.
- Reveal benefits of testing using hardware mechanisms on multiple cores
- Demonstrate how static analysis can be used along with hardware mechanism monitoring for improved test coverage

4.1 Experiment Design and Metrics

We execute THeME on an Intel Core i7 860 / 2.8 GHz quad-core machine with 4GB of memory running Linux Kernel 2.6.34. The Intel Core i7 processor was selected because it has a LBR buffer that reports a branch vector of size 16, the largest currently available.

We used the SPEC2006 C Integer Benchmarks as test programs for our system. Each program was compiled with debugging information and with no optimization options specified. Debug information is included to link the executing binary instructions to the source code branch edges.

We analyze our system based on the efficiency and effectiveness of its branch coverage calculations. The efficiency of our infrastructure is calculated based on the base run times of benchmark execution reported by the execution tool of the SPEC2006 benchmarks, `runspec`. All timing results are compared to the overheads observed from execution of full software-instrumented versions of the benchmarks. TestCocoon [11] was used to generate the instrumented benchmarks.

The effectiveness of our infrastructure is analyzed by comparing the branch coverage observed using THeME to the coverage observed using full branch instrumentation. Coverage is calculated by dividing the number of branch edges observed using each technique by the total number of branch edges in the program.

4.2 Experiments and Results

We run four sets of experiments in order to analyze 1) the effects of our fall-through enabling program modification tool, 2) the efficiency and effectiveness of monitoring using hardware mechanisms on a single core, 3) the efficiency and effectiveness of monitoring using hardware mechanisms on a single core, and 4) the benefit of incorporating static analyses in terms of effectiveness.

4.2.1 Enabling Fall-through Visibility

The first experiment analyzes the effects of the program modification tool within THeME. We first examine the time overhead effects on the modified program compared to full instrumentation. Then we examine the code growth incurred. Table 1 lists the SPEC2006 benchmarks analyzed and the associated time overheads considered in this paper. The left side of the table shows the branch coverage as reported through instrumentation, the native program's execution time, the fall-through enabled (i.e. modified) program's execution time, and the fully instrumented program's execution time when executing on the SPEC2006 *test* input set. The right side of the table shows the same information when executing on the *ref* input set.

Our fall-through enabling modification tool generates programs that have on average only 5% time overhead compared

Table 1: SPEC 2006 benchmark time overhead information.

	test					ref			
Benchmark	Branch Cov.	Time(s)	Modified Time (s)	Instrumentation Time (s)		Branch Cov.	Time (s)	Modified Time (s)	Instrumentation Time (s)
bzip2	63.49%	16.5	16.9	18.6		64.20%	1499	1514	1599
h264ref	27.53%	43.8	43.8	47.7		35.72%	1753	1786	1890
libquantum	37.79%	0.155	0.16	0.165		39.07%	1056	1178	1236
mcf	73.70%	3.66	3.86	4.08		74.01%	529	539	575
sjeng	46.29%	6.92	7.74	8.96		48.87%	1028	1162	1312

Table 2: SPEC 2006 benchmark code growth information.

Benchmark	Native Size (kB)	Mod. % Increase	Instr. % Increase
bzip2	260 kB	1.52	32.65
h264ref	2892 kB	0.69	18.39
libquantum	208 kB	0	20.00
mcf	128 kB	0	17.95
sjeng	592 kB	0.67	30.05

to native execution. Adding full software-level instrumentation, on the other hand, introduces a 14% time overhead on average. In the case of *sjeng*, full instrumentation increases the time overhead by nearly 30% when executing both the *ref* and *test* inputs, whereas our tool adds only around 12%.

Table 2 shows the percent of code size increase of the modified and instrumented programs compared to native size. Our modifications are much more lightweight than traditional instrumentation probes and payloads because ours consist of only unconditional jumps and no payloads. In our benchmarks, full branch instrumentation results in code size increases ranging from 18% to 32%. Our tool, however, generates programs that are on average only 0.5% larger than the native code.

4.2.2 Testing on a Single Core

For the next set of experiments, we implement the two techniques described in Sections 3.2.1 and 3.2.2. There are a number of ways to access hardware mechanisms such as user-level APIs like OProfile [15], PAPI [4], and Perfmon2 [10]. However, none of these yet support LBR reading. We instead use a user-level tool, libpfm4, and its kernel-level interface, perfevents [10]. Because the current perfevents and libpfm4 APIs do not provide an interface to the LBR, we modified perfevents at the kernel level to include LBR support using a proposed patch [10]. We also patched libpfm to allow it to take advantage of the underlying kernel modifications. These APIs give us the ability to setup, teardown, and read hardware performance monitors and counters. We now analyze our two techniques for accessing the LBR based on efficiency, and then we examine the code coverage obtained by sampling the LBR at various rates.

Access via Polling- Efficiency: Figure 5 shows the time overhead of branch testing when accessing the LBR using the polling approach relative to full software-level instrumentation. The results for running on the `test` inputs of the SPEC benchmarks are displayed. As expected, the repeated calls to `poll` when no data is available causes unnecessary overhead. At sampling rates of 10 and 50 million, the polling approach improves time overhead slightly compared to the

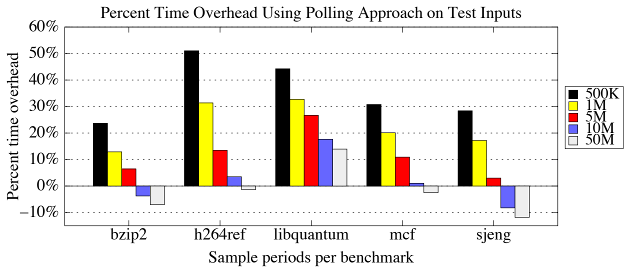

Figure 5: Time overhead for LBR sampling accessed using polling relative to full instrumentation on `test` inputs.

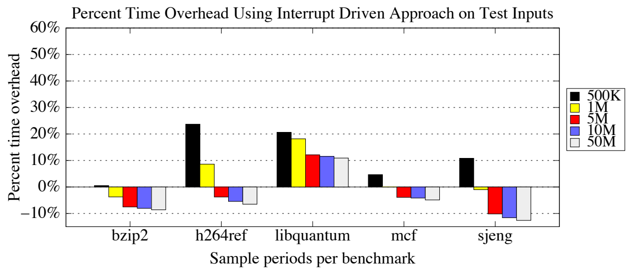

Figure 6: Time overhead for LBR sampling accessed using an interrupt-driven approach relative to full instrumentation on `test` inputs.

use of full instrumentation, performing with 12% less overhead than instrumentation in the case of `sjeng`. However, as sampling is performed more frequently, the cost due to repeatedly polling quickly rises. For example, sampling the LBR every 500K CPU cycles for `h264ref` results in 51% time overhead over instrumentation.

Interrupt Driven Access- Efficiency: Figure 6 shows the time overhead of branch testing when accessing the LBR using the interrupt-driven approach relative to full software-level instrumentation. Using the interrupt-driven approach for access substantially improves the time overhead of gathering branch vectors compared to the polling approach results depicted in Figure 5. At sample rates of five, ten, and fifty million, the time overhead of branch testing is improved over instrumentation for all benchmarks other than *libquantum*. This is because *libquantum* only executes for 0.155 seconds, as seen in Table 1, and its percent time overhead is greatly impacted by any amount of noise. *Sjeng*'s time overhead, however, can be reduced by 13% compared to instrumentation.

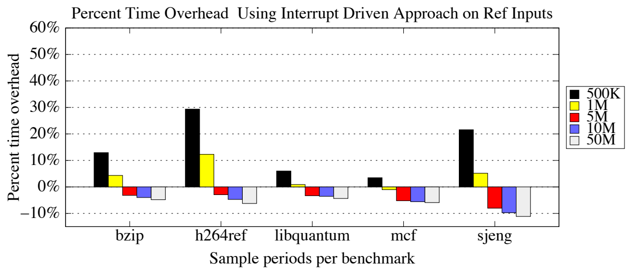

Figure 7: Time overhead for LBR sampling accessed using an interrupt-driven approach relative to full instrumentation on `ref` inputs.

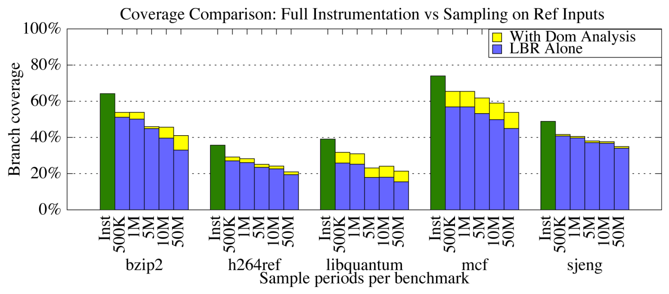

Figure 8: Coverage observed using LBR sampling via the interrupt-driven approach on `ref` inputs compared to instrumentation.

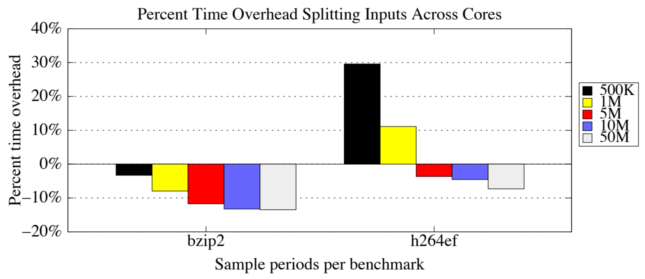

Figure 9: Time overhead for LBR sampling over multiple cores compared to using instrumentation on multiple cores.

The time overhead reported for sampling using the LBR does include the time overhead incurred by the program modification. Thus, using one of the alternative methods of detecting fall-through branches described in Section 3.1 would further improve the time overheads described here.

To better understand the effects of sampling the LBR on time overhead and coverage, we next evaluate the time overhead and branch coverage measured when reading the LBR every 500 thousand, 1 million, 5 million, 10 million, and 50 million CPU cycles while executing the *ref* inputs of the SPEC 2006 benchmarks. Each benchmark executes an average of 19.55 minutes, as shown in Table 1. The time overhead of executing larger programs with LBR sampling increases when sampling at smaller rates such as 500 thousand. This is potentially due to the operating system becoming overloaded with interrupts at lower sampling rates. At higher rates (e.g. 5 million, 10 million, 50 million), the time overhead incurred shown in Figure 7 is consistent with the time overhead when executing on the *test* inputs.

Effectiveness: On average, 76% of the coverage reported by instrumentation is observed when sampling the LBR every 500 thousand CPU cycles, as seen in Figure 8. The percent of coverage reported was nearly the same for each benchmark when using the polling or interrupt-based techniques. *Sjeng* achieves 82.61% of the coverage reported when monitoring by instrumentation, although the time overhead at that rate is 21.57% worse than instrumentation. However, at a sample rate of 50 million, *sjeng* still achieves 70.15% of the coverage reported using instrumentation while executing 12% faster than instrumentation. At a sample rate of 50 million, the average percent of coverage reported by instrumentation is reduced to 54%, but with a 6% improvement with regard to time.

4.2.3 Testing across Multiple Cores

We next observe the effect of monitoring test execution on multiple cores. Our multicore experiments focus on the two of the five SPEC2006 benchmarks tested in this research that include multiple inputs in the *ref* test set. Each input is executed on a separate core, and the coverage results were aggregated across cores as each test execution completed. The same sample rate was used on each core. Also note that the same procedure was followed when monitoring with hardware and with instrumentation. The reference input set of *bzip2* includes six inputs, and *h264ref* includes three. Because our experiments are run on a quad core machine, we executed only the first four inputs to *bzip2*.

As shown in Figure 9, the time overhead of monitoring the execution of the first four inputs of *bzip2* using the LBR was 4% to 14% less than than when using instrumentation. By removing instrumentation, the time overhead of executing test inputs on each core is improved, enabling greater time savings than when executing on a single core. As expected, the percent of actual coverage observed was the same as when executing on a single core, shown in Figure 8.

Unlike the overhead incurred by monitoring *bzip2*, the time overhead for *h264ref* using the LBR was greater than that of using instrumentation at sample rates of 500 thousand and one million. The timing results for *h264ref* are only slightly lower compared to sampling and executing on a single core. This is due to the fact that one of *h264ref*'s inputs executes for approximately 82% of the total execution time of the three inputs. Thus, the savings from executing the other two inputs on separate cores is not enough to substantially reduce the overall time overhead of monitoring *h264ref* using multiple cores versus a single core.

These experiments demonstrate that the time overhead of monitoring across multiple cores, relative to using instrumentation on multiple cores, incurs lower time overhead than when monitoring the LBR on a single core, relative to using instrumentation on a single core. Therefore, branch coverage analysis of multithreaded programs that execute on multiple cores will experience similar benefits to those of sequential or multithreaded programs executing on a single core. When the workload is evenly divided between multiple cores, we expect to observe time overhead results similar to those of *bzip2* in Figure 9.

4.2.4 *Improving Coverage at High Sample Rates*

In our final experiments, we demonstrate how the static analyses described in Section 3.3 can be used along with hardware mechanism monitoring for improved test coverage. Our dominator analyses were executed using the LLVM compiler infrastructure [13]. By incorporating these two analyses, the percent of actual coverage observed was improved from an average of 76% to 83% across all benchmarks at a sample rate of 500 thousand, as shown in Figure 8. At a rate of 50 million CPU cycles, the average percent of actual coverage is improved to 62.32% from 54% without dominator analyses. For *sjeng*, the dominator analysis improved coverage by only 1% on average across all rates. However, in *mcf*, the dominator analysis improved coverage by 9% on average. *Mcf* achieves 90% of instrumentation's test coverage with a sampling rate of 500 thousand and 72% with a sampling rate of 50 million. The results show that supplementing LBR samples with information from simple static analyses that are already performed by the compiler can potentially greatly improve coverage results depending on the program design.

5. HARDWARE MONITORING BENEFITS

From the experiment results, we see that the THeME system successfully enables a low overhead but effective branch testing technique for single and multithreaded programs. Used in conjunction with static analyses, LBR monitoring achieved up to 90% of the branch coverage observed using instrumentation with reduced time overhead and negligible memory overhead.

An approximation of coverage can be useful in a number of scenarios. One example is when testing on resource-constrained devices where the addition of instrumentation may be prohibitive in terms of time, memory, or power consumption. This technique provides a way to test on the devices themselves. Another scenario in which coverage loss is acceptable is in software tasks that only require a coverage estimate. For example, in regression test suite prioritization techniques, coverage is traditionally used to estimate fault-finding ability. THeME can be used to estimate fault-finding ability in an efficient way. Finally, while this work focuses on sampling at a constant rate throughout execution, the LBR can be turned on and off for different sections of a program based on the task at hand. Alternatively, the sampling rate can be tuned depending on what portion of the program is executing. For example, at points of concern, the rate of sampling can be increased to provide more accurate coverage estimations. Our system can additionally support applications such as data flow coverage, predicate coverage, and remote debugging.

Leveraging hardware mechanisms over using instrumentation has several additional advantages. The main advantage of leveraging hardware mechanisms rather than using instrumentation is with regard to the time overhead and lack of code growth. Both of these overheads can reduce the ability to test programs thoroughly, particularly on resource constrained devices such as tablets or mobile phones for which software generally is tested using emulators. Nearly all of these devices now include processors that have many accessible hardware counters and monitors. For example, the iPhone 3GS, Nokia n900, Samsung Galaxy Nexus, iPad2, Motorola XOOM, and the Amazon Kindle Fire all use ARM Cortex-A8 or A9 processors, which have more than fifty accessible hardware counters that can be utilized and are accessible at the kernel and user levels. These monitors include tracking of cpu cycles, branches retired, data read/writes, and more [10, 15]. Other processors used in related devices similarly have a wide array of accessible hardware counters that can be used to enable testing on the devices themselves.

Another key advantage of using hardware mechanisms over instrumentation has to do with what events can be monitored. The use of instrumentation is inflexible in that only that which is instrumented can be monitored. To improve understanding of program execution, more instrumentation must be added, further increasing the time overhead and code growth of monitoring. Instrumentation traditionally must be added into a program's source code or binary. Hardware mechanisms, however, can be used to monitor multiple events at both the user and kernel levels. Thus, instead of analyzing only a section of program execution, hardware mechanisms can also report events that occur outside the source such as in library calls and external routines, granting a much fuller picture of program execution with relation to interconnected executing programs/libraries and the system.

6. DISCUSSION

In this paper, we have described several methods used to improve the efficiency and effectiveness of our branch monitoring techniques. However, there are a number of promising opportunities to advance hardware performance monitoring technology at the hardware, kernel, and software levels that could further improve our access schemes.

6.1 Elimination of OS Shepherding

In current systems, the kernel is required to shepherd all functions related to configuring, accessing, and reading hardware mechanisms. Requiring the operating system's involvement in all of these functions comes at a cost that is significantly higher than is necessary. At the lowest level, completion of these operations requires either reading or writing registers on the processor core, which is highly efficient by nature. However, when each operation is performed via the operating system, there are a number of added sources of overhead. These include an added system call to enter the privileged kernel mode, the saving of context by spilling user-level state to memory, and restoring this state when execution is returned to the user-level application.

For many of the traditional applications that use hardware monitoring, this kernel-level usage model is sufficient. Generally, hardware counters and mechanisms are only set up and torn down at the beginning and end of an application's execution. Often, the monitored counter is left to increment until the end of the application's execution, at which point, it is read. Thus, kernel involvement is required only twice. In other monitoring techniques, hardware mechanisms can be accessed and read infrequently during program execution [18, 19]. However, because test coverage analysis requires more frequent monitoring, requiring OS involvement on each sample severely effects the time overhead of analysis. Allowing counters to be accessed directly from user mode, would result in a significant reduction in the cost of accessing the hardware mechanisms [26].

There are two ways to achieve user control of hardware mechanisms. One requires hardware modification, while the

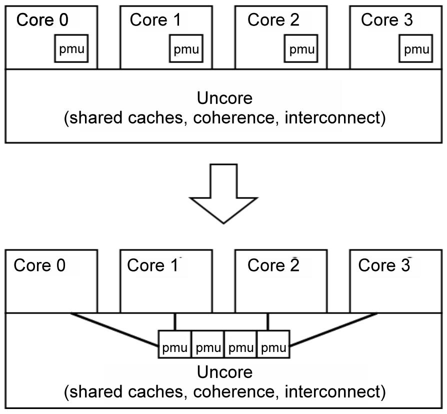

Figure 10: Moving private performance monitoring units to a global space to enable Satellite Monitoring.

other can be done using current hardware. The first approach is to allow the operating system to control the access permissions of hardware mechanisms directly by adding a simple register that can be used to specify execution modes that have direct access. While the overhead in the chip's die area to support this added permissions mode would be negligible, hardware modification would be required. The second approach is to have the kernel not set the mode of the processor back to user mode when execution returns to the application of interest. This technique would only require modifying the OS code, but it would result in a security hole that can be exploited by malicious programs.

6.2 Satellite Monitoring

Another opportunity to improve hardware monitoring efficiency is to enable what we call *satellite monitoring*. Currently, hardware monitoring information can only be collected from the core hosting the application being monitored. This necessitates that the program be interrupted to collect the needed information. However, allowing hardware monitoring information to be accessed from any core would require minimal hardware modification. It would require moving each core's performance monitoring unit (PMU), which controls hardware monitoring ability, into the "uncore."

Figure 10 illustrates moving each private PMU to the global uncore area. Making the aggregated PMU universally accessible would require an added core id assigned to each global PMU and added bus lines from each core. This approach would allow the monitoring and analysis of the application from a core separate from those hosting the application's threads, allowing the application to be unperturbed throughout execution. Using this approach would allow us to combine the advantages of multicore with those of performance monitoring technology.

6.3 Improving Effectiveness Through Instrumentation

The effectiveness of our techniques could also potentially be improved by inserting a small amount of software-level instrumentation into the application. Samples from hardware mechanisms are much more likely to observe instructions that occur along frequently executed paths. Instructions that are only hit occasionally, however, may not be seen.

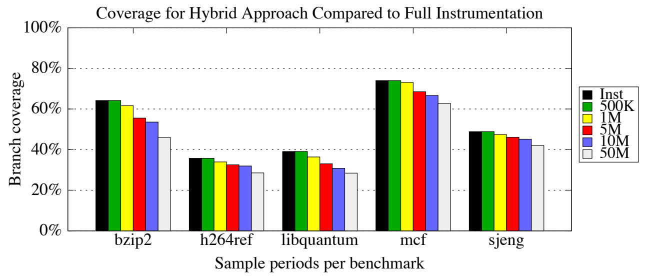

Figure 11: Percent of coverage observed when selectively instrumenting branches compared to instrumentation.

Ideally instrumentation would be added only to branches that are unlikely to be observed in LBR samples, and it would be inserted dynamically or prior to executing the test suite. However, identifying these locations is challenging. Without prior knowledge of program execution or profile information of the application, it is unclear where instrumentation should be inserted. If instrumentation is added unnecessarily, the overhead improvements from leveraging the LBR will be reduced. On the other hand, a conservative approach may have little impact on improving efficiency.

Figure 11 shows the coverage results of selectively instrumenting the benchmark applications when prior knowledge of execution, as reported by the LBR, is available. For each benchmark, instrumentation is added along any branch edge that was not observed by the LBR with a sampling rate of 500 thousand. Then the application is executed a second time to calculate the coverage obtained from both hardware and instrumentation. As is shown in Figure 11, at a sample rate of 500 thousand, nearly complete coverage information is observed in all cases. At a sample rate of 50 million CPU cycles, an average of 80% of instrumentation's test coverage is achieved.

These results are promising in terms of effectiveness. However, for a hybrid approach to be applicable for structural testing, we should assume that there is no prior analysis information available for the application and that the test suite only needs to be executed once to calculate coverage.

7. RELATED WORK

Much existing work exists that leverages hardware performance monitoring support for program counter sampling or event tracking is used for optimization, profiling, and debugging. To our knowledge, hardware performance monitoring has not previously been applied to branch testing techniques.

In our prior work [26], we discuss some of our preliminary efforts in exploiting hardware mechanisms for coverage monitoring, and a naive approach to LBR sampling is analyzed for efficiency. However, access techniques, the use of multiple cores, or effectiveness is not evaluated. In this paper, we explain the design of THeME and analyze the effects of accessing the LBR for coverage monitoring in terms of efficiency and effectiveness using different sampling rates over single/multiple cores. System improvements for effectiveness are also analyzed.

The work by Shye et al. [25] is most closely related to our research regarding using hardware mechanisms for monitoring. Their technique calculates basic block coverage, rather than branch edge coverage, using a combination of static analysis and Branch Trace Buffer (BTB) samples for the purposes of debugging. The BTB, available on the Itanium-2, is much like the LBR in that it is a circular buffer that stores the instruction and target addresses of branches executed. However, the BTB holds only four branches. In their work, after gathering all branch vector information, each vector is mapped to a partial path to calculate basic block coverage. Using this technique, they observe on average 47% of actual number of covered basic blocks using a sampling period of 100K with a performance overhead of approximately 25%. To improve coverage precision, they demonstrate the coverage increase when sampling is supplemented by a dominator analysis. Also, they perform aggregated runs, which is undesirable when testing, to try to gather more complete data, which is undesirable when testing.

The LBR is more commonly available on commodity machines and contains the last sixteen executed branches. This allows for more consecutive branch information to be observed. By using the LBR, we are able to gather more samples per period than if using the BTB, which enables us to achieve higher quality coverage data at lower sampling rates. We also implement more sophisticated sampling techniques and use multicore technology to improve the quality of our branch coverage approach.

Following the work of Shye et al. [25], Tran et al. [28] use specialized hardware to improve executed branch gathering. Using this hardware, they are able to achieve nearly 100% coverage with only 8% to 12% overhead. However, the hardware used is specialized, and the benchmarks are not standardized.

Hardware mechanisms have been successfully applied outside of testing and debugging for low overhead profiling of microarchitectural events. While hardware counters have been used in areas such as cache profiling [14], they have also proven useful for path profiling [1]. In work by Azimi et al. [3], a technique to use limited performance counters to simultaneously profile multiple events using sampling for performance analysis is introduced. Recent work by Ramasamy et al. [22] uses retired instruction events to dynamically calculate edge frequency estimates for profiling with a time overhead of less than 2% and no size increase. Mars and Hundt [18] and Chen et al. [7] use hardware performance monitors to aggressively tune dynamic optimizations. Yilmaz and Porter [29] also recently applied hardware mechanisms to distinguish failed executions from successful executions at a fraction of the runtime overhead cost of using software-based execution data. Finally, Sheng et al. [24] created a novel race detection tool that samples memory traces by sampling hardware mechanisms rather than using invasive instrumentation.

Sampling has also been used in software tasks, but without the use of hardware mechanisms, to improve efficiency. In work by Arnold and Ryder [2], instrumentation sampling is used to reduce the overhead of using complete sampling for profile collection. Their framework switches between instrumented and non-instrumented code by placing a sample condition on all method entries and backedges. A sample condition is checked, potentially causing the tool to execute fully instrumented code, based on a trigger mechanism. Using this combination of instrumented and non-instrumented code resulted in above 90% accurate profiles with 6% overhead.

Lightweight instrumentation combined with sampling of program executions has also been used for statistical bug isolation [16, 17]. Although these works do not focus on sampling techniques or applications of hardware, they demonstrate how instrumentation and sampling can be used together to produce highly accurate but low overhead results.

8. CONCLUSION

The work in this paper demonstrates that hardware mechanisms and multicore technology can be adapted for use in efficient and effective branch coverage analysis. We developed a runtime system that performs branch coverage analysis by monitoring hardware mechanisms on single and multiple cores. Monitoring program execution using hardware mechanisms was up to 11.13% faster in our tests compared to using instrumentation, but it does not provide complete coverage information. To improve coverage, we additionally perform a compiler analysis to extend the amount of coverage derived from each sample. The results show up to 90% of the actual code coverage can be determined with less time overhead and negligible code growth compared to using instrumentation.

Because these hardware approaches require only minor or no alterations to the program under test and incur low time overhead, they are ideal in resource constrained environments where testing generally cannot be performed without emulation. For this reason, they can also be applied to enable the testing of time-sensitive or multithreaded code.

In future work, we intend to combine hardware sampling with dynamic instrumentation to improve the effectiveness of our monitoring techniques. We also intend to implement THeME on resource constrained environments, such as tablets or smart phones. We are also interested in applying our hardware monitoring schemes to other coverage metrics for sequential and multithreaded programs.

9. REFERENCES

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32(5):85–96, 1997.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. *SIGPLAN Not.*, 36(5):168–179, 2001.

[3] R. Azimi, M. Stumm, and R. W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 101–110, New York, NY, USA, 2005. ACM.

[4] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 42, Washington, DC, USA, 2000. IEEE Computer Society.

[5] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–197, Washington, DC, USA, 2007. IEEE Computer Society.

[6] D. Chen, N. Vachharajani, R. Hundt, S.-w. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng. Taming hardware event samples for fdo compilation. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 42–52, New York, NY, USA, 2010. ACM.

[7] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 79–90, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. Testtube: a system for selective regression testing. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 211–220, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[9] T. Dey, W. Wang, J. Davidson, and M. L. Soffa. Characterizing multi-threaded applications based on shared-resource contention. In *To Appear: IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2011.

[10] S. Eranian. Perfmon2. `http://perfmon2.sourceforge.net`.

[11] T. S. Factory. Testcocoon - code coverage tool for c/c++ and c#. `http://www.testcocoon.org/`.

[12] Intel Corporation. *Intel 64 and IA-32 Architectures Software and Developer's Manual, Volumes 3A and 3B*. Intel Corporation, Santa Clara, CA, USA, March 2010.

[13] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[14] A. R. Lebeck and D. A. Wood. Cache profiling and the spec benchmarks: A case study. *Computer*, 27:15–26, 1994.

[15] J. Levon. Oprofile - a system profiler for linux. `http://oprofile.sourceforge.net`.

[16] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. *SIGPLAN Not.*, 38:141–154, May 2003.

[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. *SIGPLAN Not.*, 40:15–26, June 2005.

[18] J. Mars and R. Hundt. Scenario based optimization: A framework for statically enabling online optimizations. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 169–179, Washington, DC, USA, 2009. IEEE Computer Society.

[19] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention aware execution: online contention detection and response. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 257–265, New York, NY, USA, 2010. ACM.

[20] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 156–165, New York, NY, USA, 2005. ACM.

[21] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Commun. ACM*, 41:81–86, May 1998.

[22] V. Ramasamy, R. Hundt, W. Chen, and D. Chen. Feedback-directed optimizations with estimated edge profiles from hardware event sampling. In *Open64 Workshop at CGO 2008*, Boston, MA, USA, 2008. ACM.

[23] R. Santelices and M. J. Harrold. Efficiently monitoring data-flow test coverage. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 343–352, New York, NY, USA, 2007. ACM.

[24] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng. Racez: a lightweight and non-invasive race detection tool for production applications. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 401 –410, may 2011.

[25] A. Shye, M. Iyer, V. J. Reddi, and D. A. Connors. Code coverage testing using hardware performance monitoring support. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 159–163, New York, NY, USA, 2005. ACM.

[26] M. L. Soffa, K. Walcott, and J. Mars. Exploiting hardware advances for software testing and debugging (nier track). In *ICSE '11: Proceedings of the 33nd ACM/IEEE International Conference on Software Engineering*, New York, NY, USA, 2011. ACM.

[27] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 97–106, New York, NY, USA, 2002. ACM.

[28] A. Tran, M. Smith, and J. Miller. A hardware-assisted tool for fast, full code coverage analysis. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 321 –322, nov. 2008.

[29] C. Yilmaz and A. Porter. Combining hardware and software instrumentation to classify program executions. In *Proceedings of the 2010 Foundations of Software Engineering Conference*. ACM, 2010.