Exploiting Hardware Advances for Software Testing and Debugging (NIER Track)

Mary Lou Soffa
University of Virginia
soffa@cs.virginia.edu

Kristen R. Walcott
University of Virginia
walcott@cs.virginia.edu

Jason Mars
University of Virginia
jom5x@cs.virginia.edu

ABSTRACT

Despite the emerging ubiquity of hardware monitoring mechanisms and prior research work in other fields, the applicability and usefulness of hardware monitoring mechanisms have not been fully scrutinized for software engineering.

In this work, we identify several recently developed hardware mechanisms that lend themselves well to structural test coverage analysis and automated fault localization and explore their potential. We discuss key factors impacting the applicability of hardware monitoring mechanism for these software engineering tasks, present novel online analyses leveraging these mechanisms, and provide preliminary results demonstrating the promise of this emerging hardware.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitors, Testing tools*; D.3.4 [**Programming Languages**]: Processors—*run-time environments, optimization, debuggers*

General Terms

Performance, Measurement, Algorithms, Experimentation

Keywords

Branch testing, fault localization, performance monitoring

1. INTRODUCTION

Structural testing and fault localization are extremely important components of the software development process [9, 14]. In structural testing, the quality of a set of tests is measured based on the execution of structures in the source code. **Test coverage analysis** first involves determining what entities in the program need to be monitored based on the test coverage criteria desired. Then, as entities are monitored, they must be recorded. Once execution information is known, test coverage is calculated. **Fault localization** requires the identification of suspicious code that may contain bugs. This is often done by monitoring all execution leading up to a suspicious code segment or through the use of breakpoints in a debugger.

The overhead of test coverage analysis and automated fault localization 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 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 when applying such techniques are 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% [4, 8, 11]. When monitoring large scale programs or more complex structures for data-flow or paths, the overall cost of monitoring 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.

However, a new landscape of microprocessors is emerging that can change the way we think about efficiently performing online monitoring, and in fact, can potentially lead to completely new types of analyses as well as new hardware features. Hardware monitoring mechanisms have become ubiquitous on modern processors and are gradually becoming accessible at the kernel and user levels. For example, the Intel Nehalem processor provides the capability to track more than 2000 different performance events, and recent Linux kernel patches provide user-level support for nearly 200 of these mechanisms [3]. Compared to software driven approaches such as instrumentation, hardware mechanisms potentially can be used with very little overhead. The initial setup for a counter takes approximately 318μs, and reading a counter value takes only 3.5μs on average [2]. In addition to enabling monitoring with low time overhead, hardware mechanisms can remove the need for instrumentation.

Not only can these hardware mechanisms potentially be used to mitigate the costs that stem from online monitoring and analysis, advances in hardware monitoring technology provide an opportunity to develop new techniques for testing and fault localization. Although recent research has shown that the cost incurred for online monitoring and analysis for path profilers, trace selectors, and dynamic optimizers can be greatly reduced with hardware monitors [1, 6, 10], the potential of leveraging hardware mechanisms has been little researched for software testing and fault localization.

In this paper, we propose novel techniques that leverage hardware mechanisms to calculate structural coverage and perform fault localization. The goal is to determine if, as in profiling and optimization research, software testing and fault localization can benefit from the use of hardware mechanisms in terms of time overhead and code growth. First, we focus on determining which hardware mechanisms lend themselves well to these goals. Then we explore how event information can efficiently and effectively be obtained from the hardware mechanisms for low-cost monitoring. We propose novel online techniques that leverage hardware monitoring mechanisms to advance the state-of-the-art for structural testing and fault localization, and finally, we present preliminary results showing the promise of, and trade-offs between, key hardware performance mechanisms.

2. BACKGROUND

Although there are thousands of performance counters and hardware mechanisms that on many modern microprocessors can be accessed, this paper focuses on two more advanced hardware mechanisms that were recently introduced to enable debugging and precise event reporting. These mechanisms are the Last Branch Record (LBR) and the Branch Trace Store (BTS). Simple counters, such as Branch Instructions Retired, can only be used to report one instruction per sample. The LBR, however, reports the last n executed branches per sample where n is determined by the size of the LBR. For example, Intel's Core 2 processor reports the last 4 branches and the Core i7 processor reports the last 16. By increasing the amount of information available per sample period, the cost of monitoring a full program is decreased while improving the quality of information reported. In contrast, the BTS is an on-chip branch recording mechanism, which streams all executed branch information to a buffer stored in memory whose size and location is determined by the user. The BTS effectively enables the tracing of all branch events.

Current operating systems provide calls that allow us to harness hardware monitoring capabilities. Hardware performance counters provide a mechanism to trigger an OS interrupt when a particular value or condition is met. Although the hardware mechanism tracking a particular event will observe all events of that type during program execution, it is impractical to record every event that occurs due to the overhead of performing a system call. If these interrupts occur too frequently, the system can become overloaded with interrupts, slowing down execution and potentially even causing the system to appear frozen. Therefore, if the OS interface is used, a balance must be found between the amount of information collected and the overhead of collecting it.

3. CHALLENGES AND APPROACHES

The overall goal of this work is to identify potential recent hardware advances that can be leveraged to explore how these mechanisms can be used for more efficient, and indeed novel, analyses in software testing and fault localization. To do this, we will explore and develop techniques that use hardware mechanisms that are currently available on commodity machines and can be accessed at the kernel

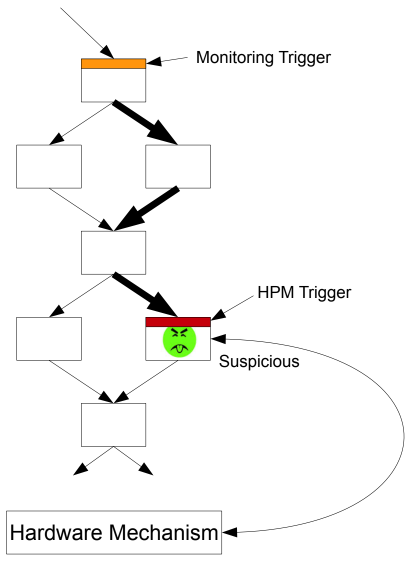

Figure 1: Deep Dynamic Inspection

and user levels. In this work, we discuss two such techniques, *efficient coverage analysis* for structural testing and a technique that we call *deep dynamic inspection* for fault localization.

3.1 Efficient Coverage Analysis

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 to report information on all executed events. To reduce the time overhead of monitoring, we need to selectively monitor hardware only during the test program's source code execution. Unlike profiling, in test coverage analysis, we are looking for the execution of particular events. Thus, sampling of the hardware needs to be performed more frequently and carefully to improve the likelihood of specific events being observed. For each mechanism used, we examine the interaction between sampling rate, coverage completeness, and the time overhead and code growth incurred.

To collect hardware monitoring information online for our efficient coverage analysis tool, we intend to use a very thin runtime layer, inspired by the lightweight introspection engine [6, 7]. This layer represents the minimal software presence necessary to manipulate core specific hardware monitoring features and either log the information for later analysis or transfer information to another core for parallel analysis. This software layer has a static code overhead of 21kb regardless of the size of the application.

3.2 Deep Dynamic Inspection

To illustrate how hardware monitoring mechanisms can be used to design novel online analyses for fault localization, Figure 1 presents our proposed *deep dynamic inspection* (DDI) technique. The goal of this approach is to provide an efficient lightweight technique for partially automated software fault localization. The programmer only needs to identify a region of suspicious code, and upon an error, our DDI analysis leverages hardware performance mechanisms,

Benchmark	Hardware % Growth	Instrumented % Growth
bzip2	1.54	50.77
gobmk	0.24	14.76
h264ref	0.73	23.37
hmmer	0.88	34.70
libquantum	0	25.00
mcf	0	21.88
sjeng	0	43.92

Table 1: Percent of code growth of hardware approach vs software instrumented approach.

namely the LBR, to report the path that leads to the error with negligible runtime overhead.

There are two classes of information reported by DDI: *concrete* path information and *fuzzy* path information. The concrete path that leads up to the error presents a complete path limited by the instantaneous size of the branch trace collected at the time of the error. Fuzzy path information provides longer probable paths using a reverse dynamic trigger injection technique. As shown in Figure 1, a hardware performance monitor (HPM) trigger is injected into the suspicious basic block. This trigger is tripped when the suspicious block is executed, and the contents of the LBR are immediately collected. The contents of the LBR at this point provide a concrete path leading up to the execution of this block. Upon this trigger, a new trigger can then be injected into the block at the head of this path. This process continues until an error occurs. If these paths are executed a number of times, longer fuzzy paths can also be provided.

4. PRELIMINARY WORK

Our preliminary work includes results which explore the potential of using the BTS and LBR in branch coverage analysis. In this work, we examine the tradeoffs between the sampling rate, time overhead, and code growth that can be obtained by sampling the LBR and BTS and applying the information to branch testing.

4.1 Code Growth

Independent of sampling technique, a leading source of low coverage monitoring effectiveness is due to the fact that branch-based hardware mechanisms alone cannot observe when fall-through branches have occurred. Fall-through branch observation is possible is several ways. One technique is to supplement the information from branch-based monitoring with other event data. Another technique to detect fall-through branches includes a static post mortem analysis of the program and observed information. However, because this paper focuses on the capabilities of using only branch-based hardware 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.

In Table 1, we compare the code size of the original program to 1) the program generated by applying our fall-through enabling tool and 2) a fully software-instrumented program. We use TestCocoon to generate the instrumented programs [4]. On average, our approach produces a binary with less than 2% code growth compared to the original. Note that alternative approaches to determine fall-through

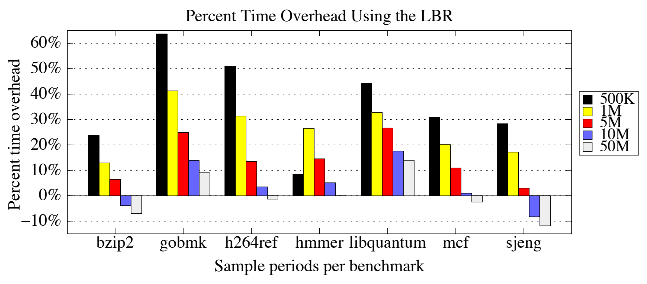

Figure 2: Actual time overhead for branch coverage analysis using the LBR through libpfm4 compared to instrumentation.

branches could be used that would incur no code growth. Our modifications are much more lightweight than traditional instrumentation, which incurred code size increase ranging from approximately 15% to 50% in the seven SPEC benchmarks that we considered. Most of these SPEC benchmarks are rather small in code size; we would expect more pronounced comparisons when using larger applications.

4.2 Branch Trace Store

Intuitively, the BTS is the most appealing hardware mechanism for use in monitoring branches for branch testing because it is capable of reporting a complete trace of executed taken branches. However, we discovered that using the BTS generated time overheads averaging 40X compared to native execution for our test applications. The extremely high time overhead is due to the fact that the BTS was designed as a debugging mechanism rather than a performance mechanism, and as it is reported in the Intel Processor Manual [5], when the BTS mechanism is turned on, the processor enters a specialized debug mode and runs 25X to 30X slower automatically (varies by application). This is further exacerbated by the fact that in the current Linux kernel, on every context switch, the BTS is disabled and reenabled, and the configuration is saved and restored in order to appropriately associate instruction pointers that are part of the branch records with the corresponding process. Thus, we find that the BTS overheads are inherently prohibitively high.

4.3 Last Branch Record

We also performed an exploratory study of the cost of performing branch coverage analysis when leveraging the LBR through a user-level hardware monitoring interface, libpfm4 [3]. To reduce the number of branches monitored that are not associated with the test program's source code, sampling begins when the binary has been loaded into memory and is stopped prior to cleanup. Figure 2 shows the effect of various sampling periods on the runtime collection of branch vectors. The time overhead is shown for five sampling periods ranging from 500K to 50M relative to the time overhead incurred by instrumentation using TestCocoon [4].

At higher sampling rates, we observe that the overhead of monitoring using the LBR is improved compared to instrumentation. However, to achieve more complete coverage information, smaller sampling periods are more desirable.

4.4 Potential Performance

Leveraging the LBR for the purposes of branch testing shows promise, especially for use in memory constrained

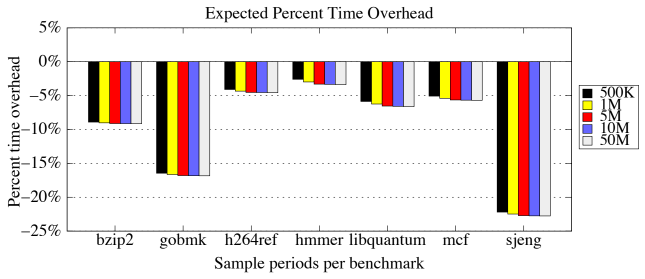

Figure 3: Expected time overhead of branch coverage analysis using lightweight approach compared to instrumentation.

systems where traditional instrumentation techniques cannot be applied. The reported time overhead of monitoring, however, can potentially be drastically reduced. Monitoring the LBR through libpfm4 during the entire test program execution leads to two undesirable results. The first is that many of the branches observed are not related to the source code with which we are concerned. Some of these branches relate to calls outside the test program. The majority, however, are due to pollution from the executing monitoring tool. The second is that the underlying cost to poll the LBR is unnecessarily high as the operating system is called.

To reduce these costs and improve coverage information, we propose to perform all monitoring at the kernel level using a lightweight tool like that described in Section 3.1. This tool will additionally turn off monitoring during execution outside the test program (i.e. during library calls or execution of outside packages). We estimate through experimentation that at the kernel level, the system calls needed to interrupt and read the LBR require only 1.2μs, while the polling function used in libpfm4 requires approximately 14600μs. Figure 3 represents the results we expect from applying these changes. Using this technique could thus reduce the time overhead of branch coverage analysis by more than 20% compared to full software-level instrumentation. This low-cost technique will also enable us to sample at smaller rates, increasing overall coverage observed.

5. RELATED WORK

The work by Shye et al. [12] is most closely related to our research regarding using hardware mechanisms for coverage analysis. Their technique calculates basic block 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, whereas the LBR in the Intel Nehalem processors contain the last sixteen, allowing 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. Tran et al. [13] use specialized hardware to improve executed branch gathering. Using this hardware, they are able to achieve achieve nearly 100% coverage with only 8% to 12% overhead. However, the hardware used is specialized, and the benchmarks are not standardized.

Yilmaz and Porter [15] 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.

6. CONCLUSION

This work explores the applicability and readiness of hardware monitoring mechanisms to common software engineering goals and presents a strong step forward to improve our understanding of the potential of leveraging recent features in structural testing and fault localization. Our preliminary work reveals that the memory needed for effective monitoring is significantly smaller than that of monitoring using instrumentation, making our techniques useful particularly in memory constrained environments. These techniques also can be used to calculate coverage with significantly less overhead than instrumentation. Additionally, we outlined a novel deep dynamic inspection analysis to enable a highly efficient fault localization technique.

7. REFERENCES

[1] 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*, pages 42–52, New York, NY, USA, 2010. ACM.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention aware execution: online contention detection and response. In *CGO '10: Proceedings of the 2010 International Symposium on Code Generation and Optimization*, pages 257–265, New York, NY, USA, 2010. ACM.

[8] 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.

[9] W. Perry. *Effective Methods for Software Testing*. John Wiley & Sons, Inc., New York, New York, 1995.

[10] 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.

[11] R. Santelices and M. J. Harrold. Efficiently monitoring data-flow test coverage. In *ASE '07*, pages 343–352, New York, NY, USA, 2007. ACM.

[12] 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.

[13] 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.

[14] T. Wang and A. Roychoudhury. Automated path generation for software fault localization. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 347–351, New York, NY, USA, 2005. ACM.

[15] C. Yilmaz and A. Porter. Combining hardware and software instrumentation to classify program executions. In *FSE '10*. ACM, 2010.