Efficient Online Computation of Statement Coverage

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 statement coverage is the problem of identifying the statements of a program that execute in one or more runs of a program. The traditional approach for statement 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 statement coverage measurement. 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 statement coverage tool. Our tool is fully automated and available for download from the Internet.

1. INTRODUCTION

Evaluation of statement coverage is the problem of identifying the statements of a program that execute in one or more runs of a program. Developers and testers use statement coverage to ensure that all or substantially all statements in a program have been executed at least once during the testing process. Measuring statement coverage is important for testing and validating code during both development and porting to new platforms. Traditionally statement coverage measurement tools have been built using static code instrumentation. During program compilation or linking, these tools insert instrumentation code into the source code, object code or binary executable. 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(SPEC, 1995) suite runs almost 20 times slower under a commercial statement coverage tool, *purecov*(Rational-PureCover, 2002), when the instrumentation code for statement 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 statement coverage tools. Our goal in this paper is to show that deletion of instrumentation code used for statement coverage produces coverage results faster for long running programs. We believe that by making statement coverage testing 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 moni-

toring(Pavlopoulou and Young, 1999) more efficient. Our fast statement 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 statement 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 statement coverage tools conservatively insert all possibly needed instrumentation code even though it may never be executed in future runs. However, such conservative instrumentation not only increases the code size, but more importantly, the overhead of preprocessing and instrumentation is significant for large programs containing a few frequently executed paths. For example, due to instrumentation overhead, the *cc1* benchmark from SPEC95 suite runs 12.1% slower when all functions are conservatively instrumented compared to when only called functions are instrumented. Thus, using incremental instrumentation of functions during program execution, we eliminate the instrumentation time and code growth for uncalled functions.

Even though we explore the use of more sophisticated binary analysis techniques to reduce the number of places instrumentation code needs to be inserted, our goal is not to find the optimal number of instrumentation points. Unlike previous research to find the optimal number of instrumentation points, we instead try to minimize the sum of the analysis and instrumentation overhead, thus reduce the runtime coverage testing overhead. In this paper, in addition to our published work(Tikir and Hollingsworth, 2002), we also compare our dynamic statement coverage approach to one of the research on finding the optimal number of instrumentation points for coverage testing. We chose to compare our approach to Agrawal's(Agrawal, 1994) approach in terms of the reduction in the number of instrumentation points needed and execution performance of the applications.

In this paper, we also explore the use of saturation counters to record limited information about the frequency of execution of statements. However, saturation counters can have a significant affect on the overall overhead due to the fact that leaving instrumentation code in executable increases the coverage overhead. If the instrumentation code is inserted along frequently executed paths, then re-executions of instrumentation code will increase coverage overhead.

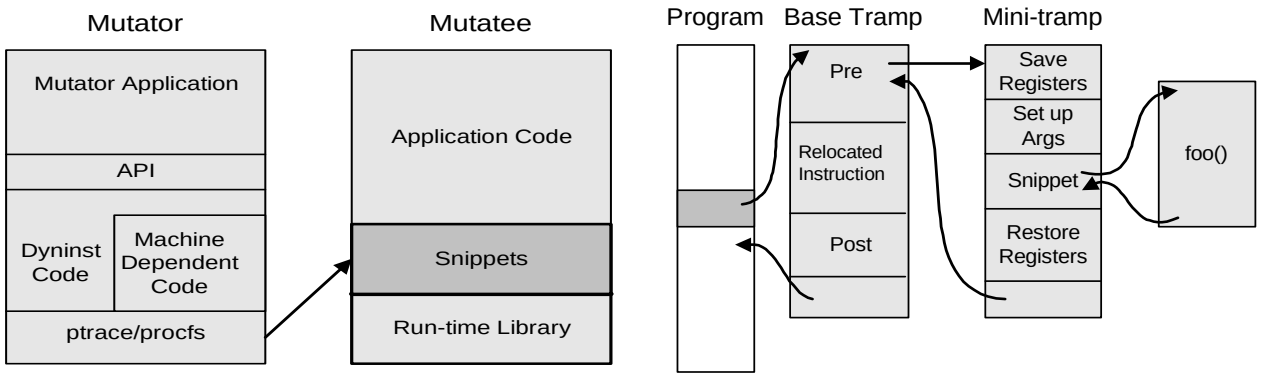
Besides statement coverage testing, our techniques can also be easily applied to other coverage testing problems, such as edge coverage testing and definition-use coverage testing. In this paper, we also explore how we apply our techniques to edge coverage testing. We chose to apply our techniques to implement and evaluate statement coverage testing, as we believe it is the most widely used coverage testing method during the development and testing phases.

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 statement 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 statement coverage, Section 5 presents the results of a series of experiments conducted to evaluate our approach, Section 6 compares our approach with an algorithm that finds the optimal number of instrumentation points for coverage testing. Section 7 presents the results of experiments conducted using different saturation counters for dynamic code deletion. Section 8 explains how we apply our techniques to edge coverage testing. Section 9 explains the graphical user interface of our statement coverage testing tools. Section 10 presents the related work and Section 11 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(Hollingsworth et al., 1997; Buck and Hollingsworth, 2000; Hollingsworth and Buck, 2000). Implementations of dyninst are currently available for Alpha, Sparc, Power, Mips, IA-64 and x86 architectures.

Figure 1(a) 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 1(b)).



(a) Structure of DyninstAPI **(b) Instrumentation Code Insertion**

Figure 1. Dyninst Instrumentation Process

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.

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 statement 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 (Aho et al., 1986). We then create the *dominator tree* and *control dependence regions* of a control flow graph using the algorithm in (Lengauer and Tarjan, 1979) and (Muchnick, 1997). 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 statement coverage tool, however, we need finer grained instrumentation at the level of basic blocks. We added the capability to the library to create arbitrary instrumentation points and instrument them. Arbitrary instrumentation points in the dyninst library are created for a given address. 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 statement coverage tools. We use properties of the immediate dominator tree and control dependence regions of a control flow graph for instrumentation point selection.

3.1 Leaf Node Instrumentation

The control flow graph (CFG) of a function is the graph of basic blocks describing possible orders of the execution of the statements in the function. A basic block *d* of a CFG 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*. A *dominator tree*(Aho et al., 1986) 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 *m* of a CFG is control dependent on basic block *n* of the CFG, if basic block *n* can directly affect whether basic block *m* is executed or not. The *control dependence regions*(Aho et al., 1986) of a CFG partition basic blocks such that all basic blocks in a region execute under certain control conditions. Control dependence regions of a CFG are created using dominator or post-dominator relation.

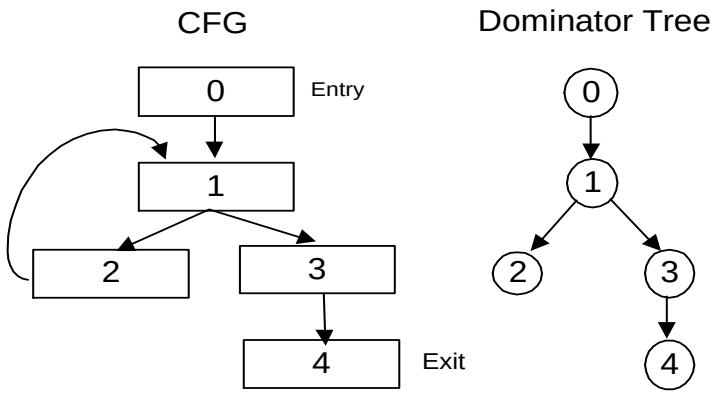


Figure 2. A simple CFG and Its Dominator Information

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. Similarly, if a basic block in control dependence region is executed all other basic block(s) in the same control dependence region is (are) also executed. Figure 2 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 statement 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*. Assume flow of control includes such an edge from basic block *n* to basic block *m*, denoted (*n*,*m*), where *n* does not dominate *m*. That is, *m* is not a descendant of *n* in the dominator tree. If the basic block *n* is one of the leaf level basic blocks in the dominator tree, leaf level instrumentation is sufficient. However if the basic block *n* is an internal node, the execution path may not include any leaf level basic blocks of the sub-tree rooted at *n*. That is, leaf level instrumentation is not sufficient since any of the leaf level basic blocks of the sub-tree rooted at *n* may not be reached, thus coverage information can not be produced for basic blocks *n* and some of its ancestors.

The reason that each edge in a CFG whose source does not dominate its target and whose source is one of the internal nodes of the dominator tree needs to be handled specially is that each such edge identifies a control dependence region in the CFG. That is, for each edge of CFG, from basic block *n* to basic block *m* where *n* does not dominate *m*, a new control dependence region is created. The corresponding control dependence region includes the basic blocks along the path in dominator tree from *n* to the lowest common ancestor, *p*, of *m* and *n* in the dominator tree, excluding *p*.

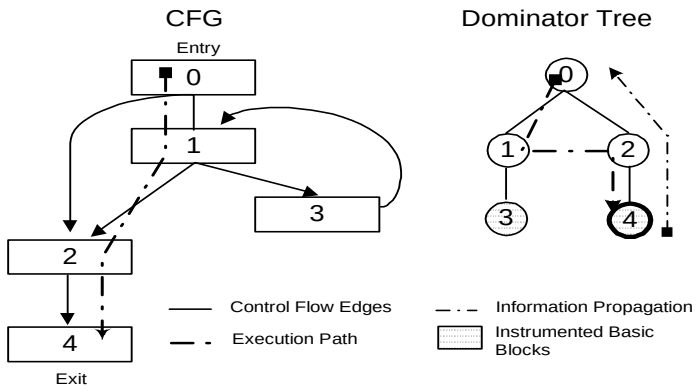


Figure 3. Why leaf level instrumentation is not sufficient

Figure 3 gives an example for an execution path that includes an edge originating from internal node in the dominator tree and whose source does not dominate its target. For this control flow graph, leaf level basic block instrumentation is not sufficient for correct statement 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 internal basic block *n* if *n* has at least one outgoing edge to a basic block *m* that *n* does not dominate. That is, we instrument internal basic block *n* if it is in a different control dependence region than any of its children or ancestor in the dominator tree. In this example basic block 1 has an outgoing edge to 2 and 1 does not dominate 2. Basic block 1 is in different control dependence region than basic blocks 0 and 3. The control dependence regions for this example

are as {0,2,4}, {1}, and {3}. Thus, we choose basic block 1 to be instrumented besides basic blocks 3 and 4.

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, unlike (Probert, 1982; Agrawal, 1994), our goal is not to find the optimal number of instrumentation points, but to minimize the sum of the analysis and instrumentation overhead. Although we use Langauer-Tarjan (Lengauer and Tarjan, 1979) algorithm that is linear in number of edges in a control flow graph, our experiments have shown that the additional post-dominator tree construction and necessary graph processing to correctly use both dominator and post-dominator information is an expensive computation relative to the limited benefit we can expect. That is, additional post-dominator information increased our binary analysis time without a significant reduction in instrumentation overhead. Thus we chose to use only dominator tree information. In Section 6, we compare using both dominator and post dominator information in terms of the reduction in the number of instrumentation points and performance improvement compared to using only dominator information.

4. STATEMENT COVERAGE ALGORITHM

We implemented two slightly different versions of our dynamic statement coverage algorithm: statement coverage with pre-instrumentation and statement 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 statement 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 statement 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 thus 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 implement and improves the execution time of the program being tested significantly. The deletion interval is a tunable parameter to our tool.

At program termination we record the results of statement coverage. For our statement 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 statement 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 more than offsets 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*), commercial statement coverage tool that uses static code editing. We measured the execution time of programs instrumented by our dynamic statement coverage tools including the setup time for control flow graph generation, dominator tree construction, and instrumentation code insertion. We tested statement coverage for *PostgreSQL*, an object-relational DBMS, using the *Wisconsin*(Bitton et al., 1983) and *crashme(MySQL-Crashme, 2002)* benchmarks, all C programs (*go, m88ksim, gcc, compress, li, ijpeg, perl, vortex*) from the *SPEC95*(SPEC, 1995) benchmarks and two of the Fortran programs (*tomcatv, hydro2d*) from *SPEC95* benchmarks, using the standard reference input data. 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 enabled the debug option to gather line information from the debug records in the executable. 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 statement coverage with pre-instrumentation, and statement 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 4 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[1]. For each program, there are four bars. The bars labeled *all blocks* correspond to the statement coverage tools with all basic blocks instrumentation and the ones labeled *dominator* indicate use of dominator tree information. The *OD* suffix indicates our statement coverage algorithms with on-demand instrumentation.

Figure 4 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 is used.

[1] Details about the statement coverage instrumentation statistics for all programs are given in Appendix A.

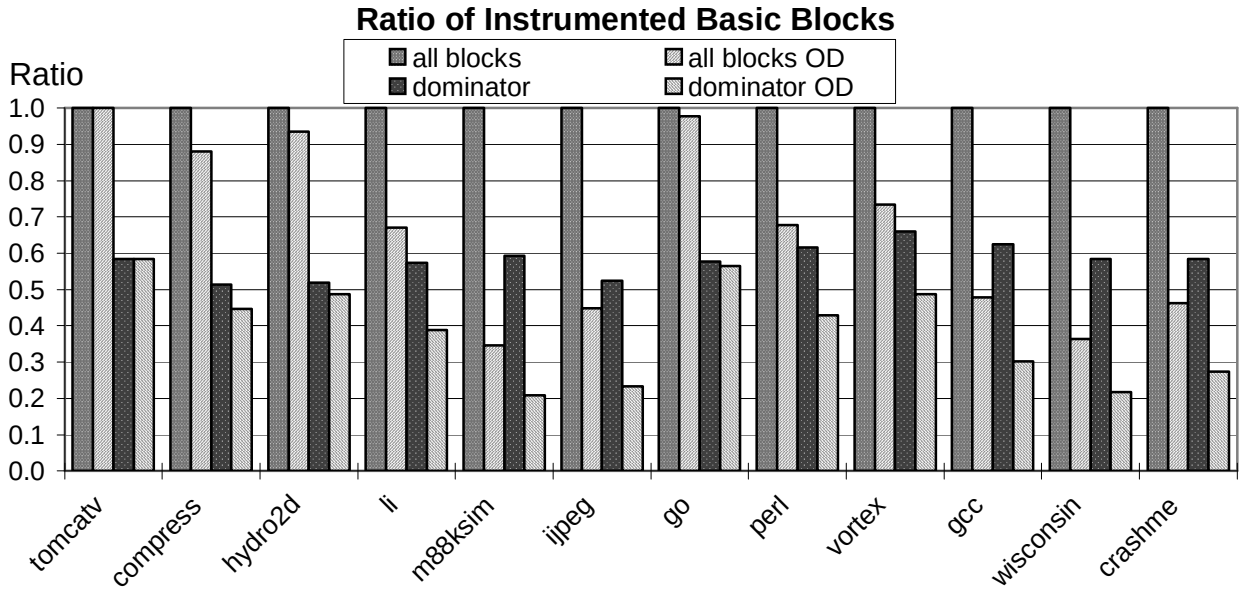


Figure 4. Ratio of Instrumented Basic Blocks to the Total Number of Basic Blocks

Figure 4 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 4 also shows that using on-demand instrumentation, our statement 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 among all versions of our statement coverage algorithm. 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 (tomcatv is fully covered in terms of basic blocks*)*. Overall we were able to eliminate instrumentation from 42% to 79% of the basic blocks in the executables.

5.2 Execution Times

In this section, we present the impact of dynamic code deletion and dominator information usage in the execution times of various applications. However, to show how rapidly they reach certain levels of coverage, we first present the source code line coverage percentage curves for the applications. To measure the source code line coverage percentage of applications, we stopped the running process every 1 second and calculated the percentage of source lines executed. We chose to present the results for *compress* and *PostgreSQL* with *Wisconsin* benchmark queries in detail as they exhibit interesting perspectives in terms of coverage for our approach. The results for the rest of the applications are presented in Appendix B.

Figure 5 shows the source line coverage percentage versus time for *compress* from SPEC95 benchmark suite. The coverage percentage in Figure 5 steeply increases to 76% in the first 18% of the execution time and stays at this level through the rest of the execution.

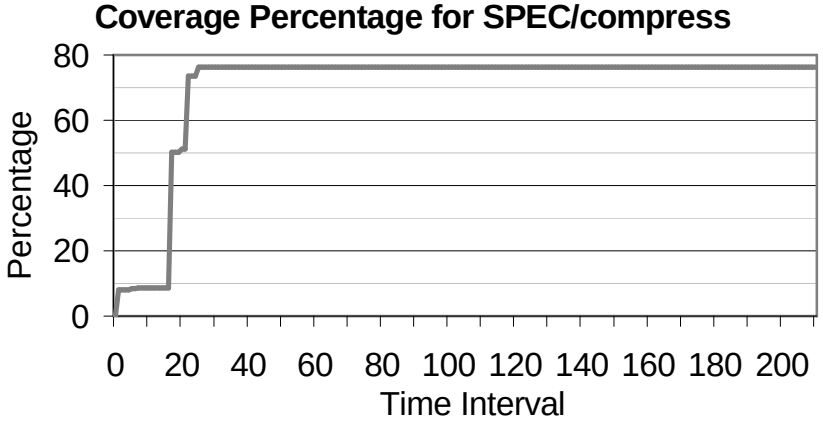


Figure 5. Source Line Coverage Percentage for compress

Figure 6 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 6.

Unlike Figure 5, 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 6 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.

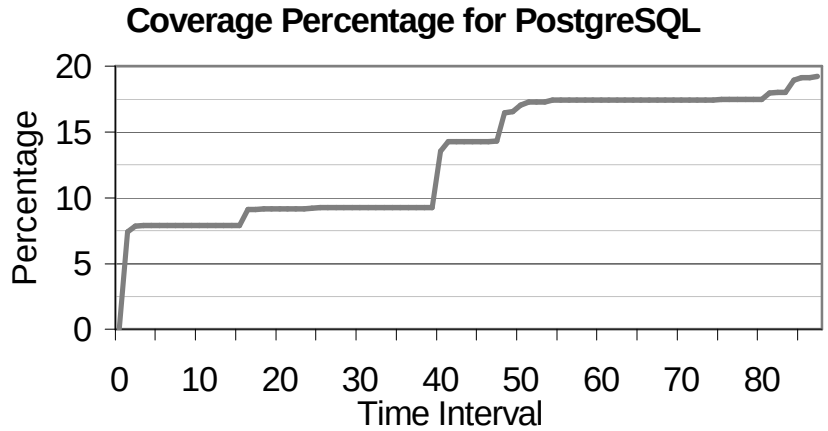


Figure 6. Source Line Coverage Percentage for PostgreSQL with Wisconsin Benchmark

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

Figure 7 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 statement coverage algorithms. Catego-

ries labeled *dominator* use dominator tree information for instrumentation where the ones labeled *all blocks* indicate our dynamic statement 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.

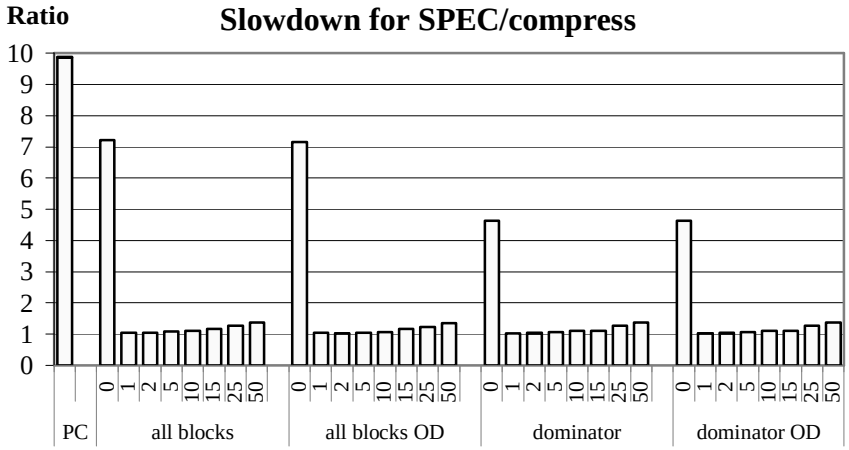


Figure 7. Execution Time Slowdown Ratios for compress

Figure 7 shows that all of our statement 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 4). Likewise on-demand instrumentation provides little benefit. Figure 7 also shows that the execution times increase slightly for larger deletion intervals for all of our statement coverage tools due to re-execution of some instrumentation code in the first couple deletion intervals.

Figure 7 also shows that without dynamic code deletion, our dynamic statement 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 *purecov* sometimes inserts additional unnecessary instrumentation code around the pseudo-instruction that implements integer division of the SPARC.

Figure 7 also shows that, for *compress*, our dynamic statement 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 statement 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 statement coverage tool introduces only a 0.3% run time overhead compared to the original execution of *compress*.

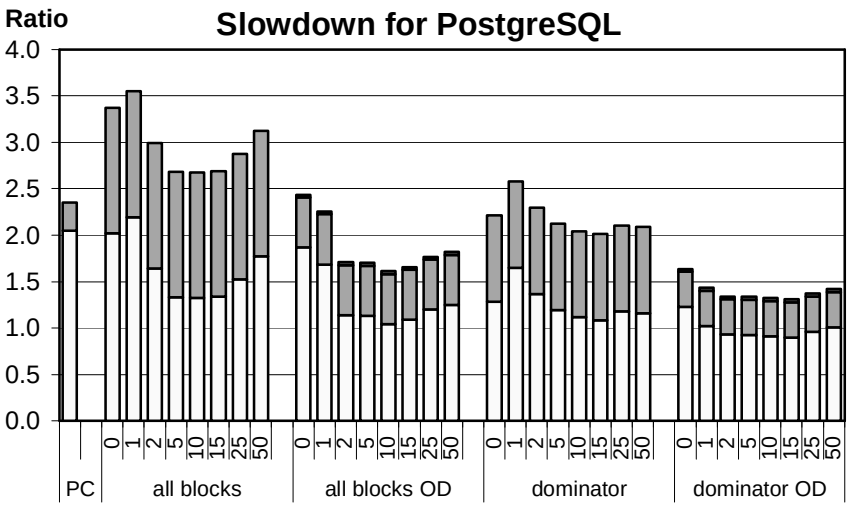


Figure 8. Execution Time Slowdown Ratios for PostgreSQL with Wisconsin Benchmark

Figure 8 presents the execution time slowdown ratios, with respect to original execution time, of *PostgreSQL* for the *Wisconsin* benchmark instrumented by purecov and our dynamic statement 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 statement coverage tools with on-demand instrumentation, the 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 7, they were so insignificant for *compress* that they were not visible).

Figure 8 shows that setup times for our statement 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 statement coverage tools with on-demand instrumentation is not significant since it only requires inserting breakpoints at the beginning of the functions. Figure 8 also shows that control flow graph generation and instrumentation of functions for our dynamic statement coverage tools with on-demand instrumentation takes significantly less time compared to our tools with pre-instrumentation.

Figure 8 shows that *purecov* outperforms our statement coverage tool with pre-instrumentation and all basic blocks instrumentation. 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 statement coverage algorithms, *purecov* does not incur overhead due to 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 8 shows that our statement 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 8 also shows that our statement coverage tools with on-demand instrumentation outperform our statement 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 instrumen-

tation code incrementally that eliminates the checks that would otherwise be done in previous deletion intervals.

Like in Figure 7, the results in Figure 8 indicate that using dynamic code deletion produces faster statement coverage results. Unlike Figure 7, every-second deletion performs slightly worse than no dynamic code deletion for pre-instrumentation case, since the more instrumentation code must be checked.

Figure 8 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.

5.2.1 Instrumentation Code Execution Frequency

Based on the data presented in Section 5.2, we suspected that the overhead for our dynamic statement 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 section presents the instrumentation code execution frequencies for *PostgreSQL* using the *Wisconsin* benchmark queries when all basic blocks are instrumented and instrumentation code is deleted every second.

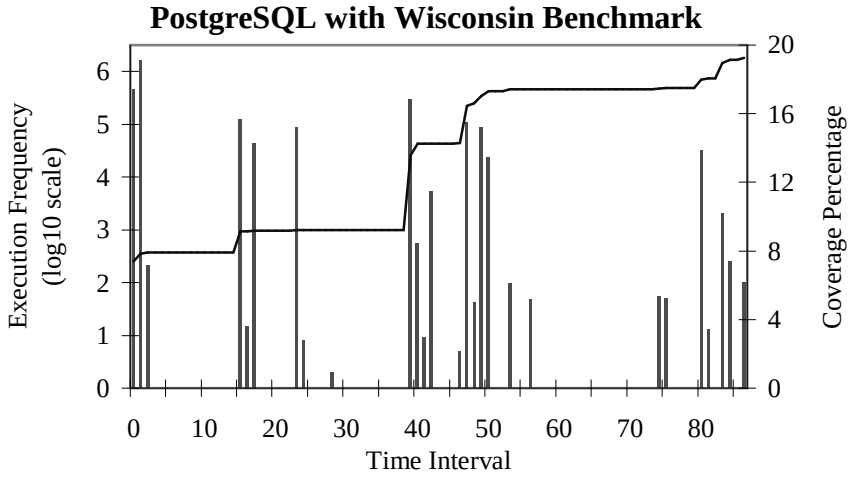


Figure 9. Instrumentation Code Execution Frequency and Source Code Line Coverage Percentage for PostgreSQL with Wisconsin Benchmark

Figure 9 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, the left y-axis gives instrumentation code execution frequencies ($\log_{10}$ scale). The right y-axis for the continuous curve shows the source code line coverage percentage of the program. Figure 9 shows that whenever the source code line coverage percentage increases, there are executions of instrumentation code.

Figure 9 also shows that during the intervals that source code line coverage percentage remains steady, there is no instrumentation code executed (Due to size and resolution of the graph, slight in-

creases in coverage percentage in Figure 9 is not noticeable even though there are executions of instrumentation code at the same interval. Thus, Figure 9 may mislead as if there are instrumentation code executions when the coverage curve remains steady.) 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.

5.3 Overall Slowdown

We also calculated the slowdown ratio with respect to the original execution time for programs instrumented using purecov and our dynamic statement coverage tool. We took the results for 2-second deletion interval for our dynamic statement 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. We chose 2-second deletion as representative of our techniques due two reasons. First, using 2-second intervals did not increase the runtime overhead significantly, unlike using 1-second interval. Secondly, 2 seconds is short enough to prevent many re-executions of a given instrumentation point.

	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 statement coverage tools with on-demand and pre- instrumentation, and *purecov*.

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 statement 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 statement 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 statement coverage tool with on-demand instrumentation slows down the execution only a factor of 1.002 to 2.6 using dominator tree information. Our statement coverage tools with on-demand instrumentation frequently outperform the ones with pre-instrumentation.

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. Hence, our statement 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. COMPARING WITH AN OPTIMAL INSTRUMENTATION PLACEMENT APPROACH

In addition to demonstrating the effectiveness of our dynamic statement coverage approach, we believe it is also necessary to compare our approach to previous research on finding the optimal number of instrumentation points for coverage testing. Recall that even though our approach reduces the number of instrumentation points needed for coverage testing, our goal is not to find the optimal number of instrumentation points. We instead try to minimize the sum of the analysis and instrumentation overhead such that the runtime overhead of statement coverage testing is reduced.

For comparison of our approach to one of the previous researches on finding the optimal number of instrumentation points for coverage testing, we chose the work described in Agrawal(Agrawal, 1994). Unlike our approach where we only use dominator tree information, Agrawal uses both dominator and post-dominator tree information of a CFG to find the optimal number of instrumentation points for the CFG to gather statement coverage information.

To find the optimal number of instrumentation points in a CFG, Agrawal first constructs both dominator and post-dominator tree of the CFG. Next, the algorithm merges both dominator trees as the union of both graphs, called *basic block dominator graph*, and calculates the strongly connected components, called *super blocks*, of the union graph. Then, a directed acyclic graph (DAG) is constructed from the super nodes, called *super block dominator graph*. Lastly, leaf nodes in the super block dominator graph and internal nodes whose coverage does not imply the coverage of its children are selected for instrumentation. Please refer to (Agrawal, 1994) for more details on Agrawal's approach on finding the optimal number of instrumentation points.

To compare our approach to Agrawal's approach, we implemented instrumentation point selection algorithms described in (Agrawal, 1994) and integrated it into our statement coverage tool to select the instrumentation points in the applications. In addition, we modified our statement coverage tool to be able propagate coverage information from the execution of the instrumented basic blocks to the uninstrumented basic blocks. For a fair comparison, we ran experiments using Agrawal's approach for both pre-instrumentation and on-demand instrumentation and recorded the number of instrumentation points needed and performance improvement in execution times for all applications we tested.

Table 2 presents the number of instrumentation points needed for the programs we tested for both our approach and Agrawal's. In the second column we give the total number of basic blocks in the applications. The group of next three columns presents the results when all functions are pre-instrumented at start of the applications, where group of last three columns presents the results when on-demand instrumentation is used. In each group, the column labeled *Dom* gives the number of instrumentation points needed when our approach is used, where the column labeled *Dom&Post-Dom* gives the number of instrumentation points needed when we use Agrawal's algorithms. Lastly, the last column in each group gives the reduction percentage in the number of instrumentation points needed for Agrawal's approach compared to our approach.

Table 2 shows that for all applications, for both using pre-instrumentation and on-demand instrumentation, Agrawal's approach requires fewer instrumentation points compared to our approach. Table 2 also shows that for pre-instrumentation, using only dominator information reduces the number of instrumentations needed by 34.0-48.7% (Average is 42.0%) compared to the total number of basic blocks in the ap-

plications. However, adding post-dominator information only reduces the number of instrumentation points by 3.2-16.3% (Average is 8.7%) more. Similarly, for on-demand instrumentation, using only dominator information reduces the number of instrumentation points by 41.5-79.1% (Average is 61.5%) and additional use of post-dominator information only reduces the number of instrumentation points by 3.2-14.4% more (Average is 7.9%). That is, Table 2 shows that even though Agrawal's approach is more effective in reducing the number of instrumentation points needed compared to ours, using additional post-dominator information is not as effective as using only dominator information in terms of reduction percentages.

	Total Basic Blocks	**Using Pre-Instrumentation**			**Using On-Demand Instrumentation**		
		Dom	**Dom& Post-Dom**	**Reduc-tion %**	**Dom**	**Dom& Post-Dom**	**Reduc-tion %**
tomcatv	53	31	30	3.2	31	30	3.2
compress	269	138	116	16.3	120	103	14.4
hydro2d	740	384	345	10.1	360	326	9.6
li	2,532	1,452	1,265	12.9	984	859	12.7
m88ksim	5,742	3,404	3,171	6.8	1,202	1,127	6.2
ijpeg	5,946	3,117	2,748	11.9	1,390	1,242	10.7
go	11,233	6,487	6,121	5.6	6,338	5,989	5.5
perl	13,181	8,127	7,744	4.7	5,647	5,390	4.5
vortex	19,047	12,579	11,682	7.1	9,286	8,743	5.9
postgres(Wisconsin)	45,140	26,364	23,605	10.5	9,841	8,879	9.8
postgres(crashme)	45,140	26,364	23,605	10.5	12,376	11,313	8.6
gcc	68,458	42,781	40,842	4.5	20,676	19,940	3.6

Table 2. Comparison of the number of instrumentation points needed when only dominator tree information is used to when both dominator tree and post-dominator tree information is used for both using pre-instrumentation and on-demand instrumentation.

Even though reduction in the number of instrumentation points is important for statement coverage testing, what really matters is the runtime overhead due to the statement coverage testing. Thus we also recorded the execution times of the applications we tested for both our approach and Agrawal's approach to investigate the tradeoffs between finding the optimal number of instrumentation points and its impact on the runtime overhead of statement coverage.

Figure 10 compares Agrawal's approach to our approach in terms of execution times when on-demand instrumentation is used (We do not present the results with pre-instrumentation since they are similar to the results with on-demand instrumentation). For each application, two bars give the slowdown for the application when Agrawal's approach is used compared to using our approach (Negative values indicate speedup). The first bar gives the slowdown for the application when instrumentation code deletion is enabled where the second bar gives the slowdown for the application.

Figure 10 shows that, when instrumentation code is not deleted, Agrawal's approach outperforms our approach by 1-14% for all applications except *gcc* and *tomcatv*. This is due to the fact that Agrawal's approach reduces the number of instrumentation points needed compared to our approach, thus results in less instrumentation that stays in the application throughout the execution. However for *gcc* and *tomcatv*, our approach outperforms Agrawal's approach even though latter reduces the number of instrumentation points around 3% for both applications. This is mainly due to the fact that the saved time due to less instrumentation does not overcome the additional binary analysis time spent by Agrawal's approach.

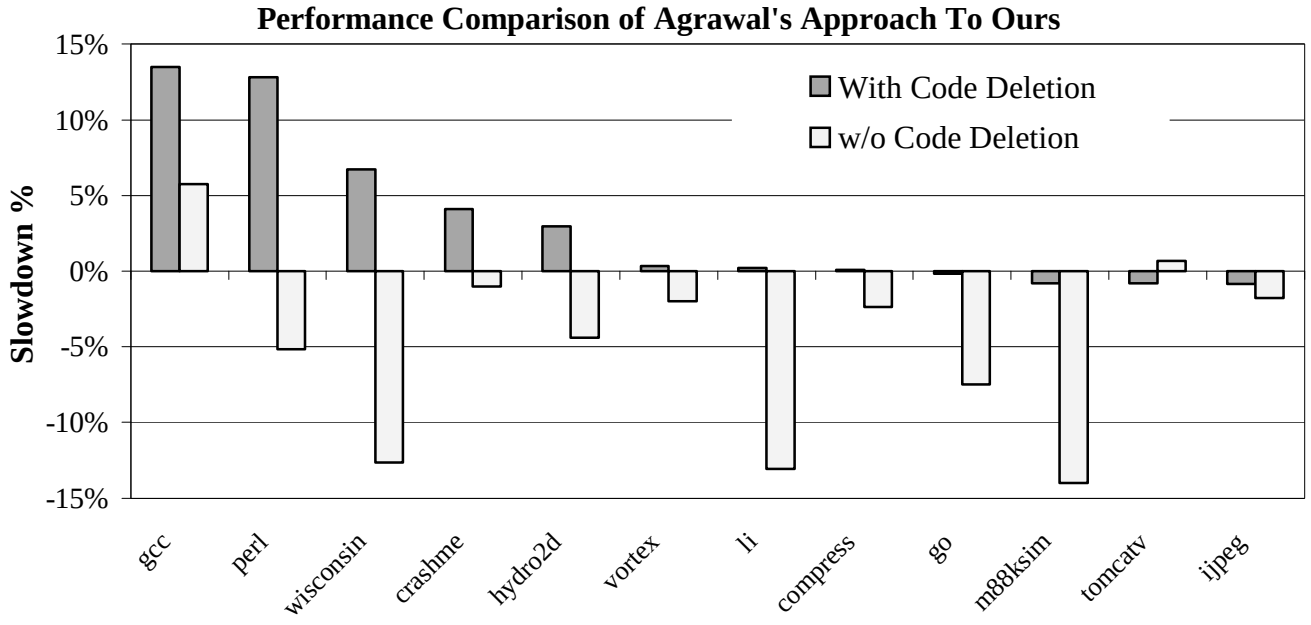


Figure 10. Comparison of Agrawal's approach to our approach for all applications in terms of their execution performance when on-demand instrumentation is used.

The results presented in Section 5.2 shows that the effectiveness of our approach comes mainly from dynamic instrumentation code deletion. Recall that the applications run 38-90% faster when instrumentation code is deleted compared to keeping all instrumentation in place throughout the execution. Figure 10 shows that when dynamic instrumentation code deletion is enabled, our approach outperforms Agrawal's approach for almost all of the applications and the improvement goes as high as 13.5% compared to Agrawal's approach. For applications *m88ksim*, *tomcatv* and *ijpeg* our approach performs slightly worse compared to Agrawal's approach. More importantly, Figure 10 shows that for the applications with high number of basic blocks (*gcc*, *perl* and *postgres*), our approach performs significantly better compared to Agrawal's approach.

7. USING SATURATION COUNTERS

In this section, we present the results of our experiments in which we used saturation counters. Saturation counters are counters with a fixed number of bits and they do not reset after they overflow but instead store the maximum value they can count. We used saturation counters to give some coarse information about the execution frequency of statements besides identifying if a statement is executed. Using saturation counters, the deletion of the instrumentation code is delayed until the counter reaches saturation.

In the experiments presented in Section 5, we used a Boolean flag to mark whether an instrumentation code is executed or not. In these experiments, the instrumentation code is deleted during the first deletion interval after the flag is set and the flag is never read again during the rest of the deletion intervals.

To clearly present the impact of different saturation counters and deletion interval values on the execution times of the applications, we divided the total execution time of an application into four segments; 1) Time spent to execute original and instrumentation code, 2) CFG creation and instrumentation time, 3) Time spent during deletion intervals excluding time spent to remove the instrumentation code, 4) Time spent to remove instrumentation code.

Figure 11 shows the results of our experiments using saturation counters for the SPEC/cc1 application using our coverage tool with dominator tree information and on-demand instrumentation. The graphs

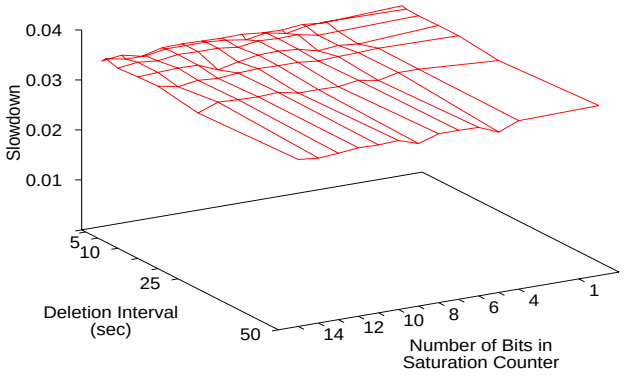
in Figure 11(a-d) present the execution time slowdown ratios due to different parts of the program execution. Figure 11(e) shows the total execution slowdown ratios. For each graph in Figure 11, the x-axis shows different deletion interval values. We conducted our experiments with deletion interval values ranging from 0 to 50, where 0 indicates no dynamic code deletion. The y-axis shows the number of bits used for saturation counters that ranges from 1 to 15. A value of 1 for the number of bits in saturation counters indicates that the instrumentation code is deleted during the first deletion interval after its first execution.

Figure 11(a) shows the overhead of creating the CFGs for the called-functions, inserting instrumentation code and dyninst initialization. Figure 11 (a) shows that, for all runs, the slowdown ratios are almost the same. This is due to the fact that the CFG creation and instrumentation code insertion are the same regardless of the number of bits in the saturation counter. However, for some runs there are minor variations. We observed that the dyninst library allocated different chunks of memory that is required by the instrumentation code among different runs. These variations occur due to the differences in the internal dynamic memory allocator in the dyninst library. The dyninst library sometimes performs memory compaction if the de-allocated memory can be re-used. Since our approach deletes instrumentation code during deletion intervals, memory compaction is often triggered. Moreover, the memory compaction pattern changes with the dynamic state of the allocated and de-allocated pages by the dyninst library in the application.

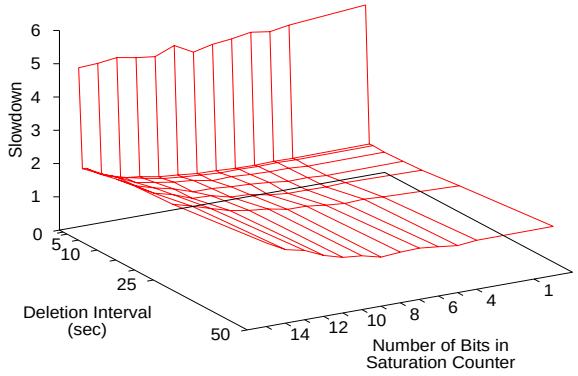
Figure 11(b) shows the slowdown ratio for the execution of the original instructions and the instrumentation code inserted in the executable. Figure 11(b) shows that when there is no dynamic code deletion, the overhead of executing instrumentation code is significant. It also shows that a deletion interval of one second introduces the lowest overhead. Figure 11(b) also shows that the overhead of executing instrumentation code increases when the deletion interval value increases as mentioned in Section 5.2. Figure 11(b) shows that using more bits in the saturation counters, the overhead of instrumentation code execution slightly increases. The increase in the overhead is due to the fact that the number of instrumentation points with repeated execution increases as the instrumentation code deletion is delayed until the saturation counters overflow.

Figure 11(c) shows the slowdown ratio for the overhead of the iterations in the deletion intervals due to reading the values of the saturation counters for all instrumentation code that is still in the executable. That is, for each instrumentation snippet that has not been deleted yet, we read its saturation counter and check whether it has saturated. Figure 11(c) shows that this overhead decreases when the deletion interval increases. It is due to the fact that for the higher values of deletion interval, we perform fewer reads and checks. Figure 11(c) also shows that this overhead significantly increases when the number of bits in saturation counters increases for the more frequent deletion intervals. Thus, most of the checks performed on the values of saturation counters are not profitable.

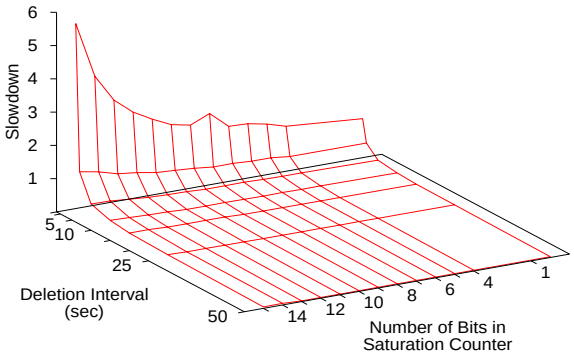
Figure 11(d) shows the slowdown ratios for the overhead of dynamic code deletions. This overhead is the time spent only to remove instrumentation code. It does not include time spent to check whether the instrumentation code has executed enough times. Figure 11(d) shows that time spent for instrumentation code deletion significantly decreases when the number of bits used for saturation counters increases. This is due to the fact that, using more bits in the saturation counters, results in the instrumentation code executing more times before it is deleted. Thus, some of the instrumentation code is never deleted since their counters never saturate.



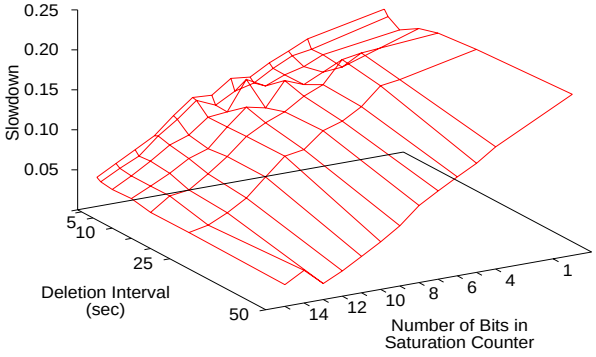
(a) CFG Creation and Instrumentation Time



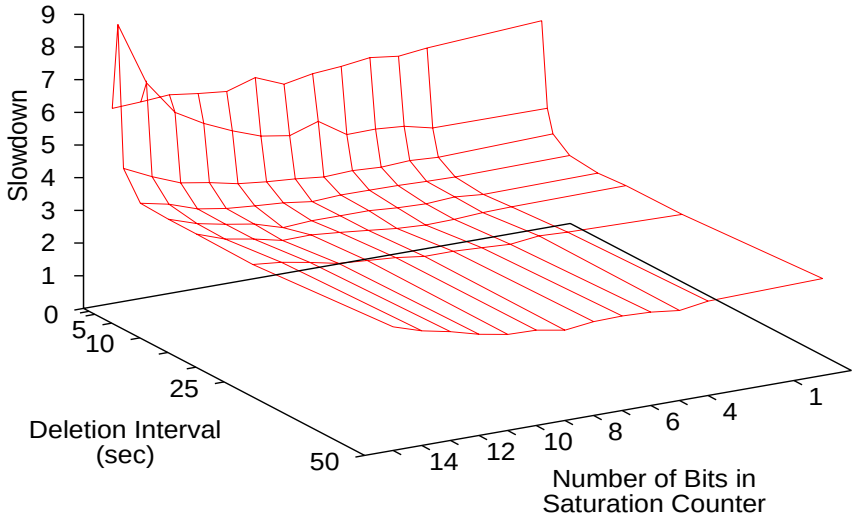
(b) Execution Time and Instrumentation Code Execution Time



(c) Deletion Interval Iteration Time



(d) Instrumentation Code Deletion Time



(e) Total Execution Time for Different Saturation Values

Figure 11. Execution Times for Different Saturation Values and Deletion Intervals for SPEC/cc1

Figure 11(e) shows the total execution slowdown ratio for different saturation values and deletion interval values. This graph shows that the execution slowdown ratios change the same by the deletion interval for all saturation values. Figure 11(e) shows that if the instrumentation code is deleted frequently, increasing number of bits in saturation counters introduces significant overhead.

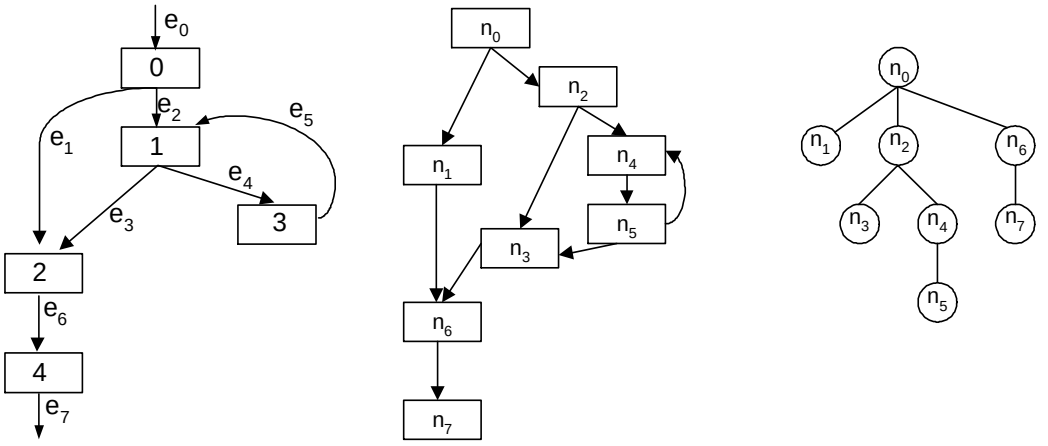
More importantly, Figure 11(e) also shows that for higher deletion interval values, the number of bits in saturation counters has a slight impact on the execution time slowdown ratios. Thus, using our coverage tools, it is possible to give some coarse information about the instrumentation code execution frequencies without introducing a significant overhead. In summary, at the recommended deletion interval (50 sec), adding saturation counters is nearly free.

8. APPLICATION OF OUR TECHNIQUES TO EDGE COVERAGE TESTING

Besides statement coverage testing, our techniques can also be easily applied to other coverage testing problems, such as branch coverage testing, edge coverage testing, or definition-use coverage testing. In this section we explain how our instrumentation technique and dynamic code deletion can be applied to edge coverage testing. In this section, we will only briefly explain the transformations needed to evaluate edge coverage testing using our techniques. We will not present any experimental results since we did not implement the tools for edge coverage testing. This is due to the fact that currently the dyninst library is not capable of efficiently inserting code along edges in a CFG.

To apply our instrumentation point selection technique to edge coverage testing, we first transform each CFG to another flow graph, G'. In the new graph, the nodes represent the original edges in the CFG and the edges represent possible execution ordering constraints over the edges of the CFG.

To transform the original CFG to the new graph, G', first we add two new edges to the original CFG to represent entry to the CFG and exit from the CFG. That is, we add an incoming edge to the entry basic block and an outgoing edge from the exit basic block of the original CFG. Second, we create a node, n_i, in G' for each edge, e_i, in the original CFG. Third, if the execution of an edge, e_j, in the original CFG may immediately follow the execution of edge e_i, we create an edge in G' from node representing e_i to the node e_j. To correctly capture the execution ordering constraints over the edges of the original CFG, we process each basic block in turn. For each basic block in the original CFG, for each possible pair of incoming edge, e_i, and outgoing edge, e_j, we insert an edge in G' from node representing e_i to node e_j.



(a) CFG **(b) Transformed Graph, G'** **(c) Dominator Tree of G'**

Figure 12. Transformation of a CFG for Edge Coverage Testing

Figure 12 shows the transformation of the CFG in Section 3.2 for edge coverage testing. Figure 12(a) shows the CFG with two additional edges e_0 and e_7. Figure 12(b) shows the transformed flow graph where there is a node for each edge in Figure 12(a) and the edges represent ordering constraints over the edges of Figure 12(a).

After the transformation of each CFG to the new flow graph, we apply our instrumentation point selection techniques to the new graph. That is, we create the dominator tree for the new graph and use its

control dependence regions as explained in Section 3. Figure 12(c) shows the dominator tree for the new graph shown in Figure 12(b).

9. GRAPHICAL USER INTERFACE

Figure 13 shows a snapshot from the graphical user interface (GUI) of our statement coverage tools. Our GUI contains panels to run the application for statement coverage testing and to get online information about the statement coverage of the application. Using our GUI, it is possible to navigate the source files and the functions in the executable. Moreover, our GUI also displays statistics on the number of lines covered for the files and functions. It also displays statistics on the instrumentation code deletions.

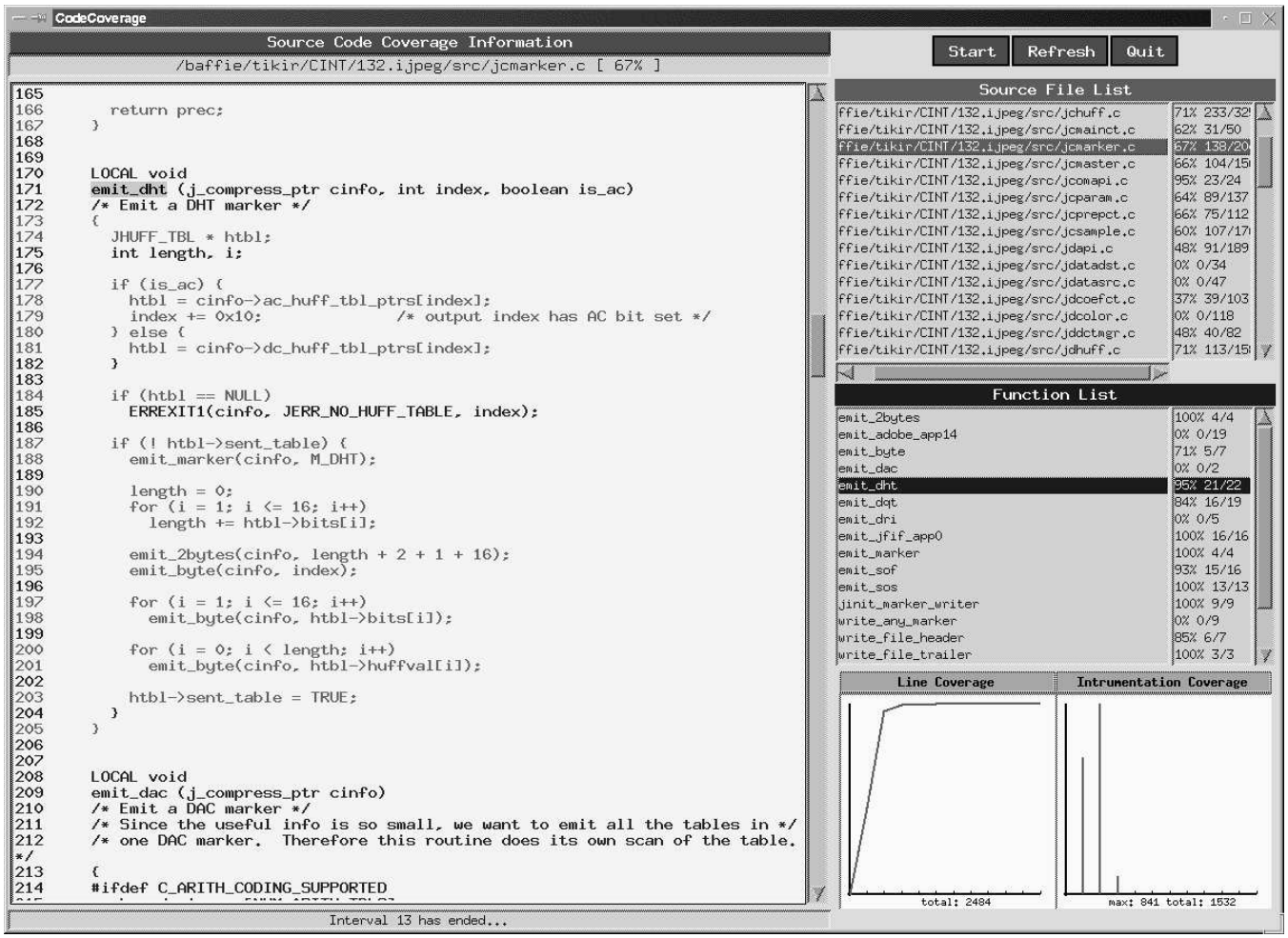


Figure 13. Graphical User Interface of Our Statement Coverage Tools

The top list on the right section of Figure 13 is used to navigate the source files executable program. Next to each source file entry, our GUI displays the number of lines that have currently executed and total number of lines in that file. These numbers are dynamically updated every time the dynamic code deletion occurs and statement coverage information is collected. Similarly, the bottom list on the right section of Figure 13 is used to navigate the functions in the selected source file. For each function, the interface also displays the number of executed source lines and total number of source lines in that function. These numbers are also updated every time the dynamic code deletion occurs.

The large sub window on the left section of Figure 13, displays the source file that is being navigated. The lines that have executed are highlighted. Every time the application is stopped and dynamic code de-

letion occurs, the display is updated to highlight the new source code lines that executed. The two small text boxes above and below the source file sub window display the summary about the source file, such as percentage of lines covered, and the execution status of the application, respectively.

The two plots on the bottom right of Figure 13 display statistics about the statement coverage execution and instrumentation deletion versus execution time. The graph on the left displays the number of distinct source code lines that have executed at least once. Similarly, the graph on the right displays the number of basic blocks de-instrumented for each deletion interval. These plots are updated when dynamic code deletion occurs. We believe these plots provide additional insight to the users about the application being tested. For example a programmer can tell if any new source lines are being executed, and how the source code line coverage curve changes during program execution.

10. RELATED WORK

The two systems most closely related to our dynamic statement coverage tool are the commercial statement coverage tools, PureCoverage(Rational-PureCover, 2002) and C-Cover(Bullseye-CCover, 2002). To locate untested areas of code, PureCoverage uses Object Code Insertion 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(Bullseye-CCover, 2002). C-Cover also displays condition/decision coverage and function coverage results. Unlike our dynamic statement 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 statement 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(Pavlopoulou and Young, 1999) 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 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(Agrawal, 1994) 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. Agrawal uses properties of dominator trees to provide programmers guidance about how to create test cases to provide significant statement coverage for each case.

Path Profiles(Ball and Larus, 1996) can be used to compute the statement 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 statement 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 (Ball and Larus, 1996) 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 statement coverage tools presented in this paper includes the analysis, setup and instrumentation time.

Digital Continuous Profiling Infrastructure (DCPI)(Dean et al., 1997; Weihl, 1997) 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.

Whole Program Paths (WPP)(Larus, 1999) can also be used to extract statement 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.

11. CONCLUSIONS

Using dominator tree information for our dynamic statement 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 statement 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 statement coverage always outperforms *purecov* execution for the programs we tested. Even if all basic blocks are instrumented, for most deletion intervals our dynamic statement 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.

Our dynamic statement coverage approach outperforms previous approach, namely Agrawal(Agrawal, 1994), for most of the applications we tested. This advantage is most significant for applications with many basic blocks where the additional binary analysis time to compute the optimal number of instrumentation points increases the runtime overhead without a significant reduction in instrumentation overhead.

More importantly, for many applications, statement coverage overhead is reduced to tens of percent of the original execution time rather than several times the execution time. By reducing statement 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 statement 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 statement coverage tools are fully automated and available for download from the Internet. The binary distribution of dyninst library and our dynamic statement coverage tools can be obtained from http://www.dyninst.org.

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.

REFERENCES

Agrawal, H., 1994. Dominators, Super Blocks and Program Coverage. In: Proc. of the 21th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 16-19, Portland, OR. ACM Press, pp. 25-34.

Aho, A., Sethi, R., Ullman, J., 1986. Compilers: Principles, Techniques and Tools. Addison-Wesley.

Ball, T., Larus, J. R., 1996. Efficient Path Profiling. In: Proc. of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, December 2-4, Paris, France. IEEE Computer Society Press, pp. 46-57.

Bitton, D., DeWitt, D. J., Turbyfill, C., 1983. Benchmarking Database Systems - A Systematic Approach. In: Proc. of the 9th International Conference on Very Large Data Bases, October 31-November 2, Florence, Italy. Morgan Kaufman, pp. 8-19.

Buck, B. R., Hollingsworth, J. K., 2000. An API for Runtime Code Patching. The International Journal of High Performance Computing Applications. **14**(4), pp. 317-329.

Bullseye Testing Tech., 2002. C-Cover Code Coverage Analyzer for C/C++, http://www.bullseye.com/coverage.html.

Dean, J., Hicks, J. E., Waldspurger, C. A., Weihl, W. E., Chrysos, G., 1997. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In: Proc. of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, December 1-3, Research Triangle Park, NC. IEEE Computer Society Press, pp. 292-302.

Hollingsworth, J. K., Buck, B. R., 2000. DyninstAPI Programmer's Guide, http://www.paradyn.org/html/manuals.html. University of Maryland.

Hollingsworth, J. K., Miller, B. P., Gongalves, M. J. R., Naim, O., Xu, Z., Zheng, L., 1997. MDL: A Language And Compiler For Dynamic Program Instrumentation. In: Proc. of the International Conference on Parallel Architectures and Compilation Techniques, November 11-15, San Francisco, CA. IEEE Computer Society Press, pp. 201.

Larus, J., 1999. Whole Program Paths. In: Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, May 1-4, Atlanta, GA. ACM Press, pp. 259-269.

Lengauer, T., Tarjan, R. E., 1979. A Fast Algorithm for Finding Dominators in a Flowgraph. ACM Transactions on Programming Languages and Systems. **1**(1), pp. 121-141.

Muchnick, S. S., 1997. Advanced Compiler Design and Implementation. Morgan Kaufmann.

MySQL Database System, 2002. Crashme Benchmark by MySQL Database System, http://www.mysql.com/information/crash-me.php.

Pavlopoulou, C., Young, M., 1999. Residual Test Coverage Monitoring. In: Proc. of the 21st International Conference on Software Enginnering, May 16-22, Los Angeles, CA. IEEE Computer Society Press, pp. 277-284.

Probert, R. L., 1982. Optimal Insertion of Software Probes in Well-Delimited Programs. IEEE Transactions on Software Engineering. **SE-8**(1), pp. 34-42.

Rational Software Corp., 2002. Rational PureCoverage for Unix, http://www.rational.com/products/purecoverage/index.jtmpl.

Standard Performance Evaluation Corp., 1995. SPEC CPU95 Benchmarks, http://www.spec.org/osg/cpu95/.

Tikir, M. M., Hollingsworth, J. K., 2002. Efficient Instrumentation for Code Coverage Testing. In: Proc. of the International Symposium on Software Testing and Analysis, July 22-24, Rome, Italy. ACM Press, pp. 86-96.

Weihl, W. E., 1997. CPI: Continous Profiling Infrastructure, Winter 1997.

APPENDIX A

In this appendix, we present details of the instrumentation statistics for our statement 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 3. Pre-Instrumentation Points using All Basic Blocks vs. Dominator Information

Table 3 presents the results for the statement 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 statement coverage with pre-instrumentation using dominator tree information. Table 3 shows that using dominator tree information reduced the number of instrumentation points needed by 34% to 49% compared to all basic blocks instrumentation.

	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 4. On-demand Instrumentation Points using All Basic Blocks vs. Dominator Information

Similarly, Table 4 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.

Unlike Table 3, Table 4 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.

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 here is exactly same with the ones presented in experiments section. Figure 14-Figure 23 present the execution time slowdown ratios for the rest of the programs. The source code line coverage percentage for these programs also 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 statement coverage tool using on-demand instrumentation and dominator tree information. For *tomcatv*, *hydro2d* and *ijpeg*, however, the best execution times for our dynamic statement 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.

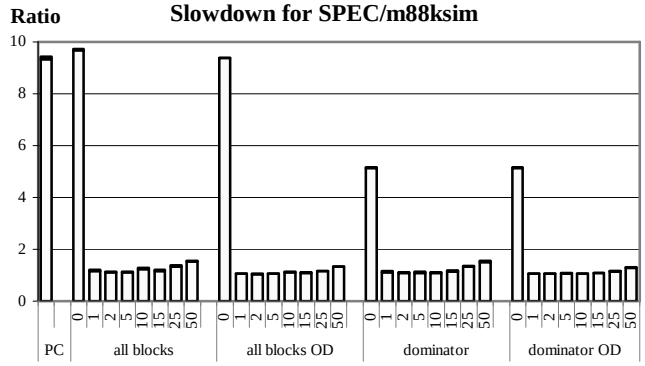


Figure 14. Execution Time Slowdown Ratios for m88ksim

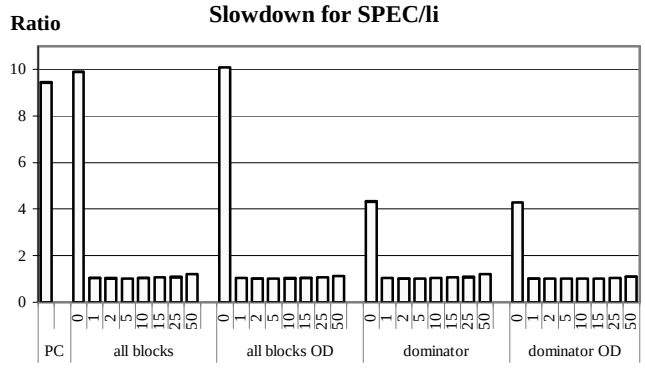


Figure 15. Execution Time Slowdown Ratios for li

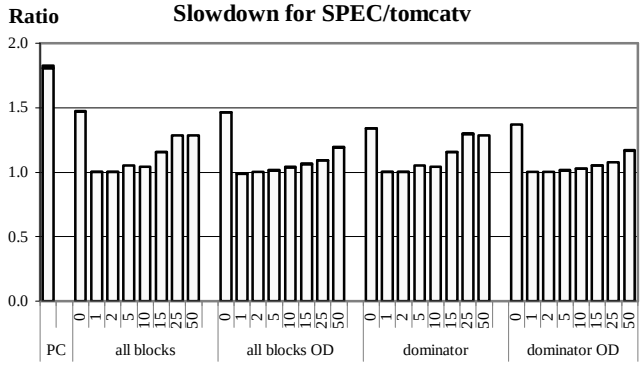


Figure 16. Execution Time Slowdown Ratios for tomcatv

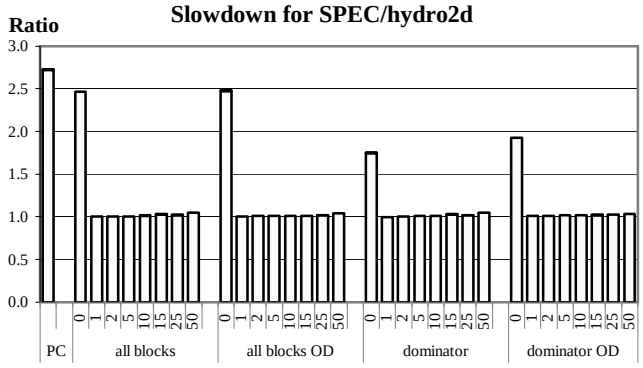


Figure 17. Execution Time Slowdown Ratios for hydro2d

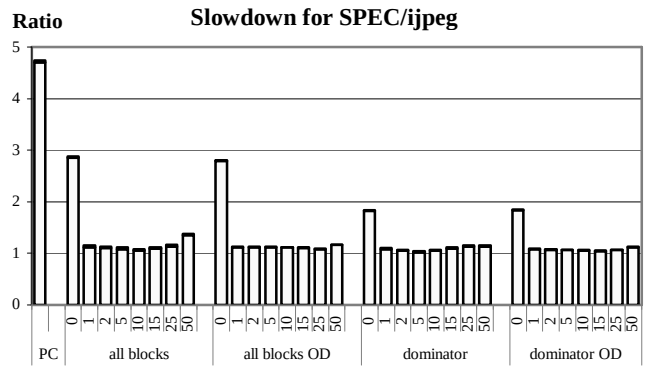


Figure 18. Execution Time Slowdown Ratios for ijpeg

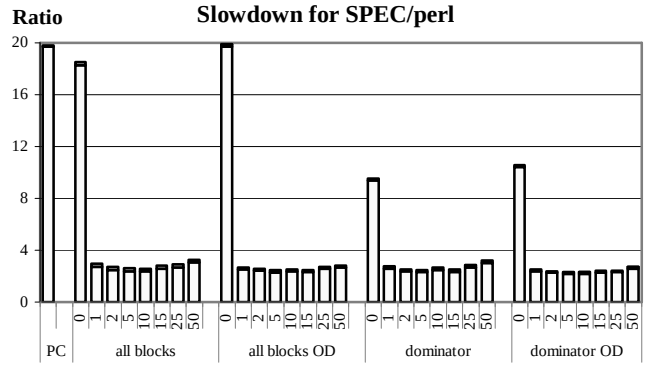


Figure 19. Execution Time Slowdown Ratios for perl

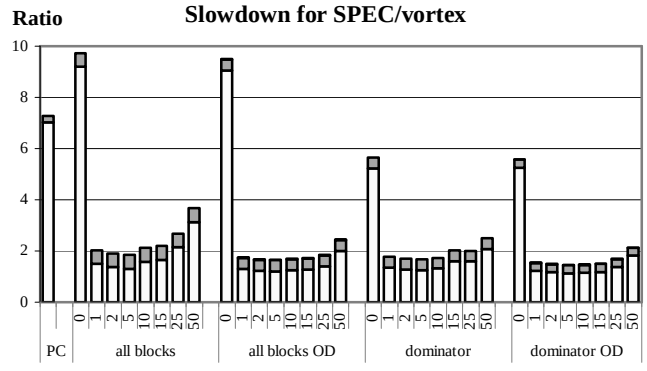


Figure 20. Execution Time Slowdown Ratios for vortex

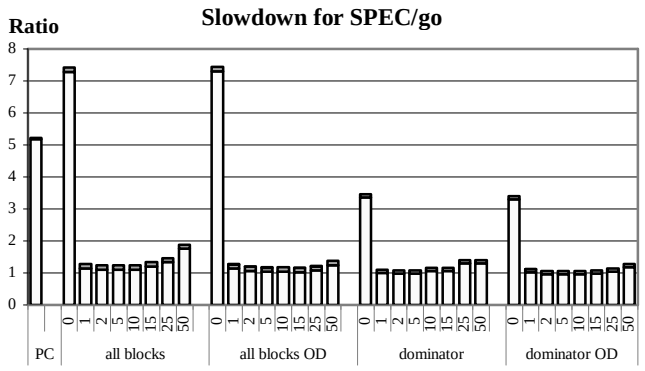


Figure 21. Execution Time Slowdown Ratios for go

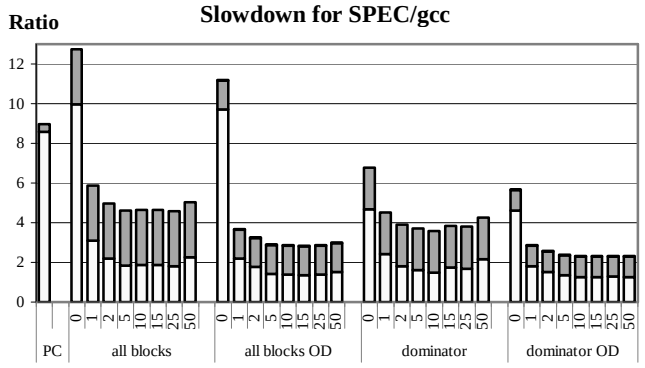


Figure 22. Execution Time Slowdown Ratios for gcc

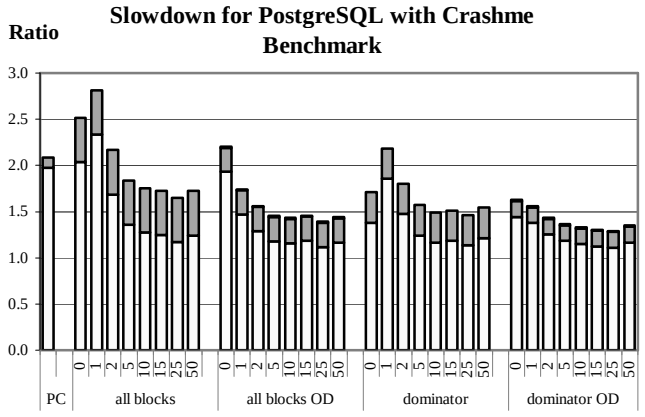


Figure 23. Execution Time Slowdown for PostgreSQL with Crashme Benchmark