Code Coverage Testing Using Hardware Performance Monitoring Support

Alex Shye Matthew Iyer Vijay Janapa Reddi Daniel A. Connors

Department of Electrical and
Computer Engineering
University of Colorado at Boulder

{shye, iyer, janapare, dconnors}@colorado.edu

ABSTRACT

Code coverage analysis, the process of finding code exercised by a particular set of test inputs, is an important component of software development and verification. Most traditional methods of implementing code coverage analysis tools are based on program instrumentation. These methods typically incur high overhead due to the insertion and execution of instrumentation code, and are not deployable in many software environments. Hardware-based sampling techniques attempt to lower overhead by leveraging existing Hardware Performance Monitoring (HPM) support for program counter (PC) sampling. While PC-sampling incurs lower levels of overhead, it does not provide complete coverage information. This paper extends the HPM approach in two ways. First, it utilizes the sampling of branch vectors which are supported on modern processors. Second, compiler analysis is performed on branch vectors to extend the amount of code coverage information derived from each sample. This paper shows that although HPM is generally used to guide performance improvement efforts, there is substantial promise in leveraging the HPM information for code debugging and verification. The combination of sampled branch vectors and compiler analysis can be used to attain upwards of 80% of the actual code coverage.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification

Keywords

Code coverage, software testing, hardware performance monitoring

AADEBUG'05, September 19–21, 2005, Monterey, California, USA.

1. INTRODUCTION

The design of software is an increasingly complex process involving issues of compatibility, conformance, functionality, time-to-market, and performance. Moreover, the size of software systems is rapidly growing and thus the potential for errors is multiplied [15]. As such, an essential step in the software development process is aggressive testing. One critical characteristic of evaluating the testing process is determining the amount of code executed by an individual test or a set of tests. Code coverage is the observation that specific code points execute during evaluation and is a common metric of testing [12, 18].

The traditional method of performing code coverage analysis [4, 9] is by using program instrumentation. The instrumentation process inserts software probes into the target application to track executed program locations. The tracked program locations have some assurance of correct operation. While instrumentation generates precise code coverage information, there is the disadvantage of significant collection overhead (between 50% and 200% [17]). Nevertheless, the generation of high-quality tests in a timely fashion is an important aspect of software development, and results in shorter development time.

Another class of code coverage system utilizes existing Hardware Performance Monitoring (HPM) support for sampling program counter (PC) addresses. The sampled PCs can be used to determine partial aspects of code coverage [3, 14]. Although these systems decrease overhead considerably, they are only able to gather a fraction of the information that full instrumentation allows.

This paper examines the potential use of a HPM support in modern processors for code coverage analysis. Modern processor support for HPM [8, 10, 16] provide facilities for sampling *branch vectors*, a set of correlated branch events representing a path of program execution. Branch vectors naturally provide more information than a single program point. Furthermore, when HPM information like branch vectors are integrated with compiler infrastructure, the inherent value of the information is dramatically extended through the use of standard analysis techniques. For example, by mapping the HPM information to the compiler's control flow graph (CFG) representation of the program, dominator analysis [2] can guarantee the execution of additional code blocks. An experimental system of these concepts is evaluated using the Itanium-2 PMU in the OpenIMPACT [13] compiler. Initial results indicate that although

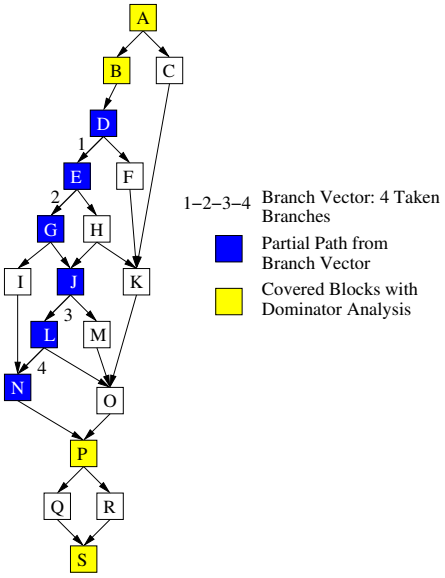

Figure 1: HPM partial path creation and path extension based on dominator analysis.

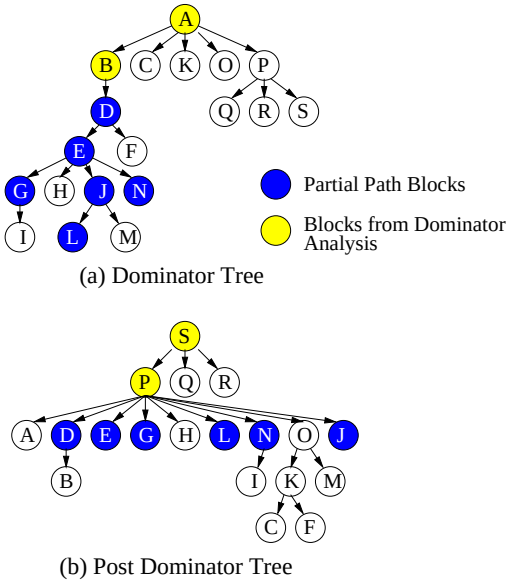

Figure 2: (a) Dominator and (b) post dominator trees for CFG in Figure 1. Partial path blocks and blocks added from dominator analysis are shown.

HPM-based code coverage is lossy, it provides a promising low-overhead alternative to program instrumentation.

2. HPM-BASED CODE COVERAGE

This section presents a HPM-based code coverage analysis framework. The framework consists of two main phases: a run-time collection phase in which PMU samples are gathered, and an off-line compiler-aided analysis phase. It is at this offline phase in the compiler framework that relationships between run-time branch information and program structures (statements, loops, subroutine calls) are known. Within the program's representation, compiler analysis can extend the inherent amount of hardware-monitoring information to provide increased code coverage results.

2.1 PMU Branch Execution Information

The experiments in this paper utilize the Branch Trace Buffer (BTB) registers supported on the Itanium-2 [10] PMU. The BTB acts as a circular buffer which is able to store the instruction and target addresses of the last four branches executed. Collectively, this set of branches defines a *branch vector*. The BTB also allows for a set of user-defined filters on the sampled branches such as sampling only taken branches or applying instruction range restrictions. In this framework, the BTB is configured to sample *only taken branches* in an effort to collect more information per sample. Since fall-through branches can be automatically tracked within a program CFG, a branch vector consisting of only taken branches indicates more information. Likewise, since compiler optimizations tend to emphasize fall-through paths over taken paths to improve code locality, aggressively optimized applications are more likely to execute fall-through branches. During run-time, the BTB is periodically sampled and the corresponding branch vector is stored in a table that keeps track of encountered branch vectors. The set of encountered branch vectors are then used in the offline phase for analysis.

2.2 Compiler Support for Code Coverage

The first component of off-line compiler analysis is to associate the branch vectors with *partial paths* of the CFG representation of program. A partial path is the set of basic blocks that are known to execute from a given branch vector. Figure 1 shows an example of the entire off-line process for a particular branch vector and program CFG. The numbered branches in Figure 1 show the four taken branches that are indicated by a single branch vector sample. In partial path creation, these four branches are mapped onto the original CFG to create the partial path shown in dark shaded blocks. Here, a partial path of six basic blocks can be determined from the branch vector. The following section discusses dominator analysis which can utilized to systematically infer other blocks which are guaranteed to have executed.

2.2.1 Dominator Analysis

A number of compiler optimizations rely on dominator analysis [1] to determine guaranteed execution relationships of blocks in a CFG. There are two commonly analyzed dominator relationships: dominance - basic block u dominates basic block v if every path from the *entry* of the CFG to basic block v contains basic block u, and post-dominance - basic block u post-dominates basic block v if every path from v to the *exit* of the CFG contains basic block u. Tree representations of the information summarize the dominance and post-dominance relationships. For example, in a dominator tree, each node dominates all nodes underneath it. The dominator tree representations for the CFG of Figure 1 are shown in Figure 2(a) and Figure 2(b). In the example, applying dominator analysis to partial path D, E, G, J, L, N guarantees that basic blocks A, B, P, S must have also executed. In this case, the execution of 10 basic blocks could be guaranteed by the collection and analysis of a single branch vector.

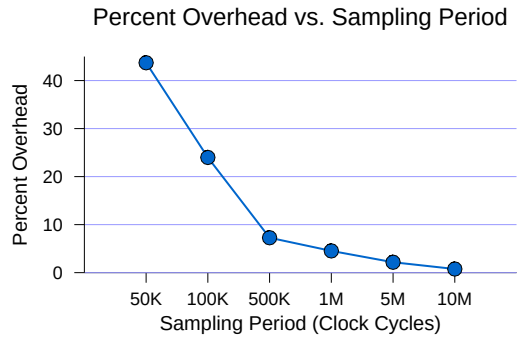

Figure 3: Overhead of run-time collection of branch vectors for various sampling periods.

Benchmark	# Ops	# Covered Ops
164.gzip	6,466	3,063 (47%)
175.vpr	23,573	12,229 (52%)
177.mesa	89,006	7,390 (8%)
179.art	2,201	1,515 (69%)
181.mcf	1,973	1,401 (71%)
183.equake	3,033	2,265 (75%)
188.ammp	19,562	5,835 (30%)
197.parser	17,541	11,271 (64%)
256.bzip2	5,095	3,138 (62%)
300.twolf	40,490	15,705 (39%)

Table 1: Number of instructions per benchmarks and actual code coverage.

3. EXPERIMENTAL EVALUATION

3.1 Methodology

The experiments in this paper are performed using a set of the *SPEC CPU 2000* benchmarks compiled with the OpenIMPACT [13] Research Compiler on an Itanium-2 with the 2.6.10 kernel. Benchmarks are compiled with the base OpenIMPACT configuration which include classical optimizations and profile-directed optimizations. The applications run using a HPM data collection tool developed using the perfmon interface and libpfm-3.1 library [5, 7]. The tool can be configured to collect branch vector samples at regular or randomized intervals. Branch vector samples are fed into an OpenIMPACT software module that performs compiler-aided code coverage analysis. Results are compared to an instrumentation-based code coverage tool developed with Pin [6].

3.2 Results and Analysis

3.2.1 PMU Sampling Overhead

There are a few main causes which contribute to the overhead incurred in collecting branch vectors. First, an interrupt occurs every sampling period which copies the BTB registers into a kernel buffer. As the sampling period increases, this causes the overhead to increase because interrupts happen more frequently. Afterwards, the data must be periodically read from the kernel buffer, processed, and stored.

Figure 3 shows the effect of sampling period on the run-time collection of branch vectors. The sampling period is varied from 50K to 10M clock cycles and the percent overhead is an average of overhead measurements across the experimental benchmarks. A trade-off clearly exists between overhead and the amount of information gathered. Using a smaller sampling period increases the number of samples and therefore the quality of information provided for code coverage analysis. However, as the sampling period is decreased, the overhead increases.

3.2.2 PMU Coverage Characteristics

Before exploring code coverage data, it is important to understand the size of the evaluated benchmarks as well as their run-time instruction footprints. Table 1 shows the size of each benchmark in number of low-level IR instructions as well as the number of these instructions that are actually covered during run-time. Code size varies greatly in this set of benchmarks ranging from *181.mcf* with 1,973 instructions to *177.mesa* with 89,006 instructions. The number and percentage of covered instructions also ranges greatly from 8% coverage for *177.mesa* to 75% coverage for *183.equake*.

Figure 4 presents PMU-based code coverage normalized to the total instruction counts shown in Table 1. 100% would mean that PMU-based code coverage has covered all the instructions that have actually been executed. For each benchmark, four sampling periods is shown; 100K, 1M, 10M and 100M clock cycles. Code coverage is divided into three main categories; Single BB, Branch Vectors, and Branch Vectors w/ Dominator Analysis:

Single BB: The first basic block of each branch vector is used for marking covered instructions. This is used to simulate PC-sampling where a single PC is mapped to a basic block.

Branch Vectors: The branch vectors are mapped to partial paths and each basic block in the path is marked as covered.

Branch Vectors w/ Dominator Analysis: Dominator analysis is performed on partial paths to mark additional basic blocks.

Figure 4 shows that *Single BB* leaves much to be desired in term of code coverage. Even at a low sampling period of 100K cycles, the highest code coverage percentage is barely over 50% for 181.mcf. 164.gzip, 175.vpr and 300.twolf perform particularly bad at only around 21-22% of the code coverage. The percentage only decreases as the sampling period increases for each benchmark. By sampling branch vectors and mapping them back to partial paths, there are substantial increases in the percentage of covered code that can be discovered using PMU-based code coverage. At the lowest sampling period, the coverage percentage increases by an average of 14%. 183.equake has the greatest improvement due to using branch vectors at around 30%.

Compiler-aided dominator analysis significantly extends the amount of code coverage information. As shown in Figure 4, for the lower sampling periods, it consistently provides an additional 15-25% to the code coverage percentage. The exception is *177.mesa* which improves less that 10% at all sampling periods.

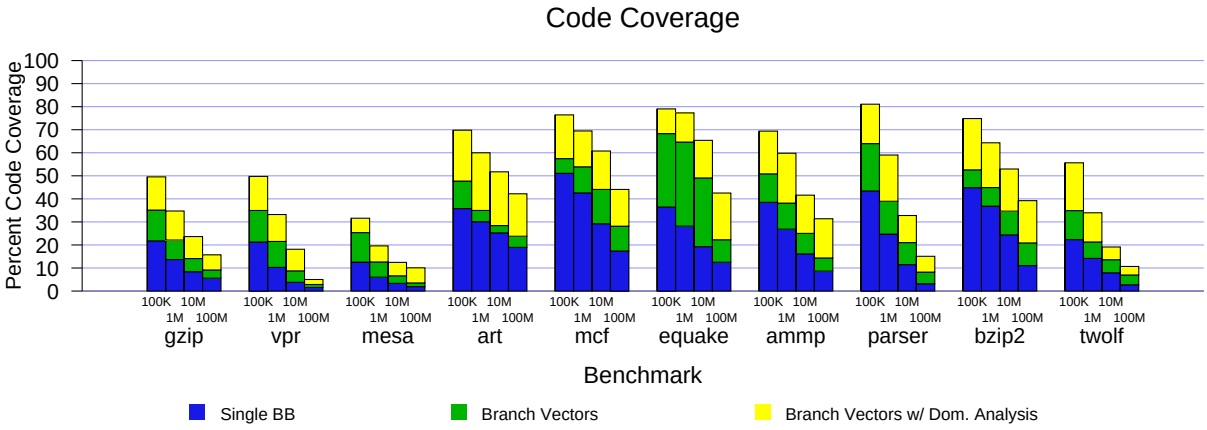

Figure 4: Code coverage across different sampling periods (100K, 1M, 10M 100M) showing the effects of 1) using a single basic block per sample, 2) using branch vectors to create partial paths and 3) extending partial path information by using dominator analysis.

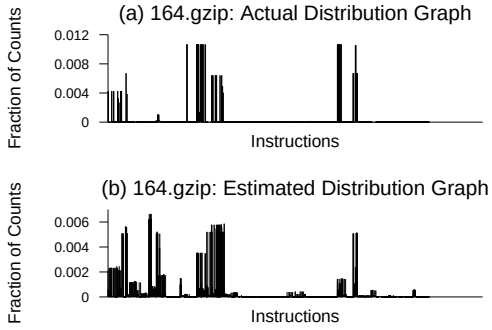

Figure 5: Instruction execution distribution across address range for *164.gzip* (a) actual and (b) PMU.

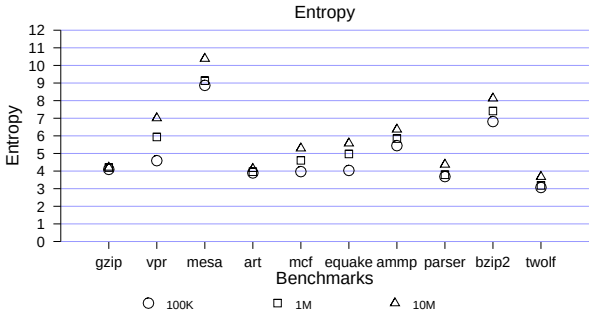

Figure 6: Entropy (Kullback-Leibler divergence) of actual/PMU coverage.

3.2.3 PMU Entropy Analysis

Figure 5 shows the probability distribution graphs for the code execution of *164.gzip* determined by both complete coverage (a) and PMU-based coverage (b). Although sampling in PMU-based coverage may miss program behavior, the instruction execution distributions appear similar. However, more detailed analysis can assess the overall ability of PMU code coverage data to accurately characterize the actual coverage. The relative entropy (Kullback-Leibler divergence [11]) defines the distance between two probability distribution functions. Let two discrete distributions have probability functions p_k and q_k. The relative entropy of p with respect to q is defined by:

$$d = \sum_{k=0}^{n} p_k log_2(\frac{p_k}{q_k})$$

Figure 6 presents the relative entropy numbers between actual and PMU code coverage. Three sampling rates are examined: 100K, 1M, and 10M. Results indicate that the average divergence between the actual and PMU distributions is about five. While the number is relative, it can be used to quantify the deviation from complete code coverage results. For instance, for *175.vpr*, *181.mcf*, and *256.bzip2* the divergence over the sampling rates increase by 3-4, indicating that sampling will have a direct role in the coverage accuracy while the divergence of *164.gzip*, which has a smaller code execution footprint, is not effected by sampling rate. These results indicate that code coverage testing should deploy variable sampling rates to maximize the trade-off between code coverage and testing overhead.

3.2.4 Aggregating Multiple Runs

One opportunity to improve the quality of the PMU-code coverage approach is to aggregate data from multiple execution runs. This is simply a matter of collecting the PMU monitoring tool's output from multiple runs and applying the off-line analysis module. Figure 7(a) shows an example of aggregating up to 20 separate runs at a regular sampling period of 100K clock cycles. There are two general trends for the benchmarks shown in Figure 7(a). The first, is that some benchmarks such as *164.gzip*, *197.parser*, and *300.twolf* significantly improve their code coverage percentage by over 10%; *164.gzip* actually improves over 20%. In these cases, the aggregation of multiple runs seems very

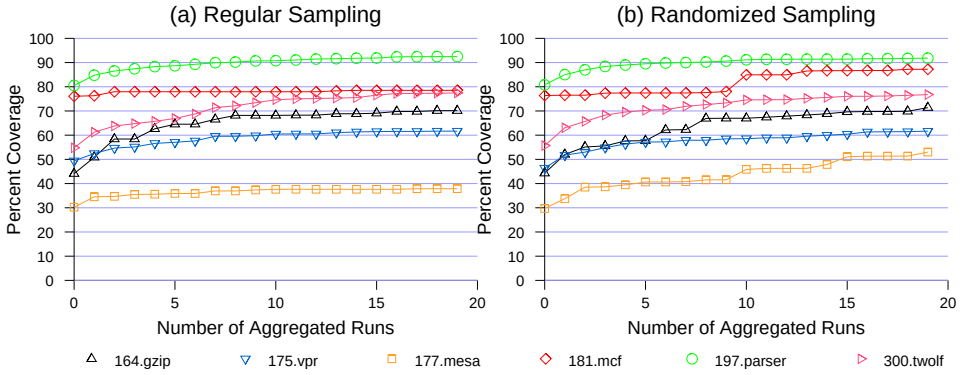

Figure 7: Code coverage percentage from aggregating multiple runs using (a) regular sampling period and using (b) randomized sampling period of 100K.

promising. Benchmarks *177.mesa* and *181.mcf* do not see as substantial improvements and their percentages stay fairly level with additional runs.

One of the issues facing periodic sampling is sampling aliasing. It is possible to miss important sections of code that periodically execute in the time between samples. Randomized sampling periods may be used in order to account for sampling aliasing. Figure 7(b) shows 20 aggregated runs using randomized sampling. The behavior of benchmarks such as *175.vpr*, *197.parser* and *300.twolf* are not significantly altered. However, *177.mesa* and *181.mcf* see substantial improvements. In these cases, the random sampling is able to uncover sections of code that regular sampling does not discover.

4. CONCLUSION

The initial rationale and results of using a PMU-based system for code coverage are presented. Overall, PMU-based code coverage shows promise since a low overhead (less than 3%) can obtain upwards of 80% code coverage, with an average of 50%. Several techniques to increase code coverage information are illustrated: the use of branch vectors, compiler analysis, random sampling periods, and the use of multiple application runs. There is a considerable amount of future work to investigate in both the compiler and testing domains, specifically: examining the effects of code optimization on HPM code coverage, exploring use of multiple application executions to create full coverage results, and studying the inter-procedural and library behaviors of applications within the HPM-based approach.

5. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. A-W Press, 1986.

[2] F. Allen. Control flow analysis. In *Proceedings of Compiler Optimization*, pages 1–19, 1970.

[3] J. Anderson and et al. Continuous profiling: Where have all the cycles gone? In *Proc. of the 16th ACM Symposium of Operating Systems Principles*, pages 1–14, October 1997.

[4] Bullseye Testing Technology. http://www.bullseye.com/.

[5] S. Eranian. The perfmon2 interface specification. Technical Report HPL-2004-200R1, Hewlett-Packard Laboratory, February 2005.

[6] C. L. et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, June 2005.

[7] Hewlett-Packard Development Company. perfmon project http://www.hpl.hp.com/research/linux/perfmon/.

[8] IBM. *PowerPC 740/PowerPC 750 RISC Microprocessor User's Manual*, 1999.

[9] IBM PureCoverage. http://www.pts.com/wp2077.cfm.

[10] Intel Corporation. Intel Itanium 2 processor reference manual: For software development and optimization. May 2004.

[11] M. Kearns and et al. On the learnability of discrete distributions. In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 273–282. ACM Press, 1994.

[12] Y. W. Kim. Efficient use of code coverage in large-scale software development. In *CASCON '03: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 145–155. IBM Press, 2003.

[13] OpenIMPACT Compiler. http://www.gelato.uiuc.edu/.

[14] OProfile System Profiler for Linux. http://oprofile.sourceforge.net.

[15] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th Symposium on Foundations of Software Engineering*, pages 241–251. ACM Press, 2004.

[16] B. Sprunt. Pentium 4 performance-monitoring features. In *IEEE Micro 22(4)*, pages 72–82, 2002.

[17] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the Symposium on Software Testing and Analysis*, July 2002.

[18] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Survey*, 29(4):366–427, 1997.