Efficient Instrumentation for Code Coverage Testing

Mustafa M. Tikir
tikir@cs.umd.edu

Jeffrey K. Hollingsworth
hollings@cs.umd.edu

Computer Science Department
University of Maryland
College Park, MD 20742

ABSTRACT

Evaluation of Code Coverage is the problem of identifying the parts of a program that did not execute in one or more runs of a program. The traditional approach for code coverage tools is to use static code instrumentation. In this paper we present a new approach to dynamically insert and remove instrumentation code to reduce the runtime overhead of code coverage. We also explore the use of dominator tree information to reduce the number of instrumentation points needed. Our experiments show that our approach reduces runtime overhead by 38-90% compared with *purecov*, a commercial code coverage tool. Our tool is fully automated and available for download from the Internet.

KEYWORDS

Testing, Code Coverage, Dynamic Code Patching, Dynamic Code Deletion, Dominator Tree, On-Demand Instrumentation

1. INTRODUCTION

Evaluation of Code Coverage is the problem of identifying the parts of a program that did not execute in one or more runs of a program. Developers and testers use code coverage to ensure that all or substantially all statements in a program have been executed at least once during the testing process. Measuring code coverage is important for testing and validating code during both development and porting to new platforms. Traditionally code coverage measurement tools have been built using static code instrumentation. During program compilation or linking, these tools insert instrumentation code into the binary executable file. The inserted instrumentation provides counters to record which statements are executed. The code inserted into the executable remains in the executable throughout the execution even though once a statement has been executed, the instrumentation code produces no additional coverage information. Moreover, these tools conservatively instrument all functions prior to the program execution even though some of them may never be executed. Leaving useless instrumentation in place increases the execution time of the software being tested especially if the program is long running and has many frequently executed paths (as most server programs due). For example, the *perl* benchmark from SPEC95 suite runs almost 20 times slower when the instrumentation code for code coverage is left in the executable during execution. Statically inserting all possibly needed instrumentation code increases the instrumentation overhead for large programs that execute only small portion of execution paths (common for the applications built from libraries).

In this paper we present a new approach to dynamically insert code and remove it when it does not produce any additional coverage information. To our knowledge, this approach has not been used in previous code coverage tools. Our goal in this paper is to show that deletion of instrumentation code used for code coverage produces faster code coverage results for long running programs. Although code coverage testing has traditionally been performed as a distinct phase of the software development process, by making it cheaper it potentially could be included in production code. This would allow feedback to developers about the behavior of the software once deployed. For rarely executed code, such as error cases, this type of feedback could be especially valuable. By significantly reducing the overhead of instrumentation code execution, our technique makes residual test coverage monitoring[8] more efficient. Our fast code coverage techniques also could be modified to sample the frequency of execution of program segments to provide additional information to a feedback-directed dynamic code optimization system.

Besides dynamic deletion of instrumentation code, we explore the use of more sophisticated binary analysis techniques to reduce the number of places instrumentation code needs to be inserted. Most existing code coverage tools insert instrumentation code at the beginning of each basic block. However, by automatically generating and using the dominator tree of a control flow graph, we can reduce the number of instrumentation points required.

We also explore the use of incremental function instrumentation to insert the necessary instrumentation code when a function is called for the first time during program execution. Existing code coverage tools conservatively insert all possibly needed instrumentation code even though it may never be executed in future runs. Thus, the overhead of instrumentation is significant for large programs containing a few frequently executed paths. Using incremental instrumentation of functions during program execution, we eliminate the instrumentation time for uncalled functions.

The rest of the paper is organized as follows: Section 2 describes the extensions made to dyninstAPI, a runtime code patching system, to implement our dynamic code coverage tool, Section 3 explains how dominator tree information is used to reduce the number of instrumentation points needed, Section 4 explains the steps of our algorithm for code coverage, Section 5 presents the results of a series of experiments conducted to evaluate our approach, Section 6 presents the related work. Section 7 summarizes our results and describes where to download the software.

2. OVERVIEW OF dyninst API

DyninstAPI is an Application Program Interface to a library that permits the insertion of code into a running program. This library provides a machine independent interface to permit the creation of tools and applications that use runtime code patching. The unique feature of this interface is that it makes it possible to insert and change instrumentation in a running program[7]. Implementations of dyninst are currently available for Alpha, Sparc, Power, Mips and x86 architectures.

Figure 1 shows the structure of dyninstAPI. A mutator process generates machine code from the high-level instrumentation code and transfers it to an application process. To insert new code, dynamic code patches, called trampolines, are placed at the point where the new code is to be inserted (shown in Figure 2).

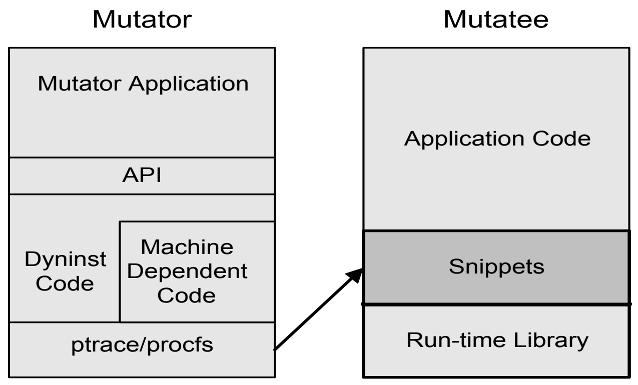

Figure 1. Structure of dyninstAPI

A base trampoline contains the relocated instruction(s) from the application address space and has slots for calling mini-trampolines both before and after the relocated instructions. Mini-trampolines store the machine code for high-level instrumentation code.

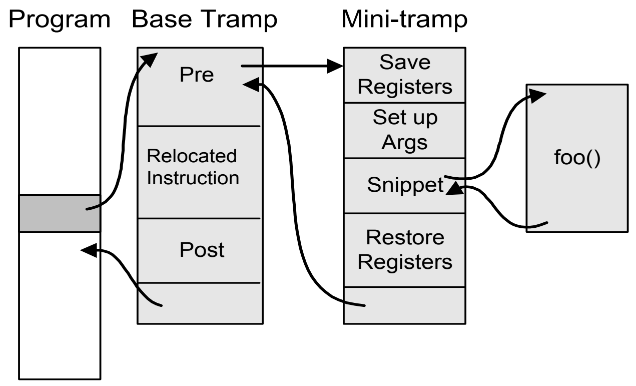

Figure 2. Instrumentation Code Insertion into a Program

The library also allows instrumentation code to be deleted. Instrumentation code deletion is a two-phase process that first removes the branch into the instrumentation code and then later deletes the trampoline to ensure that the instrumentation code being deleted is not executing.

To implement a code coverage tool using dyninst, we extended the API to provide information about control flow graphs, basic blocks, and the ability to map source code line numbers to machine instructions. To create the control flow graph of a function we use a variation on the two-pass algorithm presented in [5]. We then create the dominator tree of a control flow graph using the algorithm in [13]. In addition we extended the system to allow per instruction instrumentation.

Originally, dyninst only supported function level instrumentation. That is, the points to which instrumentation code can be inserted were function entry, function exit and call sites. For a code coverage tool, however, we need finer grained instrumentation. We added arbitrary instrumentation points to the library. At arbitrary instrumentation points, we need to preserve the machine's condition codes that are not live (and thus not saved) in function level instrumentation. We changed the base trampoline structure to save the processor state before the execution of instrumentation code, and then restore it after the instrumentation code but before executing any other user instructions.

Another enhancement to the dyninst API involves its memory allocator. Dyninst performs a number of optimizations when the memory is allocated for base trampolines and instrumentation code. One of these optimizations tries to allocate memory for code snippets close to the instrumentation point itself. By keeping the displacement to instrumentation code small, single word branch instructions can be used. Since the reachable displacement using one-word branch instructions is limited, when dyninst de-allocates memory, it compacts the free blocks. However, this optimization causes a significant instrumentation overhead when a large amount of instrumentation code insertion repeatedly triggers the compaction algorithm. Thus, we refined memory compaction to trigger only when memory for snippets runs low to improve overall performance.

3. USING DOMINATOR TREES

In this section, we explain our techniques to reduce the number of instrumentation points needed for our dynamic code coverage tools. We use properties of the immediate dominator tree of a control flow graph for instrumentation point selection.

3.1 Leaf Node Instrumentation

A *dominator tree* is a tree in which the root node is the entry basic block, and each basic block *d* dominates only its descendants in the tree. A basic block *d* of a flow graph dominates basic block *n*, *d dom n*, if every path from the entry basic block of the flow graph to *n* goes through *d*. Each basic block *n* has a unique immediate dominator *m* that is the last dominator of *n* on any path from the entry basic block to *n*.

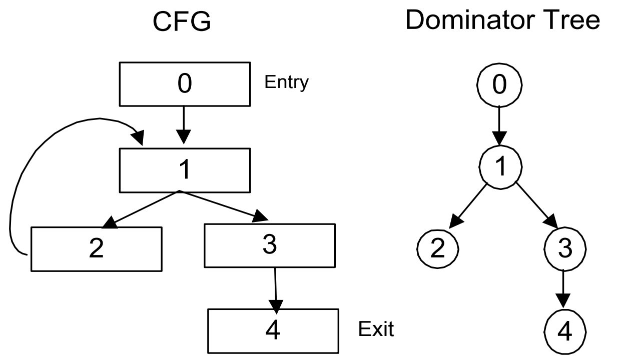

Figure 3. A simple CFG and Its Dominator Information

The key property of the dominator trees for our work is that for each basic block *n* in a dominator tree if *n* is executed, all the basic blocks along the path from root node to *n* in dominator tree are also executed. Figure 3 gives an example of a control flow graph and its dominator tree information.

Using the fact that coverage of a basic block might be inferred by coverage of other basic block(s), we can increase the coverage information obtained per instrumentation point by omitting the instrumentation code from an internal node of the dominator tree. That is, the instrumentation of the leaf nodes in the dominator tree will produce enough information to compute the coverage of internal nodes in the dominator tree.

3.2 Non-Leaf Node Instrumentation

Leaf node instrumentation is necessary but not sufficient to produce correct code coverage results. This is because the flow of control does not have to follow a path in the dominator tree. That is we cannot guarantee that execution of basic block *n* is always followed by the execution of another basic block that is dominated by *n*. In some cases there will be cross edges in the dominator tree for the execution path. If the cross edges originate at leaf level basic blocks in the dominator tree, leaf level instrumentation will be sufficient. However if there exists a cross edge originating from internal node, *n*, the execution path may not include any leaf level basic blocks of the sub-tree rooted at *n*.

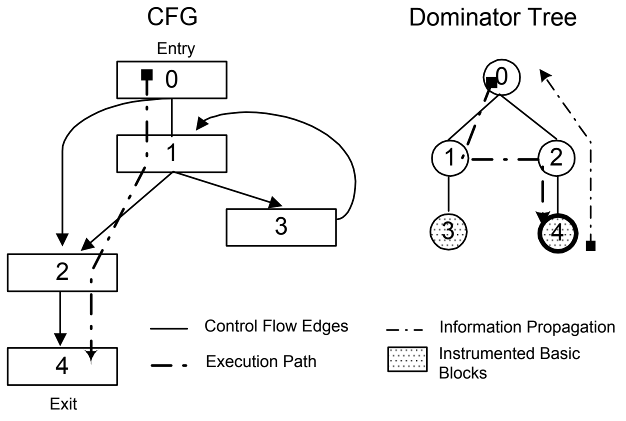

Figure 4. Why leaf level instrumentation is not sufficient

Figure 4 gives an example for an execution path that creates a cross edge originating from internal node in the dominator tree. For this control flow graph, leaf level basic block instrumentation is not sufficient for correct code coverage results. For example, if we only instrument leaf level basic blocks 3 and 4 in the dominator tree, when the flow of control leaves at the exit node of control flow graph, only basic block 4 will be marked as executed. When we propagate the information obtained from the execution of leaf node towards the root, we infer that basic blocks 2 and 0 also executed. However no information about basic block 1 will be given, thus it is assumed to be unexecuted. Since the flow of control did not enter basic block 3, the leaf level instrumentation did not give any information to us about the basic blocks that dominate 3, which are 1 and 0.

To correctly capture this case, we also instrument basic block *n* if *n* has at least one outgoing edge to a basic block *m* that *n* does not dominate. In this example basic block 1 has an outgoing edge to 2 and 1 does not dominate 2. We choose basic block 1 to be instrumented besides basic blocks 3 and 4. We selected our approach because it is fast to compute. With our online approach, binary analysis time must be kept to a minimum.

Alternatively, a combination of dominator and post-dominator tree information could be used to reduce the number of instrumentation points needed compared to using only dominator tree information. That is, the execution of a basic block can also be deduced by execution of another basic block that is post-dominated by the former. However, our goal is not to find the optimal number of instrumentation points[6][14], but to minimize the sum of the analysis and instrumentation overhead. Although we use Langauer-Tarjan[13] algorithm that is linear in number of edges in a control flow graph, dominator tree construction for a control flow graph is an expensive computation relative to the limited benefit we can expect. That is, additional post-dominator information would double our binary analysis time without a significant reduction in instrumentation overhead. Thus we chose to use only dominator tree information.

4. CODE COVERAGE ALGORITHM

We implemented two slightly different versions of our dynamic code coverage algorithm: code coverage with pre-instrumentation and code coverage with on-demand instrumentation. These algorithms differ in what functions are instrumented and when the instrumentation code is inserted. The selection of points to be instrumented is based on the same criteria in both implementations. For both algorithms, during the execution of the program being tested we determine if instrumentation code can be deleted, and remove it. At program termination, we record the results of code coverage by propagating line coverage information towards the root of dominator tree.

The first step of our algorithm with pre-instrumentation is to create the control flow graph and dominator tree for each function in the application. Next, for each control flow graph we choose basic blocks to be instrumented using the criteria explained in Section 3. For each basic block to be instrumented we create a Boolean variable which is initialized to false indicating that the block has not yet executed. We insert code at the beginning of the basic block that sets the corresponding Boolean variable to true. Our code coverage tool automatically creates the control flow graph, generates the dominator tree and inserts the instrumentation code.

With on-demand instrumentation only breakpoints are inserted at the beginning of each function in the application prior to the execution. During the execution of the program, when a breakpoint is reached, the control flow graph of that function is generated and the necessary instrumentation code is inserted. Thus, if the function is not called during the execution, neither the control flow graph nor the instrumentation code is generated for it.

For better performance for long running programs with many hot basic blocks and paths, we delete instrumentation code during the execution of the program. Deletion of instrumentation code includes restoring original instructions and de-allocating base trampoline and min-trampoline space. However, there is a tradeoff in instrumentation code deletion. Sometimes deletion may introduce more overhead than the resulting performance improvement. This is due to the fact that it takes time to check what is already executed and what can be deleted. For example, if there is a lot of instrumentation code that never execute, the checks will mostly introduce overhead instead of improvement.

Instrumentation code can be deleted using different policies. One simple method is to delete instrumentation code at fixed time intervals. Another possibility is to delete the instrumentation code automatically just after the first time it is executed. In our current implementation, instrumentation code is deleted at fixed time intervals. It is a simple approach, easy to program and improves the execution time of the program being tested significantly. The deletion interval is a tunable parameter to our code coverage systems.

At program termination we record the results of code coverage. For our code coverage algorithms that use dominator tree information for instrumentation, we simply read the values of variables assigned to basic blocks instrumented and propagate the information along the path in the dominator tree towards the root. If the variable for a basic block is set we mark all dominators as executed. Our code coverage tool then either generates a binary file that contains information about which lines were executed or displays the coverage information through its user interface.

Relative to static instrumentation that can completely re-structure a binary, our approach uses a base trampoline and a mini-trampoline for each instrumentation code inserted. Therefore the cost of each instrumentation point also includes the execution of branch/call instructions from executable address space to the base trampoline and from the base trampoline to the mini-trampolines. However, the fact that we can remove instrumentation code at runtime helps to offset this penalty.

5. EXPERIMENTS AND RESULTS

To evaluate the effectiveness of our approach, we ran a workload of test programs with and without using dominator information and varying the dynamic code deletion interval. As a comparison, we also ran the applications through *purecov* (*version 4.1 Solaris 2.6*), a commercial code coverage tool that uses static code editing. We measured the execution time of programs instrumented by our dynamic code coverage tools including the setup time for control flow graph generation, dominator tree construction, and instrumentation code insertion. We tested code coverage for *PostgreSQL*, an object-relational DBMS, using the *Wisconsin*[9] and *crashme*[2] benchmarks, the C programs and two of the Fortran programs from the *SPEC95* benchmarks[4]. Experiments were conducted on a SUN-SPARC ULTRA 10 with 500MB of main memory, and compiled with gcc version 2.95.1 with debug option enabled. We also measured the total number of basic blocks in the program being tested and the number of instrumentation points needed when dominator tree information is used. We ran the same set of experiments for both code coverage with pre-instrumentation and code coverage with on-demand instrumentation.

5.1 Reduction in Instrumentation Points

To quantify the benefits of using dominator tree information, we calculated the number of instrumented basic blocks with and without using dominator tree information. We repeated the experiments using our on-demand instrumentation algorithm.

Figure 5 summarizes the ratio of the number of instrumented basic blocks to the total number of basic blocks in the application for the programs we tested. Details about the code coverage instrumentation statistics for all programs we tested are given in the tables in Appendix A. For each program, there are four bars. The bars labeled *all blocks* correspond to the code coverage tools with all basic blocks instrumentation and the ones labeled *dominator* indicate use of dominator tree information. The *OD* suffix indicates our code coverage algorithms with on-demand instrumentation.

Figure 5 shows that using dominator tree information with pre-instrumentation reduced the number of instrumentation points needed by 34% to 49% compared to all basic blocks instrumentation. Similarly, it shows that using dominator tree information with on-demand instrumentation, we reduced the number of instrumentation points needed from 33% to 49%, which corresponds to 42% to 79% reduction in the total number of basic blocks instrumented when all blocks instrumentation and dominator tree information is used.

Figure 5 shows that the gain using dominator tree information is less for *gcc*, *perl*, and *vortex* than the other programs tested. These programs have lexical analyzers and parser functions in them. These types of functions have complex control flow graphs containing many basic blocks with few instructions and many control flow edges. These properties result in a large number of leaf level basic blocks in the dominator trees and also a large number of internal basic blocks that require instrumentation (as described in Section 3.2)

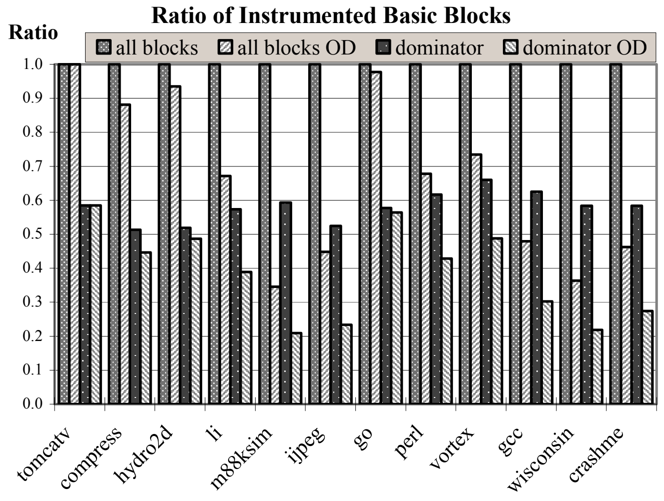

Figure 5. Ratio of Instrumented Basic Blocks to the Total Number of Basic Blocks

Figure 5 also shows that using on-demand instrumentation, our code coverage algorithm reduces the amount of instrumentation code inserted compared to static instrumentation. Combining on-demand instrumentation with dominator trees consistently results in the fewest number of instrumented basic blocks. However, the number of instrumentation points needed in *tomcatv* is not reduced by on-demand instrumentation, as *tomcatv* has no un-called functions in its execution. Overall we were able to eliminate instrumentation from 42% to 79% of the basic blocks in the executables.

5.2 Coverage Percentage Curves

In this section, we present the source code line coverage percentage versus time to show how rapidly various applications reach certain levels of coverage. We measured the source code line coverage percentage by stopping the running process at fixed time intervals and calculating the percentage source lines executed.

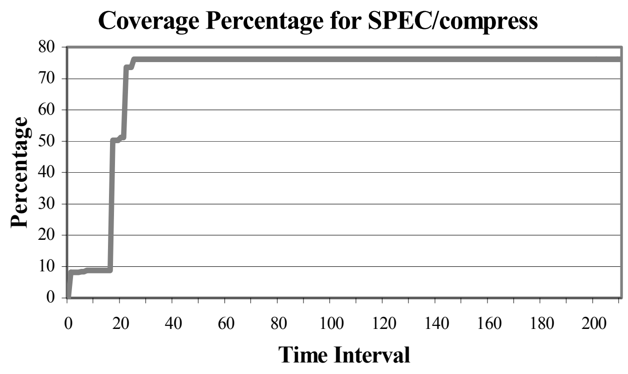

Figure 6. Source Line Coverage Percentage for compress

Figure 6 shows the source line coverage percentage versus time for *compress* from SPEC95 benchmark suite. The coverage percentage in Figure 6 steeply increases to 76% in the first 18% of the execution time and stays at this level through the rest of the execution. That is, most of the basic blocks that will execute are covered at the beginning of the program and during the rest of its execution mostly the same basic blocks are re-executed.

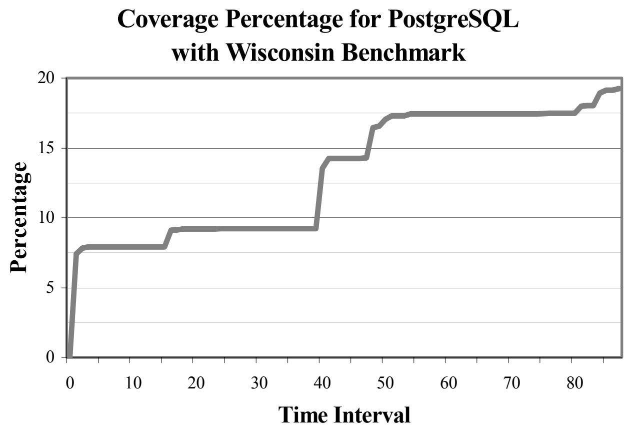

Figure 7. Source Line Coverage Percentage for PostgreSQL with Wisconsin Benchmark

Figure 7 shows the source code line coverage percentage for *PostgreSQL* using *Wisconsin* benchmark queries. The *Wisconsin* benchmark queries are designed to measure the query optimization performance of database systems using selection, join, projection, aggregate, and simple update queries. We conducted the experiments using a single-user version of *PostgreSQL*. The fact that the database was in single user mode partially explains the relatively low coverage percentage in Figure 7.

Unlike Figure 6, the source code line coverage percentage for *PostgreSQL* using the *Wisconsin* benchmark increases gradually to 19% through the whole execution, staying around 10% in the first half. However, the source code line coverage percentage mostly remains steady for several intervals during the execution indicating the existence of many frequently executed paths and re-execution of many basic blocks during these intervals.

Figure 7 also shows that, unlike *compress*, the time spent executing instrumentation code is distributed among these intervals rather than being at the beginning of the program execution.

5.3 Execution Time

We next look at the impact of dynamic code deletion and dominator information usage for the applications in Section 5.2. We present the execution times using our techniques and compare it to the commercial code coverage tool *purecov*.

Figure 8 shows the slowdown ratios of *compress* with respect to original execution time. It has five kinds of bars. The bar labeled *PC* shows the execution time slowdown ratio for *compress* instrumented using *purecov*. The rest of the bars are divided into four categories; each category corresponds to slowdown ratios of *compress* instrumented using one of our dynamic code coverage algorithms. Categories labeled *dominator* use dominator tree information for instrumentation where the ones labeled *all blocks* indicate our dynamic code coverage tools with all basic blocks instrumentation. The suffix *OD* indicates use of on-demand function instrumentation. In each category, the bars are labeled with numbers to represent different instrumentation code deletion intervals (in seconds). Bars labeled 0 indicate that instrumentation code is not deleted at all. Each bar is composed of two or three segments.

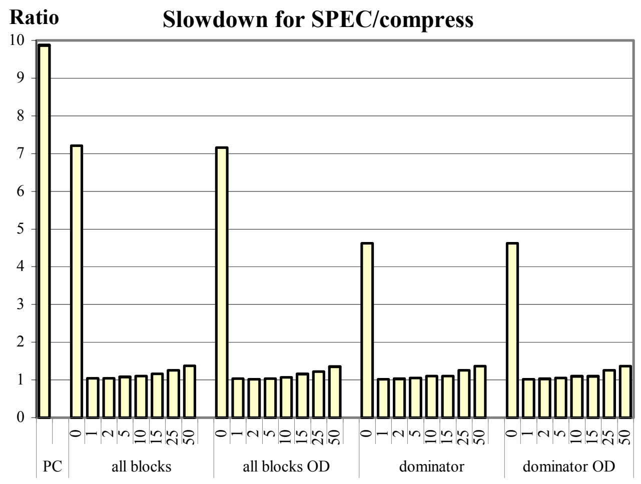

Figure 8. Execution Time Slowdown Ratios for compress

Figure 8 shows that all of our code coverage tools significantly outperform *purecov* execution for all deletion intervals studied. It also shows that there is a significant decrease in execution time when dynamic instrumentation code deletion is enabled. This is due to two reasons; 1) Most of the instrumentation code is executed at the beginning and deleted shortly after it is executed, and 2) There are few basic blocks in compress and the overhead of checking whether instrumentation code is executed or not during the deletion intervals is not significant. Even if we instrument all basic blocks, after a couple deletion intervals most of the instrumentation code is deleted. This explains the relatively insignificant gain when using dominator tree information despite the fact that dominators were able to eliminate instrumentation points for over 55% of the basic blocks (as shown in Figure 5). Likewise on-demand instrumentation provides little benefit. Figure 8 also shows that the execution times increase slightly for larger deletion intervals for all of our code coverage tools due to re-execution of some instrumentation code in the first couple deletion intervals.

Figure 8 also shows that without dynamic code deletion, our dynamic code coverage tools using dominator tree information outperform the ones using all basic blocks instrumentation. Surprisingly, our techniques outperform *purecov* execution even without code deletion when all basic blocks instrumentation is used. This is due to the fact that sometimes *purecov* inserts more instrumentation code than our code coverage tools with all basic blocks instrumentation (An examination of purecov code indicates that purecov inserts unnecessary instrumentation code around the pseudo-instruction that implements integer division of the SPARC).

Figure 8 also shows that, for *compress*, our dynamic code coverage tools with on-demand function instrumentation slightly outperform the ones with pre-instrumentation. This is due to the fact that 89.1% of the total basic blocks in *compress* are executed during the program execution and the setup time for *compress* is not significant compared to the total execution time.

For *compress* instrumented by our dynamic code coverage tools, the best execution time occurs using a 2-second deletion interval and is 90% better than *purecov* execution time. The slowdown ratio for our best execution time with respect to original execution is 1.003. That is, our dynamic code coverage tool introduces only a 0.3% run time overhead compared to the original execution of *compress*.

Figure 9 presents the execution time slowdown ratios, with respect to original execution time, of *PostgreSQL* for the *Wisconsin* benchmark instrumented by purecov and our dynamic code coverage tools. The gray segment in each bar represents the setup time for each tool where the bottom light colored segment is execution time of the program. For our dynamic code coverage tools with on-demand instrumentation, gray segment represents the control flow graph generation and instrumentation time, which is distributed throughout the execution. The dark top segment represents the time spent during instrumentation of breakpoints at function entry points. (Although setup times were shown in Figure 8, they were so insignificant for *compress* that they were not visible).

Figure 9 shows that setup times for our code coverage tools with pre-instrumentation is significant due to the existence of many complex control flow graphs and the large number of basic blocks in *PostgreSQL*. That is, the control flow generation and instrumentation code insertion for all functions in *PostgreSQL* introduces a significant overhead. The setup time for our code coverage tools with on-demand instrumentation is not significant since it only requires inserting breakpoints at the beginning of the functions. Figure 9 also shows that control flow graph generation and instrumentation of functions for our dynamic code coverage tools with on-demand instrumentation takes significantly less time compared to our tools with pre-instrumentation.

Figure 9 shows that our code coverage tool with pre-instrumentation and all basic blocks instrumentation is outperformed by *purecov* significantly. This is due to two reasons. First, even though only 36% of the basic blocks are executed, pre-instrumentation creates control flow graphs for un-called functions and inserts instrumentation code for all basic blocks. Unlike our code coverage algorithms, *purecov* does not suffer from control flow graph generation and instrumentation code insertion during execution. Second, the deletion interval overhead, for checking whether instrumentation code is executed or not, is significant when many basic blocks are never executed. That is, most of the checks during the deletion intervals are not profitable but introduce overhead. Figure 9 shows that our code coverage tool with pre-instrumentation and dominator tree information usage performs slightly better than purecov since it introduces fewer instrumentation points compared to all block instrumentation.

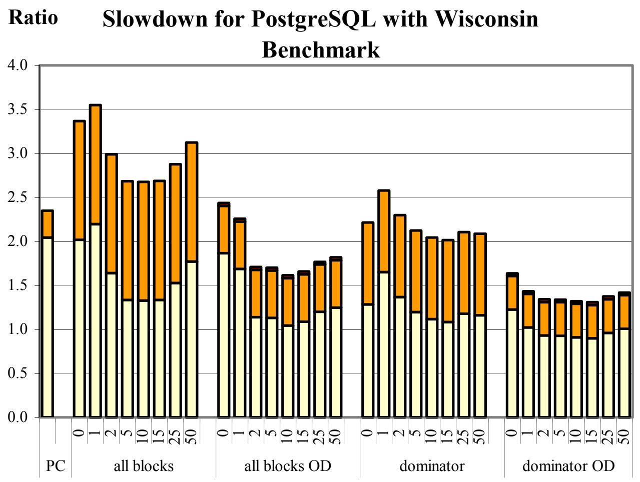

Figure 9. Execution Time Slowdown Ratios for PostgreSQL with Wisconsin Benchmark

Figure 9 also shows that our code coverage tools with on-demand instrumentation outperform our code coverage tools with pre-instrumentation since they do not generate control flow graphs for un-called functions nor insert instrumentation code for basic blocks that are not executed. Our on-demand instrumentation technique also reduces the deletion interval overhead by introducing instrumentation code incrementally that eliminates the checks that would otherwise be done in previous deletion intervals.

Like in Figure 8, the results in Figure 9 indicate that using dynamic code deletion produces faster code coverage results. Unlike Figure 8, every-second deletion performs slightly worse than no dynamic code deletion for pre-instrumentation case, since the more instrumentation code must be checked.

Figure 9 shows that combining on-demand instrumentation and dominator tree information usage is complementary. While using dominator tree information reduces the number of instrumentation points needed, using on-demand instrumentation reduces the setup time and deletion interval overhead of checking whether the instrumentation code can be deleted or not. Using both dominator tree information and on-demand instrumentation, we reduced the amount of instrumentation code inserted by 78.2% compared to the total number of basic blocks in the program.

For this application, the best execution time occurs when a 15-second deletion interval is used. The slowdown ratio for our best execution time with respect to original execution is 1.31 and it is 44% better than *purecov* execution time.

We present execution time slowdown ratios for the rest of the programs in Appendix B for the interested reader. The format of the rest of the graphs is exactly same with the ones in this section.

5.4 Overall Slowdown

We also calculated the slowdown ratio with respect to the original execution time for programs instrumented using purecov and our dynamic code coverage tool. We took the results for 2-second deletion interval for our dynamic code coverage tools (We decided to present the results for 2-second deletion interval as representative of our techniques rather than using the best deletion interval for each application).

Table 1 presents the execution time slowdown ratios (computed as the ratio of instrumented execution time to un-instrumented execution time) for the programs we tested. In the second column we give the original execution times in seconds. The next four columns give the slowdown ratios of the programs instrumented by our dynamic code coverage tools using dominator tree information and all basic blocks instrumentation for both pre-instrumentation and on-demand instrumentation. The results presented in Table 1 for our code coverage tools include setup time for control flow graph generation, dominator tree construction and instrumentation. The last column of the table gives the slowdown ratios of the programs instrumented using *purecov*.

Table 1 shows that purecov slows down the execution from 1.8 for *tomcatv* to 19.8 times for *perl*. However our dynamic code coverage tool with on-demand instrumentation slows down the execution only a factor of 1.002 to 2.6 using dominator tree information Our code coverage tools with on-demand instrumentation frequently outperform the ones with pre-instrumentation.

	Original Execution Time (sec)	Slowdown Using Dominator Tree Information		Slowdown Using All Basic Blocks Instrumentation		Slowdown using purecov
		Pre-Inst.	On-Demand Inst.	Pre-Inst.	On-Demand Inst.	
tomcatv	77.9	1.003	1.002	1.003	1.002	1.83
postgres -crashme	254.4	1.80	1.43	2.16	1.56	2.09
postgres -wisconsin	90.5	2.30	1.34	2.99	1.71	2.35
hydro2d	764.4	1.01	1.01	1.01	1.01	2.73
ijpeg	223.9	1.07	1.08	1.13	1.14	4.74
go	118.3	1.08	1.06	1.23	1.20	5.23
vortex	50.3	1.69	1.48	1.90	1.66	7.27
gcc	50.9	3.90	2.58	4.96	3.26	8.97
m88ksim	133.5	1.11	1.06	1.14	1.06	9.43
li	373.4	1.02	1.01	1.03	1.02	9.45
compress	219.4	1.03	1.003	1.04	1.02	9.88
perl	67.1	2.53	2.37	2.70	2.56	19.78

Table 1. Comparison of slowdown ratios with respect to original execution times for our dynamic code coverage tools with on-demand and pre- instrumentation, and *purecov*.

Table 1 shows that the difference between the slowdown ratios using our tools with on-demand instrumentation and pre-instrumentation is higher for *gcc*, *postgres*, *vortex*, and *perl* compared to the other programs we tested. This is due to the fact that these programs have many basic blocks and control flow edges and a significant portion of these basic blocks are not executed. That is, our code coverage algorithm with pre-instrumentation spends a significant amount of time to create control flow graphs and insert instrumentation code for un-called functions, and thus introduces a significant amount of instrumentation code that is not executed but must be checked during each deletion interval.

6. RELATED WORK

The two systems most closely related to our dynamic code coverage tool are the commercial code coverage tools, PureCoverage[3] and C-Cover[1]. To locate untested areas of code, PureCoverage uses Object Code Insertion (OCI) technology to insert usage tracking instructions into the object code for each function and block of code during a post compilation, pre-link pass. Additionally PureCoverage also counts the number of function calls for the functions and execution counts of each source line executed. However, since they use a small number of bits for each counter, only an approximate count is returned. Similarly, C-Cover automatically adds probes to C and C++ source code by intercepting calls to the compiler[1]. C-Cover also displays condition/decision coverage and function coverage results. Unlike our dynamic code coverage tool, these tools statically edit the source code or executable and the code inserted remains inside the executable during the executions. Moreover, these tools conservatively insert all instrumentation code for each function in the application. Our dynamic code coverage tool also uses dominator tree information to reduce the number instrumentation points and incremental function instrumentation to reduce the overhead of instrumentation for un-called functions.

Pavlopoulou and Young[8] present a prototype system that implements residual test coverage monitoring for Java applications. The purpose of residual test coverage monitoring is to provide feedback from actual use of deployed software to developers, helping developers to validate and refine the models they have relied upon in quality assurance. However, it is unlikely to be accepted by users unless monitoring performance impact is very small. To reduce the cost of continued monitoring, the prototype presented[8] selectively re-instruments the program under test to monitor only the coverage obligations that remain unmet. However, our technique can be used to reduce monitoring overhead by deleting instrumentation code during the program execution, which will make residual test coverage monitoring more efficient.

Agrawal[6] also uses properties of dominator trees as part of software testing. Agrawal presents techniques to find small subsets of nodes in a control flow graph with the property that if the subset is covered by a test case, the remaining nodes are automatically covered. The technique finds the strongly connected components of the union of pre- and post dominator trees of a control flow graph. Unlike our work, this approach spends a significant amount of time to find the nodes to be instrumented by running two algorithms for dominator tree construction and one to find strongly connected components. The role of dominator trees is also different. Agrawal uses properties of dominator trees to provide programmers guidance about how to create test cases to provide significant code coverage for each case. In contrast, our use of dominator trees is to efficiently measure code coverage.

Path Profiles[15] can be used to compute the code coverage via a multi-phase algorithm. The key idea behind the path-profiling algorithm is to identify the potential paths with states that are represented as integers. A minimal number of edges in a DAG are labeled with integer values such that each path from entry to the exit of the DAG produces a unique sum of the edge values along that path. At the exit node of a DAG, the unique sum is used to identify the executed path and increment the counter assigned to it. At program termination, the non-zero counter values of the paths can be used to identify covered lines in the executable after regeneration of the executed paths. However, with complex control flow graphs and many executed paths, the path-profiling algorithm requires many counters. Also the path regeneration phase may introduce significant overhead before the execution terminates. Unlike our code coverage algorithm, deletion of instrumentation code is not possible as the labels assigned to edges are required throughout the execution to identify acyclic paths that will possibly be executed. The results presented for path profiling in [15] include only the run-time overhead of the instrumentation code during the execution. That is, they do not include the time spent for minimal edge labeling of DAGs, insertion of instrumentation code and path regeneration from the unique identifiers of executed paths. In contrast, the overhead of our code coverage tools presented in this paper includes the analysis, setup and instrumentation time.

Digital Continuous Profiling Infrastructure (DCPI)[10, 11, 16] is a suite of software profiling tools that provide transparent, low-overhead profiling of complete systems. DCPI uses hardware performance counters on the Alpha processors to sample the program counter periodically, and can be setup to produce basic block flow graphs annotated with approximate execution counts. These execution counts, however, are approximate values and may not exist for each instruction or basic block in the executable, which makes it difficult to produce exact code coverage results.

Whole Program Paths (WPP)[12] can also be used to extract code coverage results that give the dynamic behavior of a program. WPP produces a trace of the acyclic paths from the execution of a program and turns the sequence of acyclic paths into a context-free grammar. The outcome of WPP, program paths or traces - sequences of consecutively executed basic blocks, offer a clear window into program's dynamic behavior. However, in existence many frequently executed paths, WPP introduces a significant runtime overhead to compute the context free grammar.

7. CONCLUSIONS

Using dominator tree information for our dynamic code coverage with pre-instrumentation and on-demand instrumentation reduces the number of instrumentation points needed by 34-49% compared to all basic blocks instrumentation. Moreover, combining our dynamic code coverage with on-demand instrumentation using dominator tree information reduces by 42-79% the total number of basic blocks that must be instrumented. However, the most significant gains come from removing instrumentation code once a block is covered rather than from binary analysis algorithms to optimize instrumentation placement.

Our dynamic code coverage always outperforms *purecov* execution for the programs we tested. Even if all basic blocks are instrumented, for most deletion intervals our dynamic code coverage algorithm outperforms *purecov* execution. That is, dynamic deletion of instrumentation code reduces the overhead for programs with many infrequently executed (or even unexecuted) paths as well as for those with many frequently executed paths. Using a combination of dominator tree information and on-demand function instrumentation, we reduce not only the setup time but also the overhead during deletion intervals by eliminating the instrumentation code insertion for un-called procedures. By combining on-demand instrumentation and dominator tree information usage, we reduce the runtime overhead by 38-90% compared to *purecov* execution.

More importantly, for many applications, code coverage overhead is now tens of percent of the original execution time rather than several times the execution time. By reducing code coverage costs, it is now possible to consider including it as part of production code. Such an approach would greatly increase information about the execution of extremely infrequent error cases and could provide additional useful feedback to developers. Moreover, our fast code coverage techniques could be modified to sample the frequency of execution of basic blocks to provide additional information to a feedback-directed dynamic optimization system.

Our code coverage tools are fully automated and available for download from the Internet. The binary distribution of dyninst library and our dynamic code coverage tools can be obtained from http://www.dyninst.org/rel3.0/index.html.

Acknowledgements

This work was supported in part by NSF awards ASC-9703212 and EIA-0080206, and DOE Grants DE-FG02-93ER25176 and DE-FG02-01ER25510.

8. REFERENCES

[1] *C-Cover Code Coverage Analyzer for C/C++*, . http://www.bullseye.com/ccover.html, Bullseye Testing Technology.

[2] *Crashme Benchmark by MySQL Database System*, . http://www.mysql.com/information/crash-me.php.

[3] *Rational PureCoverage for Unix*, . http://www.rational.com/products/purecoverage/index.jtmpl, Rational Software Corporation.

[4] *SPEC newsletter*, . September 1995, http://www.specbench.org/osg/cpu95/CINT95.

[5] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*. 1986: Addison-Wesley Publishing Com.

[6] H. Agrawal, "Dominators, Super Blocks and Program Coverage," *POPL 94*, Portland, Oregon, pp. 25-34.

[7] B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *The International Journal of High Performance Computing Applications*, **14**, Winter 2000, pp. 317-329.

[8] C. Pavlopoulou and M. Young, "Residual Test Coverage Monitoring," *International Conference on Software Enginnering*. 1999, Los Angeles, CA, pp. 277-284.

[9] D. Bitton, D. J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems - A Systematic Approach," *Ninth International Conference on Very Large Data Bases*. Oct. 31-Nov. 2, Florence, Italy.

[10] J. Dean, C. A. Waldspurger, and W. E. Weihl, "Transparent, Low-Overhead Profiling on Modern Processors," *Workshop on Profile and Feedback-Directed Compilation*. October, Paris, France.

[11] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "ProfileMe: Hardware Support for Instruction-

Level Profiling on Out-of-Order Processors," *30th Symposium on Microarchitecture (Micro-30)*. December.

[12] J. Larus, "Whole Program Paths," *PLDI '99*. May 1999, Atlanta, GA.

[13] S. S. Muchnick, *Advanced Compiler Design and Implementation*. 1997: Morgan Kaufmann, Publishers Inc.

[14] R. L. Probert, "Optimal Insertion of Software Probes in Well-Delimited Programs," *IEEE Transactions on Software Engineering*, January, 1981, pp. 34-42.

[15] T Ball and J. R. Larus, "Efficient Path Profiling," *29th Annual IEEE/ACM International Symposium on Microarchitecture*, Paris, France, pp. 46-57.

[16] W. E. Weihl, *CPI: Continous Profiling Infrastructure, DIGITAL Forefront Magazine* 1997.

APPENDIX A

In this appendix, we present details of the instrumentation statistics for our code coverage algorithms for all programs we tested. We also present percentage reduction in number of instrumentation points needed when dominator information is used.

	Total Basic Blocks	Instrumented Basic Blocks			Reduction
		Leaf	Non-Leaf	Total	
tomcatv	53	27	4	31	41.5 %
compress	269	126	12	138	48.7 %
hydro2d	740	356	28	384	48.1 %
li	2,532	1,229	223	1,452	42.7 %
m88ksim	5,742	2,844	560	3,404	40.7 %
ijpeg	5,946	2,756	361	3,117	47.6 %
go	11,233	4,571	1,916	6,487	42.3 %
perl	13,181	6,695	1,432	8,127	38.3 %
vortex	19,047	8,137	4,442	12,579	34.0 %
gcc	68,458	28,915	13,866	42,781	37.5 %
postgres	45,140	23,011	3,353	26,364	41.6 %

Table 2. Pre-Instrumentation Points using All Basic Blocks vs. Dominator Tree Information

Table 2 presents the results for the code coverage tool with pre-instrumentation. The first column contains the programs we used for experiments. In the second column we give the total number of basic blocks in the executable. For instrumentation using dominator tree information, we divide the number of instrumented basic blocks in to two parts: Leaf node instrumentation and non-leaf node instrumentation count. The last column of the table gives the percentage reduction in the number of instrumentation points needed for our dynamic code coverage with pre-instrumentation using dominator tree information.

Table 2 shows that using dominator tree information reduced the number of instrumentation points needed by 34% to 49% compared to all basic blocks instrumentation.

Similarly, Table 3 presents the results for combining dominator tree information with on-demand instrumentation. In the third column, we give the total number of basic blocks in the executed functions for the workload we used. That is, the third column is the number of instrumentation points needed for all basic blocks instrumented for our code coverage algorithm with on-demand instrumentation.

Unlike Table 2, Table 3 contains two entries for *postgres* since the number of instrumented basic blocks changes based on the workload. Similarly, the last column of the table gives the percentage reduction in the number of instrumentation points needed compared with instrumenting all basic blocks in the set of functions executed.

	Total Basic Blocks	Basic Blocks in Executed Functions	Instrumented Basic Blocks			Reduction
			Leaf	Non-Leaf	Total	
tomcatv	53	53(100%)	27	4	31	41.5 %
compress	269	237 (88%)	108	12	120	49.4 %
hydro2d	740	692 (94%)	332	28	360	48.0 %
li	2,532	1,700 (67%)	808	176	984	42.1 %
m88ksim	5,742	1,984 (35%)	959	243	1,202	39.4 %
ijpeg	5,946	2,665 (45%)	1,202	188	1,390	47.8 %
go	11,233	10,981 (98%)	4,466	1,872	6,338	42.3 %
perl	13,181	8,942 (68%)	4,582	1,065	5,647	36.8 %
vortex	19,047	13,993 (73%)	5,789	3,497	9,286	33.6 %
gcc	68,458	32,779 (48%)	13,998	6,678	20,676	36.9 %
postgres -wisconsin	45,140	16,417 (36%)	8,587	1,254	9,841	40.1 %
postgres -crashme	45,140	20,860 (46%)	10,775	1,601	12,376	40.7 %

Table 3. On-demand Instrumentation Points using All Basic Blocks vs. Dominator Tree Information

APPENDIX B

In this appendix, we present execution time slowdown ratios for the rest of the programs we tested. The format of the graphs presented in this appendix is exactly same with the ones presented in Section 5.3.

Figures 10-19 present the execution time slowdown ratios for the rest of the programs. All of the programs have source code line coverage percentage graphs similar to Figure 6. That is, the source code line coverage percentage steeply increases at the beginning of their execution and stays steady during the rest of the execution.

For all programs except *tomcatv*, *hydro2d* and *ijpeg*, the best execution time occurs when instrumented by our code coverage tool using on-demand instrumentation and dominator tree information. For *tomcatv*, *hydro2d* and *ijpeg*, however, the best execution times for our dynamic code coverage tool with on-demand instrumentation and dominator tree information usage differ from the best execution time among all by less than 1%. We suspect this difference is caused by the slight variations in the workload while we were conducting our experiments.

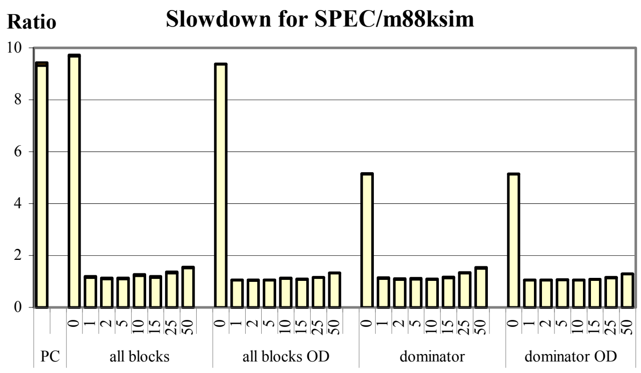

Figure 10. Execution Time Slowdown Ratios for m88ksim

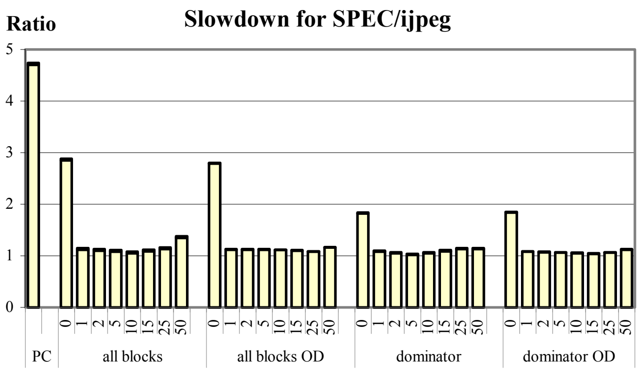

Figure 14. Execution Time Slowdown Ratios for ijpeg

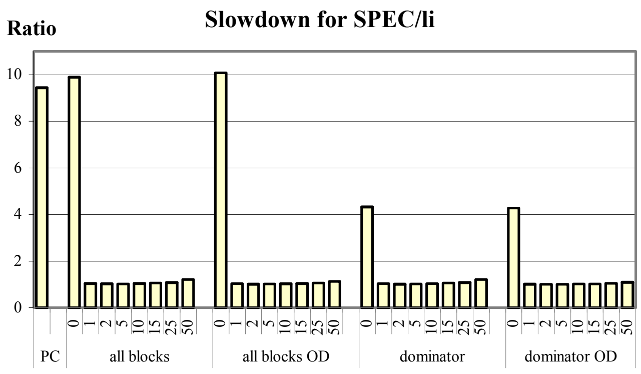

Figure 11. Execution Time Slowdown Ratios for li

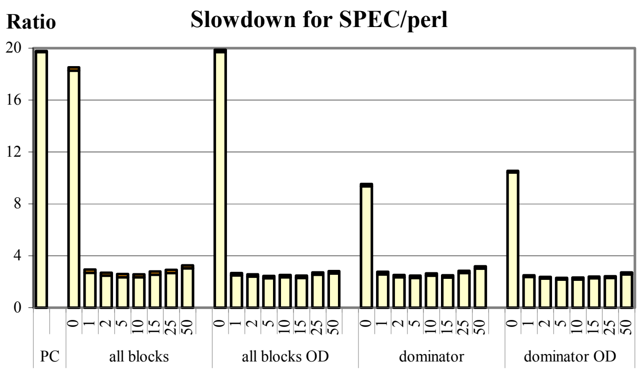

Figure 15. Execution Time Slowdown Ratios for perl

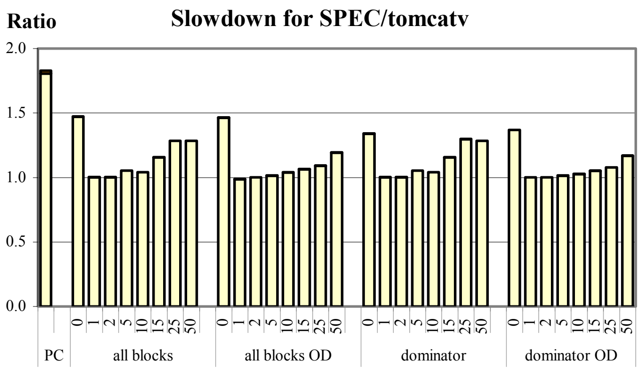

Figure 12. Execution Time Slowdown Ratios for tomcatv

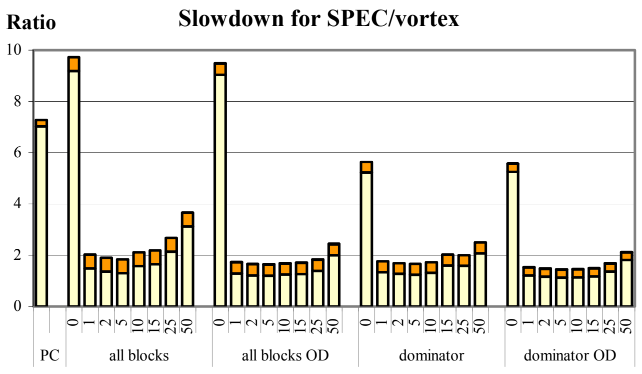

Figure 16. Execution Time Slowdown Ratios for vortex

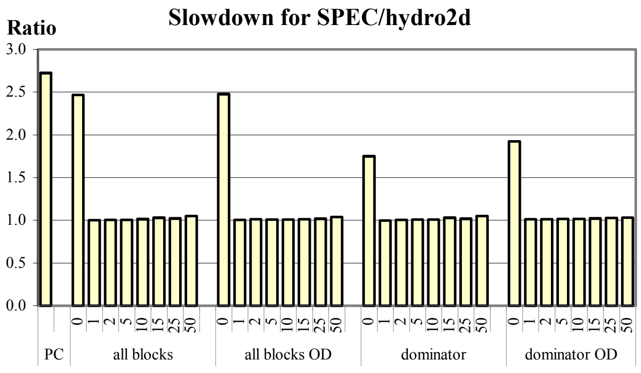

Figure 13. Execution Time Slowdown Ratios for hydro2d

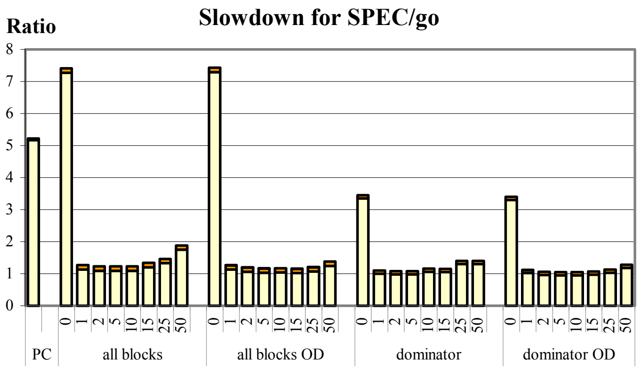

Figure 17. Execution Time Slowdown Ratios for go

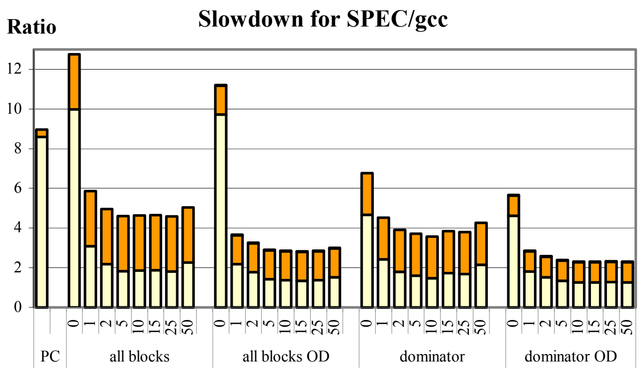

Figure 18. Execution Time Slowdown Ratios for gcc

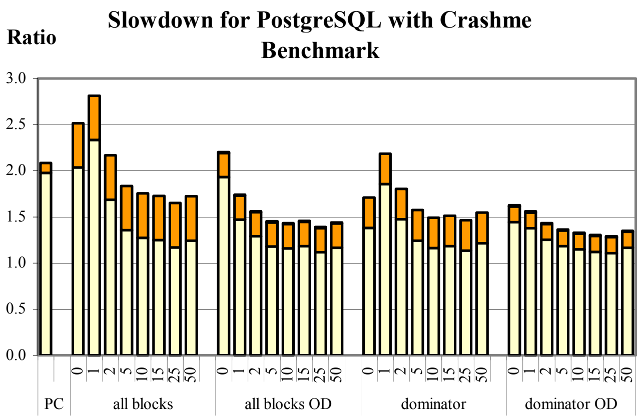

Figure 19. Execution Time Slowdown for PostgreSQL with Crashme Benchmark

Appendix C

Based on the data presented in Section 5.3, we suspected that the overhead for our dynamic code coverage system would be bursty throughout a program's execution. To investigate this hypothesis, we added meta-instrumentation code to each basic block to record the number of times instrumentation code is executed. This appendix presents the instrumentation code execution frequencies for *PostgreSQL* using the *Wisconsin* and *crashme* benchmark queries when instrumentation code is deleted every second.

Figure 20 shows the distribution of instrumentation code executions using bars and source code line coverage percentage using lines for *PostgreSQL* running the *Wisconsin* benchmark. For the bars, y-axis gives instrumentation code execution frequencies ($\log_{10}$ scale). The y-axis for the continuous curve shows the source code line coverage percentage of the program. Figure 20 shows that whenever the source code line coverage percentage increases, there are executions of instrumentation code. Figure 20 also shows that during the intervals that source code line coverage percentage remains steady, there is no instrumentation code executed. When the program enters a new phase, instrumentation code is executed for the first time. Shortly after the instrumentation code is executed, it is deleted and never executed during the rest of that phase.

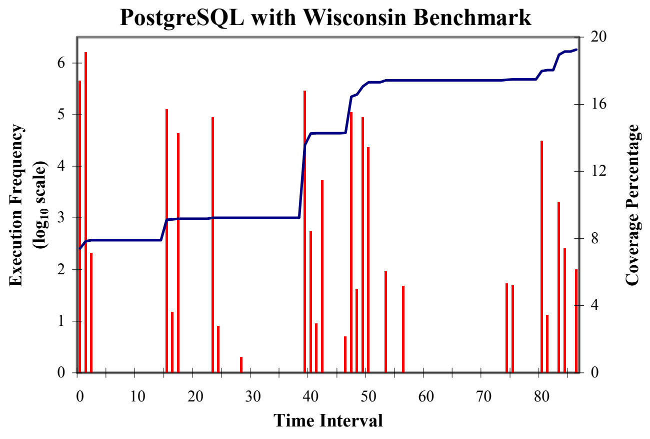

Figure 20. Instrumentation Code Execution Frequency and Source Code Line Coverage Percentage for PostgreSQL with Wisconsin Benchmark

Figure 21 shows the distribution of instrumentation code executions and source code line coverage percentage for *PostgreSQL* with *crashme* benchmark. Like Figure 20, Figure 21 shows that the instrumentation code executions occur at the intervals where the source code line coverage percentage increases and during the intervals coverage percentage remains steady there are no instrumentation code executions.

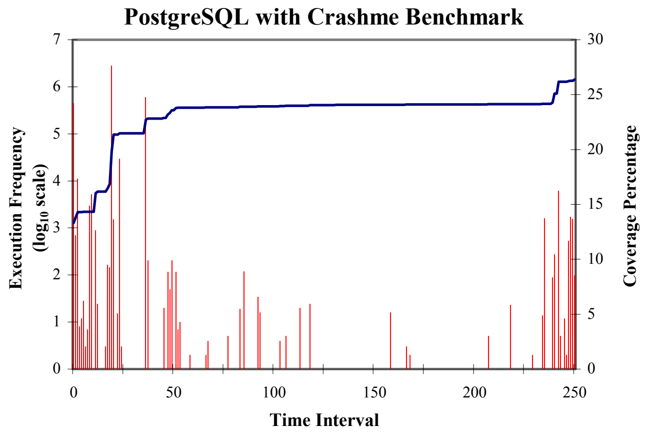

Figure 21. Instrumentation Code Execution Frequency and Source Code Line Coverage Percentage for PostgreSQL with Crashme Benchmark

Unlike Figure 20, Figure 21 shows occasional bursts of instrumentation code execution during the middle section of program execution (from 75 to 225 seconds). However these bursts do not affect source code line coverage percentage significantly.