Exploiting Hardware Mechanisms and Multicore Technology in Software Testing

---

A Dissertation Proposal by

Kristen R. Walcott

9 November 2010

Submitted to the graduate faculty of the

Department of Computer Science

at the University of Virginia

School of Engineering and Applied Science

in partial fulfillment of the requirements

for the Dissertation Proposal and

subsequent Ph.D. in Computer Science

Approved By:

Westley Weimer, Committee Chair

Mary Lou Soffa, Advisor

Sudhanva Gurumurthi

Gregory M. Kapfhammer

Gregory J. Gerling

Outline

Abstract

Multicore technology and many hardware monitoring mechanisms have become ubiquitous in commodity machines in recent years. These hardware advances have been exploited for various software engineering tasks such as path profiling and dynamic code optimization, but they have not been leveraged in structural and data-flow testing.

The overhead of software testing is dominated by the costs of monitoring for events and analysis of the events. Monitoring is generally performed by adding code instrumentation, which can prohibitively increase the execution time and code size of the program. In branch testing, for example, the execution time on average increases 10%-30% while the code size grows 60%-90%. Analysis involves determining what entities in the program need to be monitored and calculating coverage. Although traditional static techniques exist to analyze code coverage, it is challenging to implement them such that they are scalable and precise.

This research will explore the potential of exploiting hardware advances to more efficiently perform structural and data-flow testing while maintaining a high level of effectiveness. First, our work will focus on determining which hardware mechanisms lend themselves well to testing. Then we will explore how event information can efficiently and effectively be obtained from the hardware mechanisms for low-cost monitoring. Finally, we will also investigate how multiple cores can be exploited to monitor and analyze event data.

Using hardware mechanisms introduces challenges in monitoring and in coverage analysis. One main challenge is determining how to gather the most complete event information possible while generating acceptable overhead. Hardware mechanisms are accessed through sampling. Sampling too frequently can lead to overloading the system, while infrequent sampling results in a sparse view of program execution. Thus, a balance must be found between the number of samples taken and the amount of event information reported.

Once execution events have been gathered, the analyses must take the sparsity of the data into account. Sparse data affects the completeness of structural testing. In data-flow testing, sparse data additionally can lead to incorrect assumptions regarding executed paths. Thus, additional analyses are necessary to improve coverage information and to protect correctness.

We will investigate leveraging hardware advances for the purposes of software testing in four phases. Because several hardware mechanisms lend themselves well to branch monitoring, we will begin our work with branch testing. In the first phase of our research, we will explore how multiple cores and a set of hardware mechanisms can be used for branch coverage. Data-flow testing is generally more effective than branch testing but is more expensive. Thus, in the second and third phases, we will design and implement approaches that monitor multiple hardware mechanisms for use in data-flow monitoring. The second phase will focus on data-flow testing of sequential programs, while the third will concentrate on multithreaded programs. Multiple cores will be used to aid in precise and scalable analyses of the sparse event data monitored. Finally, in the fourth phase, we will combine hardware monitoring with software-level instrumentation for comparison against pure hardware monitoring and software instrumentation approaches.

The approaches developed in these four phases will demonstrate the potential of using hardware mechanisms and multiple cores in testing. We will examine the tradeoffs between the efficiency and effectiveness of each of our techniques, and we will compare our results to the overheads and effectiveness of monitoring program events using traditional software-level instrumentation and analysis.

1 Introduction

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 have these features. Multicore processors can enable more efficient program analysis and modification, while hardware mechanisms enable architectural event monitoring at very low cost.

Hardware mechanisms and multicore architectures have been leveraged very successfully in software tasks including profilers and code optimizers [6, 11, 12, 13, 38, 47]. However, using hardware mechanisms and multiple cores has been explored very little within software testing. This research explores the potential of exploiting these architectural advances to more efficiently perform structural and data-flow testing while maintaining a high level of effectiveness. We investigate which hardware mechanisms lend themselves to different types of software testing and how execution information can be efficiently monitored using them. We also focus on how multiple cores can be used to aid in the tasks of monitoring and analysis.

Monitoring is generally performed using code instrumentation. To instrument code, the program is analyzed, either statically or dynamically, to determine points of interest. Each is marked by a probe, which is usually a jump or call to payload code. When the probe is reached during program execution, the payload is executed, and it analyzes the monitoring information needed. Usually the code inserted into the executable remains throughout execution, further increasing its expense unnecessarily. The time overhead of branch testing has been reported to be, on average, between 10% to 30%, with code growth ranging from 60% to 90% [20, 41, 48]. When monitoring large scale programs or more complex structures for data-flow or paths, the overall cost of monitoring can become prohibitive in both time and space.

As an alternative to software-based monitoring, in this research, we explore the potential of exploiting hardware mechanisms for monitoring in software testing. Modern-day processors include sets of hardware performance counters and monitors that were built to assist in tracking and measuring various aspects of program execution in situ. For example, the Intel Nehalem processor provides the capability to track more than 175 different kinds of execution events [19]. For generality, we will use the term hardware mechanism to include hardware performance counters, monitors, and hardware mechanisms in the rest of this proposal.

Compared to software profilers, hardware mechanisms can be used with very little overhead because they typically use in-CPU registers. The initial setup for a counter takes approximately 318μs, and reading a counter value only takes 3.5μs on average. Because hardware monitoring can remove the need for instrumentation, monitoring has the potential to be used with negligible code growth relative to the original program. Because the time overhead of acquiring a full trace of events using hardware mechanisms alone is likely to be prohibitive [27, 32], the proposed work will also explore the potential use of multiple cores for monitoring.

Efficient test coverage analysis is another critical component of software testing. Analysis first involves determining what entities in the program need to be monitored based on the type of test coverage desired. Then, as the entities are monitored, they must be recorded in some way. Once execution information is known, test coverage is calculated. When analyzing structural coverage such as branch or statement coverage, analysis is generally simple to perform, but more complex analysis is necessary for other forms of testing that are likely to produce greater fault detection rates [21, 26]. In data-flow testing, for example, the lifetime of data variables must also be analyzed. Our approaches must also take into account that full trace information likely will not be available from hardware monitoring alone. Thus, additional analysis techniques will be needed to more accurately determine partial paths based on sparsely observed events.

In this research, we propose that hardware mechanisms and multicore technology can be exploited in structural and data-flow testing of sequential and multithreaded programs for efficiency and effectiveness. The goal is to develop techniques that demonstrate the potential of using a variety of hardware mechanisms and multiple cores for more efficient test execution monitoring and analysis. There are four phases in this research. Because several hardware mechanisms lend themselves well to branch monitoring, we will begin our work with branch testing. In the first phase, we will develop techniques that monitor branch information using a variety of hardware mechanisms and multiple cores. Each technique will be compared to the overheads and effectiveness of monitoring program events using full software-level instrumentation. Then, because data-flow testing is more effective than branch testing, but is generally more expensive, the second phase of our proposed work will design and implement approaches that monitor and analyze memory loads and stores for use in data-flow testing of sequential programs. As executed branch information can also be useful for inferring data-flow information, our data-flow testing approach will monitor both memory and branch

instructions from hardware mechanisms. We will also explore how multiple cores can be leveraged to more efficiently analyze executed paths and inferred data-flow associations. The third phase of our work will extend our data-flow testing techniques to the more challenging problem of gathering and analyzing execution events to calculate data-flow coverage metrics for multithreaded programs. Finally, any software technique that gathers event information from hardware mechanisms is likely to miss some events, even when information is gathered frequently and precisely. Thus, in our fourth phase, we will develop a technique that is a hybrid approach to software testing, supplementing pure hardware monitoring with software-level instrumentation. In each of the four phases, all techniques will be evaluated in terms of efficiency and effectiveness compared to monitoring using full software-level instrumentation and performing analysis on a single core.

2 Background

Hardware mechanisms and multicore technology have been effectively used in a number of software tasks. However, there is little research on their application to software testing. In this section, we provide a brief background of a few hardware mechanisms available in commodity machines and a description of how multiple cores can be utilized for improved analyses. Also, we give a background of software testing as it applies to our work.

2.1 Architectural Advances in Commodity Machines

Recently multicore technology has become ubiquitous, and new chip architectures include performance-monitoring counter registers and performance monitors that enable the tracking of architectural events. In our proposed work, we develop techniques that exploit hardware advances for software testing. We first discuss how executed event information is gathered from hardware mechanisms. Then we describe three advanced mechanisms that can potentially improve the precision and overhead when gathering data. Finally, we discuss a satellite analysis technique that exploits multicore technology.

2.1.1 Sampling Hardware Mechanisms

Most microprocessors now support hardware event sampling through hardware mechanisms. When a designated hardware event has occurred a specified number of times, the Instruction Pointer and other register contents are recorded. This information identifies the instruction that caused the sample to be recorded.

Although a hardware mechanism tracking a particular event will observe all events of that type during program execution, it is not feasible to record every event that occurs. If too low a count reset value is set for a counter, the system can become overloaded with counter interrupts and appear as if the system has frozen. If this happens, it may be impossible to bring the system back to a workable state. There is no way to provide real security against this happening, other than making sure to use a reasonable value for the event counter reset. For example, setting monitoring `CPU_CLK_UNHALTED` events with an extremely low reset count (e.g. 500) is likely to freeze the system [32]. Therefore, a balance must be found between the amount of information collected and the overhead incurred by sampling.

2.1.2 Debugging and Advanced Hardware Mechanisms

Although information from hardware mechanisms must be gathered using sampling, some mechanisms are capable of gathering more information in fewer samples and of reporting more precise information. Beginning with the Intel P6 family of processors, breakpoints can be set on taken branches, interrupts, and exceptions, and single-stepping from one branch to the next is possible for the purposes of debugging and profiling. The Intel P6 also has the ability to log branch trace messages in memory. Such enhancements enabled the creation of the Last Branch Record (LBR) and Branch Trace Store (BTS) mechanisms. Processors based on the Intel Core microarchitecture also support Precise Event-Based Sampling (PEBS), in which an extra set of state information is stored for precise event monitoring.

Last Branch Record (LBR) The LBR was intended as a profiling tool for sampling partial branch paths in the operating system. The LBR branch vector of registers is available in many processors, and the number of branches that the LBR can hold is increasing with each new processor family [27]. When the LBR is turned on, the processor records a running trace of the most recent branches, interrupts, and exceptions taken by the processor. Each branch edge is stored as a source and destination address

The LBR can be sampled whenever an performance monitor interrupt (PMI) is generated. An interrupt occurs when a performance counter detects an overflow. At that point, the LBR branch vector can be polled

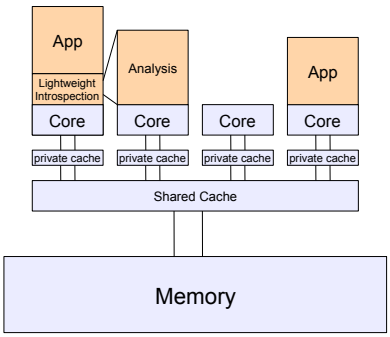

Figure 1: Satellite Analysis Framework

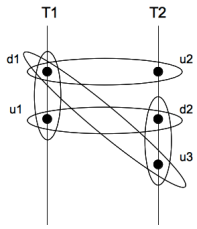

Figure 2: Parallel Def-Use

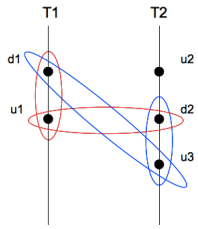

Figure 3: LR-Def

for branch data, and the information can be processed. The n branches polled from the LBR on each PMI define a *branch vector*, which represents a partial path of program execution through the branches.

Branch Trace Store (BTS) The BTS was developed as part of the processor's debugging facility and makes up half of the Debug Store (DS) mechanism [27]. The BTS works in conjunction with the LBR by requesting that the LBR send each branch record out on the system bus in addition to recording it. Simply turning on the BTS can greatly reduce the performance of the processor due to the effects of sending out and storing the branch record on *every* taken branch. The BTS mechanism provides the capability of saving the branch source and destination addresses in a memory-resident BTS buffer, and it can be configured to generate an interrupt when the buffer is nearly full so that all branch records can be observed.

Precise Event Based Sampling (PEBS) PEBS makes up the second half of the DS mechanism [27]. When a performance counter is configured for PEBS, a record is stored in the PEBS buffer after the counter overflow occurs. This record contains the machine state at retirement of the instruction that caused the counter overflow. When the state information has been logged, the counter is automatically reset to a preselected value, and event counting begins again. In other words, PEBS mode allows for accurate identification of the instruction address that follows the instruction that caused the sample to be taken. This mitigates issues caused by out of order execution and allows events in the instruction space to be accurately profiled.

On processors based on Intel Core microarchitecture, PEBS is supported for a subset of retired architectural events. These include the *instructions retired* (`INST_RETIRED`), *branch instructions retired* (`BRANCH_INSTRUCTIONS_RETIRED`), and *memory instructions retired* (`MEM_INST_RETIRED`). Note that we refer to regular sampling mode without PEBS as non-PEBS.

2.1.3 Satellite Analysis Using Multicore Technology

A multicore architecture consists of two or more independent processing cores integrated onto a single chip multiprocessor. Cores in a multicore chip may be coupled together tightly or loosely in that cores may or may not share caches, and they may implement message passing or shared memory inter-core communication methods. For example, the cores on the IntelCore i7 chip share an L3 cache and thus can communicate with low overhead. The Intel Pentium 4, Intel Core i7, and AMD's Phenom X4 multicore architectures have four cores per chip, while Sun's Niagra has 8 or 16 cores per chip.

One way to take advantage of having multiple cores during analysis is through a satellite analysis framework [38]. As an application thread is running on one core, online analysis can be performed on a separate processor core. Figure 1 shows an example configuration of a satellite analysis that can use the typical processor organization of today. As the figure shows, the runtime analysis executes on a separate core from the host application. A very thin layer, called the lightweight introspection engine, is shown lying under the host application, which represents the minimal software presence necessary to manipulate core specific performance monitoring features and transfer information to another core.

2.2 Structural and Data-flow Testing

In this proposed work, our techniques will leverage hardware mechanisms in conjunction with static analyses to calculate structural and data-flow coverage in sequential and multithreaded programs.

Structural testing is performed by monitoring a program's execution to determine what elements represented in its control-flow graph (CFG) or Parallel Execution Graph (PEG) are covered. Data-flow testing is a related control-flow testing technique which additionally examines the lifecycle of data variables. Coverage is defined as any metric of completeness with respect to a test selection criterion [7]. The most basic and most frequently used testing criteria involves monitoring program elements such as branches or statements. In branch testing, we aspire to test all branches in the program source code at least once. Statement coverage involves monitoring program execution for the coverage of all nodes or statements.

Testing techniques that cover data-flow relationships are generally more effective than those that only monitor structures such as branch or statements [21, 26], but they also are more expensive. One data-flow testing strategy is to cover all relationships between variable accesses. Definition-use association (DUA) testing uses instrumentation to record each definition `d` of a variable `v` during execution. For example, when a use `u` of `v` is executed, the last recorded `d` of `v` is stored, signifying that the path from `d` to `u` is covered for the variable `v`. The most effective data-flow criteria is path coverage, but its monitoring incurs a much higher cost than tracking definitions and uses of variables.

A number of multithreaded program test criteria have been proposed [9, 36], but those involving data-flow analysis have been little researched due to their expense in calculation. Two data-flow coverage criteria that have the potential to identify many bugs in multithreaded programs [21, 36, 53] involve partial interleaving monitoring based on definition-use pairs and consecutive access pairs. Instead of monitoring definition and use pairs in a single thread, they are tracked across all threads. For example, in Figure 2, there are five def-use pairs involving a single variable that should be covered for complete coverage: `(d1, u1)`, `(d1, u2)`, `(d1, u3)`, `(d2, u1)`, and `(d2, u3)`. To reduce the cost of full def-use monitoring, a more aggressive variation of the def-use model, LR-Def, has also been proposed in which every read access u where u both reads a value defined by the local thread and by a different thread is monitored[35]. For example, in Figure 3 the interleaving space gets complete coverage under LR-Def if the following pairs are covered: `(d1, u1)`, `(d1, u3)`, `(d2, u1)`, and `(d2, u3)`. Note that `u2` is not instrumented since it is only defined in `T2`.

3 Proposed Work

The overall goal of this research is to explore and develop techniques that exploit hardware mechanisms and multicore technology for structural and data-flow testing of sequential and multithreaded programs. Our research will focus on answering three main questions:

- What hardware mechanisms are most applicable for use in different areas of software testing?
- How can event information efficiently and effectively be obtained from the hardware mechanisms?
- How can multiple cores be exploited to more efficiently and effectively monitor and analyze event data to calculate coverage metrics?

We will demonstrate the potential of using hardware mechanisms and multiple cores in testing, and we will examine the tradeoffs between efficiency and effectiveness of our techniques when taking advantage of a variety of mechanisms. Each technique will be compared to the overheads and effectiveness of monitoring program events using software-level instrumentation. The specific steps of our research are as follows:

- Explore and develop techniques that monitor branch information using a variety of hardware mechanisms and multiple cores.
- Develop approaches that combine the use of multiple hardware mechanisms and multiple cores for use in data-flow coverage monitoring and analysis for sequential programs.
- Extend our techniques for data-flow testing of sequential programs to multithreaded program testing.
- Design, implement, and evaluate a tool that combines hardware monitoring with software-level instrumentation for comparison against pure hardware monitoring and software instrumentation approaches.

In this section, we first discuss these four phases of research that will allow us to achieve our goals. We then discuss the experiment design and empirical evaluation that will be used for each phase.

3.1 Exploiting Hardware Advances in Software Testing

Hardware mechanisms and multicore technology have been used successfully to monitor and analyze monitored data in many areas of software development including debugging, path profiling, and monitoring. However, the use of these hardware advances has been explored little in software testing.

3.1.1 Exploiting Hardware Advances for Branch Testing

The goal of the first phase of our work is to explore and develop techniques that will evaluate the potential of exploiting hardware mechanisms in branch testing. A number of hardware mechanisms lend themselves naturally to be used in branch testing. These include the BTS buffer, the LBR mechanism, and two performance events. The performance events are `INST_RETIRED` and `BRANCH_INSTRUCTIONS_RETIRED`, which report the instruction or branch instruction respectively that caused the sample rate counter to overflow. These can be sampled using PEBS or non-PEBS, where PEBS guarantees that the address reported for a counter overflow corresponds to a dynamic instruction that caused the counter to increment. In non-PEBS, the instruction address associated with an event by the PMU is often not the true address at which the event occurred. We will demonstrate the tradeoffs in time and memory overhead and effectiveness when exploiting each of these hardware mechanisms in branch testing.

Regardless of the hardware mechanism selected, taking advantage of hardware mechanisms in branch testing has several challenges. The main challenge is determining how to gather the most complete branch information possible while generating acceptable overhead. Obtaining a complete set of taken branches from hardware is extremely difficult because event information is gathered from hardware mechanisms through sampling. For each sample, an interrupt occurs and the operating system becomes involved. Thus, a balance must be found between the number of samples taken and the amount of event information reported.

The BTS can be sampled the least frequently because it stores all executed branches in a large buffer, which only needs to be sampled when full. However, the BTS was designed as a debugging mechanism and not for performance analysis, and its implementation involves writing to memory on every executed branch [27], regardless of sampling frequency. Although the BTS can be used to gather complete taken branch information, the cost of using it will likely be higher than desired. Still, to obtain full branch information, use of the BTS is likely to be less expensive than sampling every single branch.

Other hardware mechanisms can be used to gather smaller groups of event information per sample. Tracking the information in these mechanisms is less expensive than using the BTS, and the principle overhead comes from taking the sample itself. The LBR is a promising mechanism because each LBR sample is made up of a branch vector of the last n executed branches, where n depends on the architecture. `INST_RETIRED` and `BRANCH_INSTRUCTIONS_RETIRED` only report one executed branch or instruction per sample, but they are better supported at hardware and kernel levels, as described below. These two events can be monitored using PEBS or non-PEBS. Using PEBS is promising because it can incur less overhead than regular event sampling. This is because the CPU collects event samples on its own using a microcode routine and stores samples into a buffer supplied by the kernel module. An interrupt is generated and the OS is involved only when the buffer becomes full [47].

An additional challenge involves the amount of execution information that is observed by hardware mechanisms. In structural testing, we are concerned with gathering branch information only for branches in the source code. Hardware mechanisms, however, monitor ALL taken branches or instructions executed on the system. Because performance monitor interrupts are expensive to perform, the frequency of interrupts greatly impacts the time overhead of sampling branch data. Thus, sampling performance could be significantly improved through the use of filtering. On some processors such as those in the Intel Nehalem family, it is possible to filter the branches that are collected based on the branch type or privilege level, but kernel level support for filtering is not yet available for the LBR [27, 19]. Privilege level filtering is supported for the BTS buffer and performance events. Branch type filtering is only supported for the `BRANCH_INSTRUCTIONS_RETIRED` and `INST_RETIRED` performance events.

Another challenge of sampling from hardware mechanisms is the threat of sampling bias. One source of sampling bias is introduced by sample synchronization [12, 47]. If one selects a period that is synchronized with a part of the application, a few instructions will receive all of the samples. To reduce the risk of sample intervals synchronizing with program execution, randomization of the sampling period can be used. Hardware does not support randomization of every sample for BTS, LBR, or PEBS-based sampling. However, randomization support is available for non-PEBS sampling of our performance events. Another source of sampling bias is due to the time delay between the PMU counter overflow and the arming of the BTS, LBR, or PEBS hardware [31]. During this period, events cannot be detected due to the timing shadow. Additional sampling or analysis may need to be used to alleviate these sources of bias.

A final challenge arises from the fact that the hardware is monitoring the execution of binary code. Instrumentation monitors on a source code level and thus tracks both taken and fall-through sides of a

branch. Hardware that monitors branches, however, only can detect branches based on a jump from a source to some target. Thus, fall-through paths are not recorded by branch monitors. A supplementary technique is needed to account for both edges of a branch when sampling from the LBR, BTS, or BRANCH_INSTRUCTIONS_RETIRED event. Obviously, the INST_RETIRED event is capable of observing both taken and fall-through branches, although a separate analysis is needed to determine exactly which branch edges were executed based on the instructions observed.

Monitoring Branches We will first develop techniques in which branch testing is performed by exploiting each of the hardware mechanisms. Because we predict that BTS monitoring will produce the largest time overheads [27], our first technique will demonstrate the costs associated with accumulating a complete branch trace generated by sampling the BTS. We will then examine the effects of varying the sampling period of a traditional event-based sampling technique when using the LBR and non-precise performance events. Because branch testing is only concerned with whether a branch has been covered during execution but not in what order branch execution occurred, our techniques will begin with non-precise sampling of BRANCH_INSTRUCTIONS_RETIRED and INST_RETIRED. However, we will also examine the effects of using PEBS to sample because related work has observed that PEBS can incur less overhead [47].

We will apply filters to reduce the amount of extraneous data gathered by the hardware mechanisms. The BTS is filtered at the user privilege level, and the LBR is unfiltered. The BRANCH_INSTRUCTIONS_RETIRED event is filtered to monitor only conditional branches at the user level, and INST_RETIRED is filtered only to the user privilege level.

In addition to using filters, the techniques will also enable visibility of fall-through branches prior to sampling the branch-based monitors. Because three of the four hardware mechanisms being used are branch monitoring based, they can only report about 50% of the actual code coverage due to fall-through branch invisibility. To enable visibility, we will modify the program under test prior to execution to include innocuous unconditional branches along existing fall-through branches for use in all branch-based monitoring techniques.

After observing the efficiency and effectiveness of each monitoring technique using different sampling periods, we will then analyze the impact of sampling bias on our techniques' effectiveness by applying randomization to our techniques. Because randomization is only available on a per sample basis when sampling using BRANCH_INSTRUCTIONS_RETIRED and INST_RETIRED in non-PEBS mode, we will first observe the difference in branch coverage attained with and without randomization of these two events. We will also randomize the sample period between branch vector samples when monitoring the LBR.

Our final technique in this phase will take advantage of multiple cores to monitor branch data. We will take a simple approach in which the program is executed on each core and sampling is performed in a skewed way. In this scheme, the initial sampling rate is divided by the number of available cores. For example, with an initial polling period of 10,000 and 4 cores, the first core would monitor at 10,000, 20,000, and so on as normal. The second would take its first sample at 2,500 branches and proceed to sample every 10,000 branches after that. The third would being at 5,000 and the fourth at 7,5000. Using this approach, coverage precision similar to that achieved by the $\frac{\texttt{initial period}}{\texttt{\# of cores}}$ sampling rate can be achieved.

Branch Coverage Analysis Branch coverage analysis is simple to calculate when monitoring branches. Prior to execution, a table will be built of all branch edges occurring in the source code. If innocuous edges are added to the source code to enable branch-related hardware mechanisms to observe fall-through branch edges, these will be annotated in the table. As branches are observed using hardware sampling, they are checked off in the table. Final branch coverage is calculated by dividing the number of branches observed that are associated with the source code by the total number of branches in the table.

In techniques in which the INST_RETIRED event is sampled, our original table will group all instructions into basic blocks. As instructions are observed, they will be marked in the table as executed. Executed branch edges will be extrapolated in a post analysis, and branch coverage will be reported.

3.1.2 Exploiting Hardware Advances for Data-flow Testing of Sequential Programs

In the second phase of our proposed work, we will modify and extend our branch testing techniques that exploit hardware mechanisms to examine the applicability of hardware mechanisms in data-flow testing for sequential programs. Branch and statement coverage are simple and inexpensive to calculate from taken branch information, but covering data-flow relationships such as definition-use associations (DUAs) is often more effective [21, 26]. However, data-flow coverage metrics generally are more expensive to compute. In this

phase, we first demonstrate the potential of sampling memory instructions for data-flow testing of sequential programs. Then, to improve coverage, we will also incorporate branch/instruction information. For accurate and efficient coverage analysis using hardware-monitored event data, additional algorithms will be necessary.

One challenge in performing data-flow monitoring is that because data-flow testing examines the lifetime of data variables, partial path information must be tracked in addition to structural coverage. When performing branch testing as described in Section 3.1.1, exact observed branch ordering is unnecessary. However, for data-flow analysis, the instructions must be reported precisely. For example, when calculating def-use association coverage, there may be more than one coverage order for the nodes that constitute a *du* pair, including *kill* nodes. This occurs when two or more *du* nodes are mutually reachable or enclosed in a common cycle. No hardware mechanisms exist that will report a vector of memory instructions, like the LBR for branches, so our monitoring and analysis techniques must sample more thoroughly and extrapolate executed instructions based on those observed. Also, when monitoring taken data, execution ordering information must also be recorded.

Another challenge is that although PEBS supports precise sampling of instructions retired that contain both loads and stores, sampling memory instructions alone will likely result in a sparse picture of the lifetime of variables. While nearly complete memory instruction information can be obtained by reducing the sampling period to every instruction, this is infeasible due to the time overhead incurred. Therefore, additional monitoring or analysis is necessary to give a more complete view of program execution.

A third challenge lies in the data-flow analysis. Traditional data-flow analysis techniques are iterative by nature [2, 25]. These techniques are $O(n^2)$ in the worst case, where n is the number of nodes in the program's control flow graph. However, the traditional algorithms can be parallelized to more efficiently execute for large programs using multiple cores.

Data-flow analysis used in conjunction with hardware-monitored data is also challenging due to the sparsity of event information reported. For example, only half of a du-pair may be observed during hardware monitoring. To improve the completeness of our coverage analysis, we must include an approach to extrapolate other events that necessarily executed based on the observed coverage information.

Although a definition and an associated use may be observed by our monitoring techniques, our analyses must also take all associated kill nodes into account. For example, assume (d1, u1) is a valid du-pair in the program. An alternative path through the program may include a definition that kills d1. By monitoring using hardware mechanisms alone, this kill node may be unobserved. Thus, our data-flow analysis algorithms must additionally be able to determine events that may have executed in order to protect correctness.

Definition-Use Monitoring We will first develop a monitoring technique that precisely samples memory instructions for DUA testing. This technique will sample the `MEM_INST_RETIRED` performance event, which can be filtered to monitor instructions retired that contain a store or a load. The event will be monitored using PEBS because PEBS guarantees that the address reported for a counter overflow corresponds to a dynamic instruction that caused the counter to increment, preventing ordering issues that arise from out-of-order execution. We will use two different sampling schemes. First, we will sample using traditional event-based sampling rates, which are likely to produce an extremely sparse view of DUA behavior. In our second sampling scheme, we will use a bursty rate, in which, for small periods of time, all memory instructions are reported. The size of the bursty intervals will be determined through experimentation.

To improve the quality of coverage, we will next incorporate branch information gathered through additional event-based sampling. Santelices and Harrold [48] demonstrate that it is possible to infer the coverage of many DU-pairs based on branch coverage alone [48]. We will combine the sampling of branches, definitions, and uses to infer executed DUAs. Branch instructions will be sampled using techniques developed and tested in Section 3.1.1.

DUA Analysis for Sequential Programs An interprocedural flow graph (IFG) will be used to represent the program under test. An IFG is a collection of single-procedure control-flow graphs linked by interprocedural control-flow edges and is described by Harrold and Soffa [25]. Our approach first will leverage the multicore architecture by parallelizing the traditional techniques for calculating intraprocedural and interprocedural def-use associations [2, 25, 44]. From this, a DUA coverage matrix m will be formed, where each column corresponds to a use, and each cell in a column represents a definition for that use [48].

During program monitoring, we will track the last observed definition of each variable and mark in the matrix that it was executed. When a use is observed, it will also be marked in the matrix, and if a last

observed definition is stored for it, the pair will be marked as potentially covered. Branches will be marked as covered using a separate table. It would also be beneficial to efficiently record the order in which the samples are observed.

Our technique will next attempt to infer DUAs based on partial branch and def-use data. Our inference algorithm will be based on the technique described by Santelices and Harrold [48], which uses complete branch coverage information to infer DUAs. Although our monitoring technique will provide incomplete execution information, by also recording some information regarding the order in which samples are observed, we can potentially infer more DUAs.

We will first use a dominator analysis to determine definitions and uses that, based on program structure, must have executed. Since partial branch and def-use information is known, these can be used as starting points to guide the analysis. To refine our DUA coverage analysis, we will then use an infeasibility analysis as described by Bodik et al. [8] to remove def-use pairs that could not have executed. For example, assume that there are two du-pairs that share the same use, (d1, u1) and (d2, u1), where (d2, u1) lies along an infeasible path. Assume also that there exists a path from d2 to d1. Although (d2, u1) cannot ever be executed, our monitoring technique could potentially report d2 then u1 and miss d1. Thus, we would incorrectly identify (d2, u1) as the executed du-pair. However, if these are the only two partial paths that exist to reach u1, we can infer that (d1, u1) was executed.

Based on these two refining analyses, our algorithm will then calculate the set of DUAs that can be inferred based on the observed samples.

3.1.3 Exploiting Hardware Advances for Data-flow Testing of Multithreaded Programs

The third phase of our proposed work will extend our data-flow testing techniques for sequential programs, described in Section 3.1.2, to calculate data-flow coverage metrics for multithreaded programs . While many coverage metrics have been proposed for multithreaded programs, we will focus on the Def-Use and LR-Def coverage metrics, which have been proposed but have not been implemented or analyzed for multithreaded program testing [36].

The first challenge in monitoring and analyzing multithreaded programs involves determining a representation of the program that describes the interaction of the threads adequately but also has bounded size. The representation should also be statically built, based on a compile-time representation of the application.

A second challenge is that our techniques must account for inter-thread dependencies when analyzing shared memory multithreaded programs. Existing data-flow analysis techniques for concurrent programs typically either conservatively kill facts about all data that might be shared by multiple threads or uses a precise thread-interleaving analysis that determines which data may be shared, and thus which data-flow facts must be invalidated. The former approach can suffer from imprecision, whereas the latter may not scale. We must take into account that shared memory locations may be overwritten by concurrent threads.

Monitoring multithreaded programs is also challenging depending on the hardware mechanism and technique selected. PEBS is necessary when monitoring for data-flow testing, but if the program under test executes on multiple cores, each sample reported using PEBS will include multiple executed events. Without additional information, it will be unclear how each event in a set of samples interleaves.

As in analyzing hardware monitored information for sequential programs, we will again only be able to observe a sparse picture of the lifetime of variables for multithreaded programs. Therefore, additional analysis is necessary to give a more complete view of program execution and to protect correctness.

Definition-Use Monitoring To avoid the need for additional analysis or monitoring to determine interleaving behavior across cores, our monitoring technique will bind the program under test to a single core during execution. We use PEBS to sample memory instruction events across multiple threads to maintain precision of the events reported.

Def-Use and LR-Def Analysis We will represent multithreaded programs using a parallel execution graphs (PEGs). The PEG is a superstructure of a normal control ow graph with special edges and nodes incorporated to explicitly represent potential thread communication and synchronization [43]. PEG creation assumes that there exists a known upper bound on the number of instances of each thread class, and thus the actions of each thread can be uniquely represented in the graph. The nodes in PEGs are written as triples; a communication method such as a wait or notify is written as the triple (object, name, caller). The object is the entity that controls the communication, name is the method name, and caller is the caller thread name.

Nodes that do not represent communication methods have the symbol * in the object field. The CFGs for all threads in the program are combined in a PEG by adding four special edges between nodes from different CFGs. These edges represent thread starting edges, transitioning from a waiting state to a notified state, notifications between threads.

Data-flow analysis techniques for multithreaded programs have been adapted from sequential data-flow analysis techniques in recent years. However, these adaptations have major drawbacks. Krinke's approach [28] involves adapting the CFG to reflect explicit programmer annotated parallel functions and is limited in the memory models supported. Other approaches assume no shared variables [33] or only support restricted classes of programs or memory models [23, 49]. The most promising technique for a wide range of programs and based on a statically built program representation was designed by Chugh et al. [14]. They developed a conservative technique that generates a static non-null analysis and later uses data race detection to kill facts that parallelism no longer guarantees to be true.

We will first adapt traditional data-flow equations for Def-Use and LR-Def analysis in parallel programs based on the PEG representation [2]. We will then adapt the approach by Chugh et al. to use these equations with the PEG. As with sequential programs, we will also need to refine our observed event information based on infeasible interleavings and a dominator analysis to improve coverage results. To our knowledge, these algorithms have not yet been adapted for multithreaded programs.

3.1.4 A Hardware/Software Hybrid Approach to Software Testing

In the final phase of our proposed work, our goal is to develop a technique to improve the completeness of our testing approaches that monitor events using hardware mechanism sampling by incorporating software-level instrumentation. Sampling hardware mechanisms alone will produce incomplete coverage information because of imprecision and sampling bias, even when very small sample rates are used. Small sampling rates will also generate a much higher time overhead, possibly negating the advantages of harnessing hardware with regard to time. Although software-level instrumentation is an expensive form of monitoring, its cost can be reduced by placing instrumentation only along infrequently executed paths. Frequently executed events are also more likely to be observed in samples from hardware mechanisms. Therefore, this phase will examine how sampling of hardware performance monitors can be supplemented with software-level instrumentation to improve the efficiency and effectiveness of software testing.

The main challenge of supplementing hardware monitoring with software-based instrumentation is that it is unclear at what program points instrumentation should be added. To achieve complete coverage information, instrumentation can be added at events unobserved by hardware monitoring. In a two step execution process, this is simple, and execution of a partially instrumented program incurs little overhead. However, performing two separate runs is not suitable in a testing environment.

Monitoring with Hardware and Software To improve monitoring quality, we will selectively add instrumentation to the program under test as the program is running on one core, using a satellite analysis framework on another core. Instrumentation will be added along paths that are unlikely to be executed frequently based on already observed execution information. For example, when one branch edge is executed, we could instrument the alternate branch edge under the assumption that if the branch is executed again, if is likely to follow the same path. Frequently executed events are more likely to be observed using hardware monitoring, whereas events occurring along alternative paths are likely to be missed. One might also add instrumentation based on branch prediction information gathered from hardware.

In order to combine hardware and software monitoring, we will pause the executing program at points determined based on experimentation or observed behavior. Program execution state must be preserved to allow execution to resume following instrumentation. A satellite analysis framework on a separate core will instrument the program as desired and recompile it. The instrumented program will then start execution on the first core at the location where the original program was paused.

Analysis Coverage analysis will be performed using the techniques developed in Section 3.1.1, 3.1.2, or 3.1.3, depending on the coverage metric being analyzed.

3.2 Implementation and Evaluation

All experiments will be performed on a Intel Core i7 860 / 2.8 GHz quad-core machine with 4GB of memory running Linux Kernel 2.6.32. The Intel Core i7 processor was selected because it is part of the Nehalem family, which has a LBR buffer that contains the most recent 16 taken branches executed as well as BTS

and PEBS support. Filtering is supported at a hardware level on the Nehalem for the LBR, BTS, and the retired events supported by PEBS.

Although there are a number of APIs available for taking advantage of performance monitoring hardware including OProfile [32], PAPI [10], and Perfmon2 [19], none of these yet support LBR or BTS monitoring. Perfmon is supported by a user-level tool, libpfm, and a kernel-level interface, perfevents [19]. However, because Perfmon has not been updated to take advantage of libpfm4, the most recent helper library to perfevents, we will modify libpfm4 and perfevents directly to enable LBR and BTS support. The libpfm4 library helps encode performance events to use with the operating system kernel's performance monitoring interface, and it contains all of the Performance Monitoring Unit (PMU) model-specific information such as the events names and encodings, and the various constraints between events. It is one of the most robust and flexible PMU interfaces and supports a wide range of micro-architectures.

While the current perfevents and libpfm4 do not provide an interface to the LBR, we can modify perfevents at the kernel level to include LBR support using a proposed patch [19]. The LBR is accessed through a new `PERF_SAMPLE_BRANCH_STACK` sample type. This allows for sampling of all taken branches without any filtering capabilities. An addition kernel patch would be necessary in order to access the Nehalen's filtering abilities. Without filtering, the LBR records all branches, interrupts, and exceptions at both user and kernel level. No patch yet exists to enable filtering at the kernel level. We will patch libpfm4 to enable the setup and polling of the LBR. All sampling techniques will be implemented into the libpfm4 package. Libpfm4 and perfevents already support BTS and PEBS monitoring with filtering.

3.2.1 Metrics

To achieve our experiment goals, three metrics will be considered: the percentage of actual coverage, time overhead, and code growth.

In order to evaluate the effectiveness or completeness of our monitoring techniques, we first calculate the total number of the desired program structure that exists in the benchmark's source code, *total*. We then calculate how many of the structures were observed by the hardware and additional analyses, *hardware_total*. For comparison, we also will calculate the actual number of structures covered, *actual_total*, using Pin, a commercial software-based dynamic binary instrumentation tool [37]. The *hardware observed coverage*, $\frac{hardware\_total}{total}$ is compared to the actual coverage, $\frac{actual\_total}{total}$ to determine the *percent of actual coverage* that each hardware sampling technique and any additional analyses achieve.

The efficiency of the sampling techniques is calculated based on the base run times of benchmark execution reported by the execution tool of the SPEC benchmarks, `runspec`. All timing comparisons are made to the overheads observed from execution of full software-instrumented versions of the benchmarks. Full software instrumentation will be performed using TestCocoon [20] for branch instrumentation. Data-flow instrumentation will be performed by hand.

Code size measurements are taken using linux's `du` utility. The increase in the code size of modified programs is calculated by comparing the size of the original binaries to the size of the modified binaries and to the size of the software-instrumented binaries.

3.2.2 Benchmarks

Our experiments for single threaded programs will be carried out using the SPEC2006 C Integer Benchmarks. This set is made up of nine programs. Our experiments on multithreaded programs will use the SPECOMP2001 C benchmarks and the NAS Parallel Benchmarks in C. Each program will be compiled with debugging information and with no optimization options specified. Debugging information is only used once to map source level information such as line number and file name to the binary.

4 Preliminary Results

Our preliminary work includes results which indicate the viability of using the BTS and LBR in branch testing. In this work, we implemented a kernel module to explicitly control the Debug Save and BTS buffers to more efficiently gather taken branch information than the existing user-level BTS interface. We also examined tradeoffs between branch coverage precision and time and space overheads that can be obtained by sampling the LBR. LBR sampling inherently has several challenges described in Section 3.1.1, each of which we address using an assembly-level program modification tool, sampling randomization, and a simulation of LBR filtering. We also examined how LBR sampling across multiple cores can be used to increase branch coverage calculation precision.

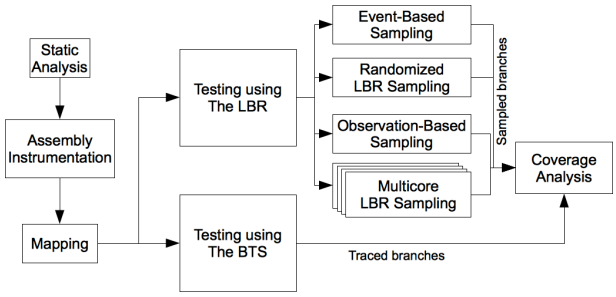

Figure 4: Overview of infrastructure to adapt BTS and LBR monitoring to branch testing.

An overview of our testing infrastructure is shown in Figure 4. Before executing and monitoring a program, a simple static analysis is first used to identify and store the branch edges in the program's source code. The branch edges are stored in a hash table along with information pertaining to the associated source code lines, obtained from gcc's debugging information. This table is used as a checklist of branches with which we are concerned and is later used to calculate overall branch coverage. Next, each of the benchmarks is instrumented to enable the LBR to monitor fall-through branch edges as well as taken branch edges. The program modification process is explained in Section 4.1.2.

In the next step, the program is executed while the LBR or BTS performs monitoring. The BTS is configured to generate an interrupt when the buffer is nearly full, thus building a complete branch trace as it is sampled. When the BTS buffer threshold overflows, all branches in the BTS buffer are processed. Similarly, when the branches executed counter overflows because it has reached the desired sampling period, the 16-branch branch vector in the LBR registers is read. Each branch is checked against the hash table of source code-level branch edges. If a branch is found in the table, the branch is marked as having been taken. Once the program under test has finished executing, the sampled branch coverage is calculated based on the number of source code-level branches observed divided by the total number of source code-level branches.

4.1 Experiments and Results

Experiments were run in order to analyze the time overhead and percent of actual coverage that can be achieved by sampling the LBR and BTS. We first evaluate the potential of applying the BTS to branch testing to evaluate its applicability. We then apply our program modification tool to enable the LBR to observe fall-through branches and analyze the time and memory overhead that it introduces. The modified programs are then monitored using three LBR sampling techniques to demonstrate the the trade-offs between efficiency and precision of code coverage calculation using traditional event-based sampling of the LBR. Finally, we create a simulation of the LBR hardware filtering mechanism to evaluate its potential success.

4.1.1 BTS Tracing

Because the BTS promises a certain time and space overhead just by turning it on, we start our experiments by tracing taken branches only using the BTS to gauge its potential. A buffer of size of 2 pages is used, and to minimize data loss, the threshold is set to overflow after 1 page. Because BTS filtering is supported at the user level, we constrain logged branch information to non-kernel level branches only. Branch type cannot be filtered, so unconditional jumps, calls, exceptions, and other branches will still be monitored.

A subset of our results are described in [54]. We discovered that using the BTS generated time overheads averaging 40X with the lowest being 2X and the highest being 90X compared to native execution. Increasing the buffer and threshold sizes had negligible effects on the time overhead of the branch tracing. The extremely high time overhead is due to the way the BTS mechanism is implemented in hardware. Specifically, the cost is an effect of the trace store occurring on every taken branch. On each 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. Simply turning the BTS on without sampling any of the branch data can account for 25 to 30X overhead [27], even when using filtering and not tracking fall-through branches. While these high time overheads may be acceptable for debugging, for the purposes of branch testing, the BTS overheads are inherently prohibitively high.

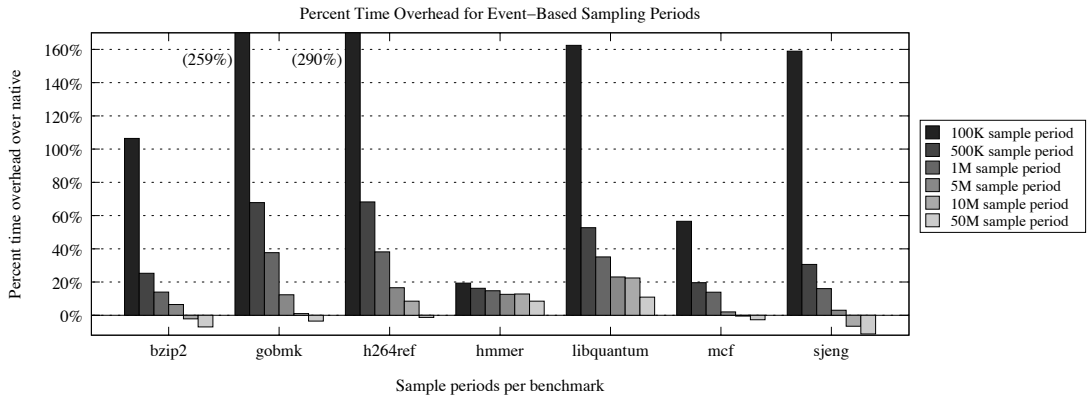

Figure 5: Time overhead for event-based sampling on a single core relative to full instrumentation.

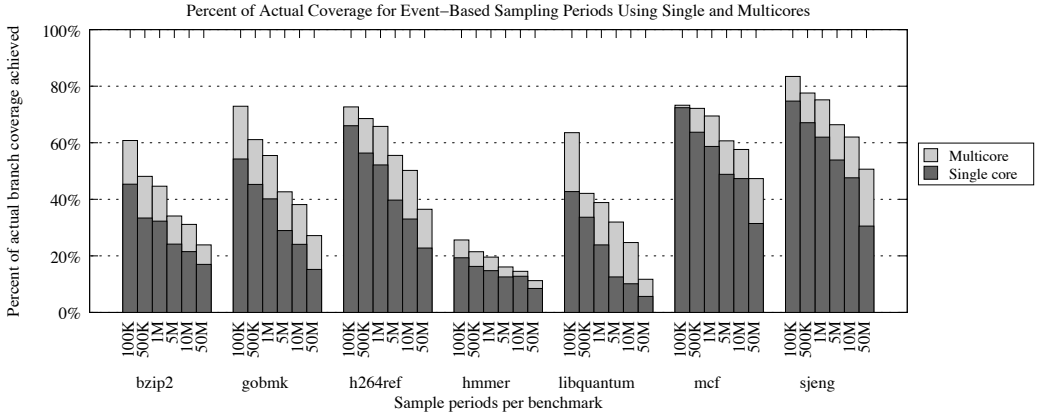

Figure 6: Percent of actual coverage from event-based sampling on single and multiple cores.

4.1.2 Program Modification Overhead

Before evaluating sampling techniques for the LBR, we first analyze the time overhead and code size increase caused by our program modifying tool. Our fall-through enabling modification tool is applied to all benchmarks prior to sampling.

To enable fall-through branch coverage monitoring through the LBR, our tool first compiled each benchmark down to assembly code. Each instruction in the program is examined, and if the instruction is a conditional branch, a `jmp 9f; 9:` is added immediately after it. This added instruction is an unconditional branch that can be seen by the LBR and simply jumps to the original fall-through instruction. Once these innocuous branches are added along each conditional branch fall-through path, the assembly code is compiled using gcc to generate new executables.

We compare the time overhead and code size of the original program to 1) the program generated by applying our modification tool and 2) a fully software-instrumented program. We use TestCocoon to generate the instrumented programs [20]. The time and code size comparisons are shown in [54]. On average, our modification tool generates a program with less than 5% time overhead and only 0.57% larger code size compared to native execution. Our modifications are much more lightweight than traditional instrumentation probes and payloads because ours consist of only unconditional jumps.

As this branch modification is the only contributor to increased memory overhead when sampling the LBR, we find that hardware monitoring techniques can be especially useful for testing in memory constrained environments. This is not the case for instrumentation. Full branch instrumentation of these 7 programs

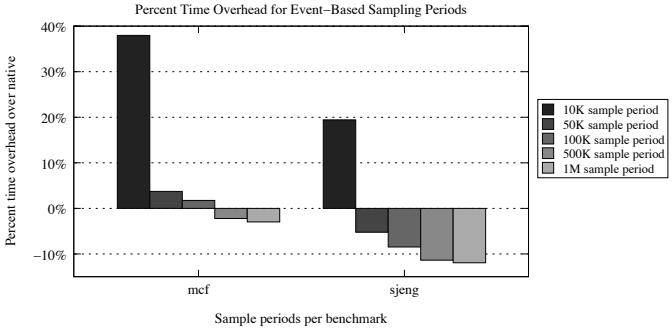

Figure 7: Time overhead relative to full instrumentation of a simulation of using a filtering mechanism.

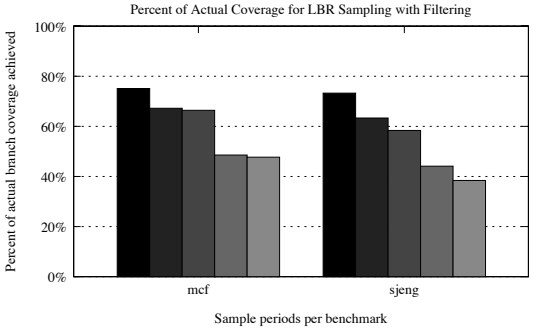

Figure 8: Percent of actual coverage obtained during a simulation of using a filtering mechanism.

using TestCocoon results in time overhead ranging from 4.5 to 30% and code size increase ranging from 15 to 51%. On average, these values are low for full branch instrumentation, as is observed in related work [41] and in the TestCocoon documentation [20], but even the reported code size overheads could cause instrumentation to be inapplicable in certain settings.

4.1.3 Event-Based Sampling

After a benchmark is modified to have observable fall-through paths, it is executed with the LBR turned on and configured to perform event-based sampling. Figures 5 and 6 show the effect of the sampling period on the runtime collection of branch vectors. The time overhead and percent of actual code coverage are shown for six sampling periods ranging from 1K to 5M relative to the time overhead incurred by performing full instrumentation using TestCocoon [20]. The interrupt thrown at the end of each sampling period is the main source of time overhead. As the sampling rate increases, the overhead also increases because more interrupts are thrown. Polling the LBR and processing the branches in the vector had negligible overhead.

As seen in Figure 6, the percent of actual coverage grows as the sampling rate increases, although at a much slower rate than the change in the time overhead. Despite that more samples are being processed, not all of these samples are associated with our source-code level branches, and of those that are, many have been seen and recorded before. The figures show that sampling with periods of 1 million results in an average of 24.2% time overhead, relative to instrumented code, with 40% of actual coverage for our benchmarks. However, some benchmarks perform much better. *Mcf*, for example, is able to achieve nearly 50% of the actual coverage for 2% overhead over testing using instrumentation.

At smaller sampling rates, the percent of actual coverage is much improved. For a sampling rate of 1000, for example, 84.8% of the percent of actual coverage is observed. However, the time overhead required at this rate is prohibitively high. Other sampling techniques are therefore necessary to reduce the frequency of interrupts in order to improve the potential of applying LBR sampling to branch testing.

4.1.4 Filtering Potential

Although filtering is not accessible at the user level yet for the LBR, we next analyze the potential efficiency and effectiveness improvements that can be obtained through its use. We develop a simulation of a filter that mimics abilities of the LBR filtering mechanism, capturing only application level branches associated with source-code level branches. We ensure that no other user processes are being executed during LBR monitoring and assume for the sake of the experiment that sampling bias does not occur.

To perform this simulation, we first generate a trace of all branch edges taken during program execution. We then sample the trace, as the LBR does, by collecting the last 16 branches executed on each sample period. This allows us to calculate the potential percentage of actual coverage that can be obtained. To estimate the time overhead of using such a filter in hardware, we calculate the number of samples m_p that would be taken for each sampling period p. We then execute the LBR on the benchmark using sampling period p, but stop the sampling mechanism once m_p samples have been taken. By using this technique, the actual times for LBR setup and teardown, sample processing, and performing the interrupts are incorporated.

Note that the coverage results may be slightly high because in our technique, we assume that all branches seen are associated with the source code. However, when filtering only application level branches, we will

still see branches that we are not concerned with such as branches from library and linking code.

The results of this experiment can be seen in Figures 7 and 8. By filtering, samples contain more information that is useful, allowing coverage rates to increase. Because fewer samples are needed to achieve a high percentage of actual coverage, fewer interrupts occur, drastically reducing the time overhead. This shows a filtering mechanism can substantially improve the applicability of the LBR for branch testing.

4.1.5 Randomization and Multicore Sampling

We also conducted experiments using randomization sampling and sampling over multiple cores using the LBR. Randomization had little effect, likely because filtering was not enabled due to the kernel's lack of filtering support. Shifting sampling across multiple cores increased the percentage of actual coverage over single core monitoring but had little impact on the time overhead compared to sampling on a single core at a smaller rate. Details are described in [54].

4.2 Discussion

From these experiment results, we see that the LBR shows much potential in enabling a low overhead but effective branch testing technique. With an average memory overhead of only 0.57%, LBR sampling can be applied in situations where memory is constrained, making code instrumentation infeasible. The time overhead can also be improved relative to using instrumentation when only an estimate of complete coverage information is needed.

Applying a filter to the LBR provides us with our most promising results regarding the potential of using the LBR for effective branch testing monitoring. Filtering greatly decreases the amount of samples that need to be gathered during the monitoring process and helps ensure the relevance of the samples taken for branch testing. Thus, both efficiency and effectiveness are improved when applying a filter.

Based on these results, we hypothesize that sampling of more simple hardware mechanisms, namely the `BRANCH_ INSTRUCTIONS_RETIRED` and `INST_RETIRED` events will improve the time overhead generated by our techniques. Also, use of Instructions Retired, will reduce the memory overhead of our program under test to 0 because no modifications to the original program are necessary for its use.

The design, implementation, and evaluation of our techniques have been submitted to ICSE 2011 [54].

5 Related Work

Hardware mechanisms and multicore technology have been applied to many software engineering tasks, although very little research has been done in the area of software testing. We next discuss the related work that has been done with regard to exploiting hardware advances in software engineering.

5.1 Hardware Mechanisms in Software Testing

The work by Shye et al. [51] is most closely related to our research regarding using hardware mechanisms for monitoring. 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.

Following up on the work by Shye et al. [51], Tran et al. [52] use specialized hardware to improve executed branch gathering. They use the `TRACE` unit in the Analog Devices family of embedded Blackfin processors [4] to increase the branch vector size to sixteen and to include a hardware-driven loop compression algorithm. By including these two enhancements, 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.

5.2 Hardware Mechanisms in Other Software Engineering Tasks

Outside of software testing, performance counters and monitors have shown to be a good tool in enabling low overhead profiling of micro-architectural events for use in many areas of software engineering. Many performance studies have been conducted using hardware mechanisms to judge of quality of software [46, 24]. Techniques have also been presented to use limited performance monitors to simultaneously profile numerous events for profiling [6]. Moreover, as hardware mechanisms are becoming more complex [17], the design and

support of hardware mechanisms has continued to advance. In recent years, this has been taken advantage of in software engineering research that relies on low-cost monitoring capabilities. These areas include path profiling, trace selection, and dynamic optimization.

5.2.1 Path Profiling and Trace Selection

One area in which hardware performance monitors and multicore processors have been leveraged is in path profiling. Some of the first research in this area was performed by Conte et al. [15]. In this work, traditional branch handling hardware was used to generate profile information in real time with an execution slowdown of only 0.4%-4.6%. Ammons et al. [3] later analyzed how to improve the precision of reporting hardware performance metrics such as instructions executed, cycles executed, instruction stalls, and cache misses to more effectively identify paths through a programs call graph. Other techniques have been designed to determine, through profiling, where program execution time is being spent [5]. Merten et al. [40] additionally explored using a branch behavior buffer to collect branch profile data for edge execution. Other research has used performance counters and monitors to predict phase and program path behavior based on observed events [18, 22, 29]. Profiling has also been performed for multithreaded programs using multicore systems [56].

Hardware mechanisms and multicore technology have also been used to form dynamic hot traces with low program overhead. Chen et al. [13] developed a technique to take advantage of the branch trace buffer of the Itanium to assist in trace selection. The Adore system [34] also proposes using the branch trace buffer to identify hot traces. Mars and Soffa [39] extended these techniques to exploit the multicore architecture to form higher quality traces without perturbing program execution.

5.2.2 Dynamic Optimization

Hardware mechanisms have most commonly been exploited for feedback-directed optimizations [45, 47, 11, 38, 50, 1, 34, 55, 42, 30, 16, 12]. Cavazos et al. [11] used hardware mechanisms and machine learning together to find better compiler optimization settings for applications. In the work by Mars and Hundt [38], functions are statically multiversioned for different dynamic scenarios that can be identified by monitoring hardware events. Recent work by Ramasamy et al. [47] and Chen et al. [12] uses retired instruction events to dynamically calculate edge frequency estimates for feedback-directed compilation. We also see hardware mechanisms being used in Java virtual machines and just-in-time compilers to steer optimization [50, 1, 16].

6 Conclusion and Merits

This proposed work represents the first effort to exploit hardware mechanisms in branch and data-flow testing for sequential and multithreaded programs. Hardware mechanisms and multicore architectures recently have been leveraged in other software engineering research for tasks such as profiling and optimization. We propose that software testing can similarly benefit from their use. In this work, we design and analyze of a set of techniques that will improve our understanding of the potential of leveraging architectural features to produce better software testing tools. In addition to exploring how five different hardware mechanisms can be exploited in software testing, we also combine sampling from multiple mechanisms to gain a more complete view of program execution, evaluate the tradeoffs between using different sampling techniques, and explore how multiple cores can aid in both monitoring and analysis.

Our preliminary work demonstrates that hardware mechanisms can be used successfully to significantly reduce the amount of code growth in branch testing, while also reducing the time overhead. Our remaining work will continue to examine how multiple cores and a variety of hardware mechanisms that are available on commodity machines can be exploited. These architectural advances can improve both monitoring and analysis abilities in terms of overhead. While the majority of our proposed work focuses on removing the need for software-level instrumentation, we will also examine how instrumentation can be used alongside hardware monitoring to attain a higher level of efficiency and effectiveness than available testing techniques.

Bibliography

[1] Ali-Reza Adl-Tabatabai, Richard L. Hudson, Mauricio J. Serrano, and Sreenivas Subramoney. Prefetch injection based on hardware monitoring and object metadata. *SIGPLAN Not.*, 39:267–276, June 2004.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley Publishing Company, USA, 1986.

[3] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32(5):85–96, 1997.

[4] Analog Devices. *ADSP-BF5333 Blackfin Processor Hardware Reference Manual*. July 2006.

[5] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? *SIGOPS Oper. Syst. Rev.*, 31:1–14, October 1997.

[6] Reza Azimi, Michael Stumm, and Robert 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.

[7] B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, New York, 1990.

[8] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Refining data flow information using infeasible paths. *SIGSOFT Softw. Eng. Notes*, 22(6):361–377, 1997.

[9] Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. Applications of synchronization coverage. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 206–212, New York, NY, USA, 2005. ACM.

[10] 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.

[11] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier 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.

[12] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin 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.

[13] Howard Chen, Wei-Chung Hsu, Jiwei Lu, Pen-Chung Yew, and Dong-Yuan 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.

[14] Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner. Dataflow analysis for concurrent programs using datarace detection. *SIGPLAN Not.*, 43:316–326, June 2008.

[15] Thomas M. Conte, Burzin A. Patel, Kishore N. Menezes, and J. Stan Cox. Hardware-based profiling: an effective technique for profile-driven optimization. *Int. J. Parallel Program.*, 24:187–206, April 1996.

[16] John Cuthbertson, Sandhya Viswanathan, Konstantin Bobrovsky, Alexander Astapchuk, and Eric Kaczmarek. Uma Srinivasan. A practical approach to hardware performance monitoring based dynamic optimizations in a production jvm. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 190–199, Washington, DC, USA, 2009. IEEE Computer Society.

[17] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. Profileme: hardware support for instruction-level profiling on out-of-order processors. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 292–302, Washington, DC, USA, 1997. IEEE Computer Society.

[18] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 220–, Washington, DC, USA, 2003. IEEE Computer Society.

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

[20] TestCocoon Software Factory. Testcocoon - code coverage tool for c/c++ and c#. `http://www.testcocoon.org/`.

[21] P.G. Frankl and E.J. Weyuker. An applicable family of data flow testing criteria. *Software Engineering, IEEE Transactions on*, 14(10):1483–1498, Oct 1988.

[22] Andy Georges, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Method-level phase behavior in java workloads. *SIGPLAN Not.*, 39:270–287, October 2004.

[23] Dirk Grunwald and Harini Srinivasan. Data flow equations for explicitly parallel programs. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 159–168, New York, NY, USA, 1993. ACM.

[24] Swathi Tanjore Gurumani and Aleksandar Milenkovic. Execution characteristics of spec cpu2000 benchmarks: Intel c++ vs. microsoft vc++. In *Proceedings of the 42nd annual Southeast regional conference*, ACM-SE 42, pages 261–266, New York, NY, USA, 2004. ACM.

[25] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Trans. Program. Lang. Syst.*, 16(2):175–204, 1994.

[26] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[27] 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.

[28] Jens Krinke. Static slicing of threaded programs. *SIGPLAN Not.*, 33:35–42, July 1998.

[29] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, pages 236–247, Washington, DC, USA, 2005. IEEE Computer Society.

[30] Roy Levin, Ilan Newman, and Gadi Haber. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, HiPEAC'08, pages 291–304, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] David Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors.* Intel Corporation, Santa Clara, CA, USA, 2009.

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

[33] Douglas Long and Lori A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of the symposium on Testing, analysis, and verification*, TAV4, pages 21–35, New York, NY, USA, 1991. ACM.

[34] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei chung Hsu. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, 6:2004, 2004.

[35] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. A study of interleaving coverage criteria. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 533–536, New York, NY, USA, 2007. ACM.

[36] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. *SIGARCH Comput. Archit. News*, 34(5):37–48, 2006.

[37] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[38] Jason Mars and Robert 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.

[39] Jason Mars and Mary Lou Soffa. Mats: Multicore adaptive trace selection. In *2008 Workshop on Software Tools for Multicore Systems (STMCS)*, Boston, MA, USA, March 2008.

[40] Matthew C. Merten, Andrew R. Trick, Erik M. Nystrom, Ronald D. Barnes, and Wen-mei W. Hmu. A hardware mechanism for dynamic extraction and relayout of program hot spots. *SIGARCH Comput. Archit. News*, 28:59–70, May 2000.

[41] Jonathan Misurda, James A. Clause, Juliya L. Reed, Bruce R. Childers, and Mary Lou 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.

[42] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Matthew Iyer, Dan Fay, David Hodgdon, Joshua L. Kihm, Alex Settle, Dirk Grunwald, and Daniel A. Connors. Dynamic run-time architecture techniques for enabling continuous optimization. In *Proceedings of the 2nd conference on Computing frontiers*, CF '05, pages 211–220, New York, NY, USA, 2005. ACM.

[43] G. Naumovich, G.S. Avrunin, and L.A. Clarke. Data flow analysis for checking properties of concurrent java programs. In *Proc. International Conference on Software Engineering*, pages 399–410, 1999.

[44] H.D. Pande, W.A. Landi, and B.G. Ryder. Interprocedural def-use associations for c systems with single level pointers. *Software Engineering, IEEE Transactions on*, 20(5):385 –403, may. 1994.

[45] Nitzan Peleg and Bilha Mendelson. Detecting change in program behavior for adaptive optimization. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 150–162, Washington, DC, USA, 2007. IEEE Computer Society.

[46] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. *SIGARCH Computer Architecture News*, 35(2):412–423, 2007.

[47] Vinodha Ramasamy, Robert Hundt, Wenguang Chen, and Dehao Chen. Feedback-directed optimizations with estimated edge profiles from hardware event sampling. In *Open64 Workshop at CGO08*, Boston, MA, USA, 2008. ACM.

[48] Raul Santelices and Mary Jean 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.

[49] Vivek Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '97, pages 94–113, London, UK, 1998. Springer-Verlag.

[50] Florian T. Schneider, Mathias Payer, and Thomas R. Gross. Online optimizations driven by hardware performance monitoring. *SIGPLAN Not.*, 42:373–382, June 2007.

[51] Alex Shye, Matthew Iyer, Vijay Janapa Reddi, and Daniel 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.

[52] 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. STUDENT PAPER-2 pages.

[53] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 334–345, New York, NY, USA, 2006. ACM.

[54] Kristen Walcott and Mary Lou Soffa. Exploiting hardware performance mechanisms for branch testing. In *Submitted to International Conference on Software Engineering (ICSE)*, Waikiki, Honolulu, Hawaii, USA, 2011. ACM.

[55] Weifeng Zhang, Brad Calder, and Dean M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 87–98, Washington, DC, USA, 2005. IEEE Computer Society.

[56] Qin Zhao, Ioana Cutcutache, and Weng-Fai Wong. Pipa: pipelined profiling and analysis on multi-core systems. In *CGO '08: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 185–194, New York, NY, USA, 2008. ACM.