RESIDUAL COVERAGE MONITORING FOR COMPILED PROGRAMS

by

PRASANNA VINJAMURI

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School at the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 2000

“Residual Coverage Monitoring For Compiled Programs,” a thesis prepared by Prasanna Vinjamuri in partial fulfillment of the requirements for a Master of Science degree in the Department of Computer and Information Science. This Thesis has been approved and accepted by:

[illegible]

Dr. Michal Young, Chair of the Thesis Committee

26 May 2000

Date

Committee in Charge:	Dr. Michal Young, Chair
	Dr. Allen Malony

Accepted by:

[illegible]

Dean of Graduate School

An Abstract of the Thesis of

Prasanna Vinjamuri for the degree of Master of Science

in the Department of Computer and Information Science to be taken June 2000

Title: RESIDUAL COVERAGE MONITORING FOR COMPILED PROGRAMS

Approved: [illegible]

Dr. Michal Young

Monitoring the residues of test coverage in actual use or during beta testing activities can validate any assumptions of their impact on quality. But this is unlikely to be accepted by customers unless the performance impact is small. A previous work presented a proof-of-concept tool written for Java programs to demonstrate that the performance overhead of running instrumented executables is low if instrumentation is removed from already executed basic blocks. It is however not clear whether and to what extent this result depends on the high run-time overhead of the Java platform. It is also an open question whether residual coverage monitoring would have a significant advantage relative to complete coverage monitoring if the complete monitoring were implemented using the best known monitoring techniques. This work answers the above questions by using EEL, an tool for instrumenting compiled languages, a quick profiler, QPT2 and standard SPEC CINT95 benchmarks.

CURRICULUM VITA

NAME OF AUTHOR: Prasanna Vinjamuri

DATE OF BIRTH: May 26, 1972

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
University of Cincinnati
Indian Institute of Technology, Madras

DEGREES AWARDED:

Master of Science in Computer and Information Science, 2000, University of Oregon
Master of Science in Mechanical Engineering, 1996, University of Cincinnati
Bachelor of Technology in Mechanical Engineering, 1993, Indian Institute of Technology

AREAS OF SPECIAL INTEREST:

Computer Aided Design, Engineering and Manufacturing

PROFESSIONAL EXPERIENCE:

Product Development Engineer, Structural Dynamics Research Corporation, Cincinnati, 1995-2000

AWARDS AND HONORS:

University Graduate Scholarship, University of Cincinnati, 1993-1995

ACKNOWLEDGEMENTS

The author would like to thank Prof. Michal Young for his valuable help and guidance throughout this effort. The author also thanks Prof. Allen Malony, member on my thesis committee, for giving valuable feedback for this work. The author wishes to express gratitude to the WARTS program members from the University of Wisconson for access to the EEL and QPT2 software distributions. The author acknowledges the help provided by the Standard Performance Evaluation Corporation in access to the SPEC benchmark source. The author also wishes to acknowledge Prof. Santhosh Pande and Srikanth Malladi at the University of Cincinnati for help in system access at the university. Finally the author also wishes to acknowledge Mr. Saad Assad for challenging him to complete the MS program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Beta testing activities are typically used by software companies to predict program behavior at customer sites before the software is released to the field. However feedback from these activities is often limited to problem reports and does not include the kind of feedback about the program dynamic behavior that can help the developers and testers validate and refine the models they have relied on during quality assurance. An example of such feedback is structural coverage information. Structural coverage information about the program can not only offer insight into the effectiveness of the beta test that was conducted but also can help expose areas in the software that were not covered by unit and system tests.

Software is often released without 100% structural coverage, which means that testers are explicitly or implicitly assuming that the remaining test obligations or residues are either unfeasible or occur so rarely that they have negligible impact on quality. Monitoring the residue of test coverage in actual use or beta testing activities can validate these assumptions. But this is unlikely to be accepted by customers unless its performance impact is small. The goal of residual test coverage monitoring is to provide deployed software with monitoring of adjustable level and focus to address varying performance requirements with user control to address concerns of security, and confidentiality [15]. The main challenge to this approach, performance (the other being security and confidentiality concerns), is addressed by reducing execution overhead to acceptable levels by instrumenting executables such that they contain instrumentation code only along previously unexecuted paths. A proof of concept prototype tool, which selectively monitors basic blocks in Java programs, has been constructed in a previous work [14]. The tool inserts instrumentation code in each basic block of a user program that has not been executed in previous runs. When the user program is executed under some test cases, information about which monitored basic blocks have been covered is produced. The run time impact of such residual coverage monitoring for example Java programs was shown to be reasonably low (typically 3-5%) after a few iterations of instrumentations and runs.

It is however not clear whether or to what extent this result depends on peculiarities of the platform studied, i.e., relatively slow interpreted code. It is possible that while the marginal cost of residual coverage monitoring is low with respect to execution that is already incurring large overheads for interpretation, the cost of residual coverage monitoring could be much higher relative to compiled code with fewer inherent run-time overheads. It is also an open question whether residual coverage monitoring would still have a significant advantage relative to complete coverage monitoring if the complete monitoring were implemented using the best known monitoring techniques. Program profiling counts the number of times that each basic block or control-flow edge in a program executes. Profiling tools are widely used to measure instruction set utilization, help determine program hot-paths useful for compiler optimizations. These tools can be modified and used for determining software test coverage. Recent work has resulted in optimized profiling tools like *QPT2* [6] and *PP* [10], which significantly reduce the run-time overhead of instrumentation. These algorithms reduce measurement overhead by both inserting less instrumentation code and by placing instrumentation carefully in the program control flow where they are least likely to be executed.

Current Work

This body of work tries to answer the above questions raised about residual coverage monitoring. The Ball and Larus quick profiler, *QPT2*, is used as the primary tool to carry out our experiments. Though this tool primarily profiles and traces code, it was suitably modified to study program statement coverage. QPT2 is built on the EEL executable editing framework that allows us to instrument compiled imperative languages (C, C++, FORTRAN). We compiled SPEC CINT95 [7] benchmarks and instrumented the resulting executables in the following manner: QPT2 is run with a *-s* option, which places instrumentation in every basic block of the executable and creates a fully instrumented (and therefore slower) version of the executable. QPT2 is then run with a *-q* option such that it places instrumentation optimally as defined by Larus and Ball and creates an optimal version of the instrumented executable. Since each benchmark executable comes with several data-sets, after each run of the slow/residual version for a particular data set, the residual version of the executable is re-instrumented such that the any basic block that has already executed in a previous run is not re-instrumented. For each data set, the run time of the original executable is compared to that of the version that has quick instrumentation and the version that has the residual instrumentation. Figure 1 illustrates the whole process.

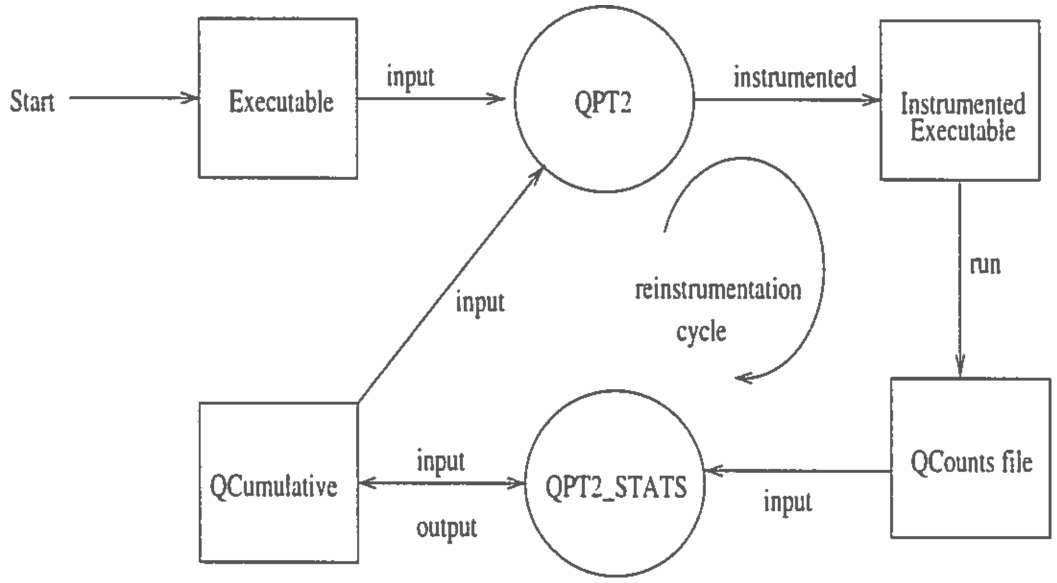

FIGURE 1. Instrumentation and Execution Process

A cumulative coverage file captures the structural coverage by the data sets for the benchmark executable. For a few benchmarks we also reran the above by changing the order of the input data sets (for example, we changed the order of the Perl programs we run when we instrument the 134.Perl benchmark) to see how it effects the performance of the instrumented executable.

The availability of different data sets for each benchmark execution offers a couple of advantages. First, it provides an unambiguous and unbiased set of input data sets. The second main benefit is that it helps in the comparison of residual coverage monitoring against any future monitoring technique. Though the test statistics were collected over several instrumentation runs, the cost of re-instrumentation was not considered to be a factor in evaluating the usefulness of this technique. This is because in practice one would re-instrument only after several test executions. We however have quantified the effect of instrumentation on the size of the executables.

Related Work

Though quite a bit of work has been done to study program run-time behavior using program profiling, including tracing as well as path profiling [5,6,10,11,19], most of it has been done from the perspective of program optimization. Structural coverage is often cheaper than performance profiling and tracing in that it does not require knowledge of the number of times a piece of code executes, but only whether a particular piece of code has been executed at all. Work in the area of program coverage as well as automatic test case generation tools closely resembles our work. Agarwal [2] presents a technique to find a small subset of nodes in a program flow graph with the property that if the subset is covered, the remaining nodes are automatically covered. Thus the tester needs to develop test cases targeted to cover the nodes in the subset rather than the entire set. Our work focuses on determining software program coverage in the beta phase or the deployed environment assuming that all the test obligations are not met, whereas Agarwal's work focuses trying to get meet all the test obligations in the testing phase.

There seems to be very little work done in the area of operational testing or beta testing though an exception to this is in the area of reliability estimation, which is fundamentally different. Podgurski et. al. [3] make the following distinction between synthetic and operational techniques in software testing: Synthetic testing involves selecting test data systematically based on an analysis of a program or it's specification, whereas Operational testing involves having the intended users of the software employ it in the field as they see fit. They describe a technique for operational testing that does not rely only on ordinary beta users and which is intended to provide accurate and economical estimates of the reliability that the software has exhibited in the field. The techniques involves collecting execution profiles of captured beta test executions and applying automatic cluster analysis to the profiles in order to partition the executions based on the dissimilarity of the profiles. A stratified random sample of executions is then selected, reviewed for conformance to requirements, and used to estimate the proportion of failures in the entire population of captured executions. They go on to show with experiments how stratified random sampling produced significantly more accurate estimates of failure frequency than did simple random sampling without requiring larger samples.

Run time assertion checks are also a kind of residual monitoring in that they can be turned off if need be i.e. when performance becomes an issue. The Gnu Nana [13] tool provides flexible ways to deactivate run-time assertion checks for C and C++ programs. Some of the work in this area has been to enrich assertions in Ada to include facilities for formal specification of program behavior [16].

CHAPTER II

INSTRUMENTATION AND PROFILING TOOLS

Residual coverage monitoring addresses performance overhead concerns by instrumenting executables such that they contain instrumentation code only along previously unexecuted paths. In order to be able to do this, we need ways to insert and remove instrumentation from the executables. There are three possible approaches to this: source level instrumentation, compiler modifications and executable editing using compiler-like tools. We chose the latter approach, as it was the most portable as well as the least intrusive solution. This chapter discusses our choice of the executable editing tools and frameworks for instrumentation. It also discusses the details of the executable profiling tools that we used to do performance comparisons and evaluations of executables with residual instrumentation.

Executable Editing Tools

We mainly considered two publicly available tools/frameworks for executable editing and for building the program analysis tools. Executable Editing Library (EEL) [12] is a

C++ library that hides much of the complexity and system-specific details of editing executables. It provides abstractions that allow for the building of a tool without having to be concerned with particular instruction sets, executable file formats, or the consequences of adding foreign code or deleting existing code. Analysis Tools with OM (ATOM) [17], allows limited modification of existing instructions, and invokes foreign (analysis) code through a function call. We chose EEL over ATOM as our instrumenting framework. One of the primary reasons for rejecting ATOM for instrumentation was because ATOM was available on the Ultrix Operating System, which was difficult for the author to gain access to. The choice of the EEL framework was also simplified since profiling tools (QPT and PP) were already available.

The choice of Quick Profiler and Tracer (QPT2) as the optimized profiler was important for several reasons. QPT2 was built using the EEL toolkit, which offered better C++ abstractions, and so was simple to modify. Additionally, Larus and Ball had already done some benchmark calculations of performance improvements with optimized profiling using QPT2 [6]. So, in addition to determining the performance gains that were realized using executables with test residues, we could also compare our results against those of optimized profilers. Finally and most importantly, QPT2 and EEL distributions were available on SPARC architecture and the SunOS operating system which was the main operating system the author had convenient access to.

EEL Abstractions

EEL provides five major abstractions for examining and modifying executables: *executable*, *routine*, *CFG*, *instruction* and *snippet*. An *executable* contains code and data from an object, library or executable file. A tool opens an *executable*, examines and modifies its contents, and writes an edited version. An *executable* primarily contains *routines* and also non-executable data. A tool can examine and modify routines in any order and place them and new routines in the edited executable in any order. EEL represents a routine's body with two further abstractions, *control-flow graphs* (CFGs) and *instructions*.

A *CFG* is a directed graph in which nodes are basic blocks and edges represent control flow between the blocks. EEL provides extensive control-flow and data-flow analysis for *CFG's*. Blocks contain a sequence of *instructions*, each of which is a machine-independent description of a machine instruction. A tool edits a *CFG* by deleting *instructions* or adding code *snippets* to blocks and edges. A *snippet* encapsulates machine-specific foreign code and provides context-dependent register allocation. EEL modifies calls, branch, and jumps to ensure that control flows correctly in the edited program. EEL's internal representation is a register-transfer level instruction description where EEL's instructions try to capture the semantics of a machine instruction.

Figure 2 shows a block diagram of the different EEL abstractions and the relations among them. Later chapters will refer to the above abstractions while specifying the details of the instrumentation. More details of the different abstractions are available in the reference [12].

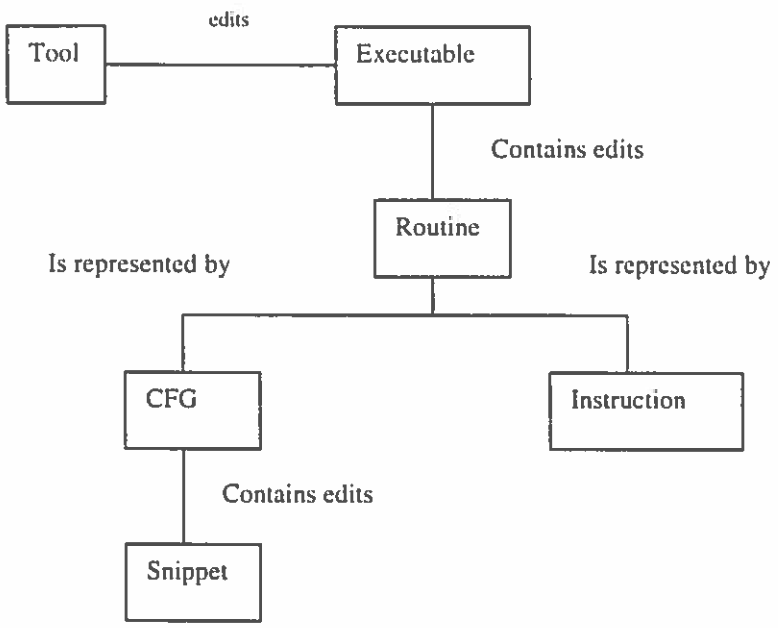

FIGURE 2. EEL Abstractions

Quick Profiling and Tracing Tool (QPT2)

QPT2 is an exact (as different from program statistical profiling tools like the UNIX *prof* and *gprof*) and efficient program profiling and tracing tool built using the EEL toolkit. The programs under consideration are assumed to be written in an imperative language with procedures in which control flow within a procedure is statically determinable. It rewrites a program's executable file by either inserting code to record the execution frequency of every basic block (straight-line sequence of instructions) or control-flow edge or by inserting code to trace every instruction and data reference.

After the instrumented program executes, another program, *qpt2_stats* can calculate the execution cost of the procedures in the program by using results collected during the execution. Unlike the Unix tools *prof* and *gprof*, *qpt2* records exact execution frequency, not a statistical sample. *qpt2* operates in two profiling modes. In a slow mode, it places a counter in each basic block in a program in the same manner as the MIPS tool *pixie*. In a quick mode, *qpt2* places counters on an infrequently executed subset of the edges of the program's control-flow graph. This placement can reduce the run-time cost of profiling up to 3-4 times compared to profiling every basic block. The quick algorithm requires more program analysis and consequently increases the run time of *qpt2* and *qpt2_stats*. The additional cost to instrument a program and report the results, however, is quickly gained back when profiling long-running programs. Additionally, the algorithms used to determine the placement of instrumentation along edges are based solely on the CFG information and does not use any other semantic information that could be derived from the program text like via constant propagation or induction variable analysis.

QPT2 Algorithm Details

QPT2 instrumentation algorithms use the intraprocedural control-flow structure of programs derived from the EEL framework to determine where to place instrumentation code. Interprocedural control flow occurs mainly by procedure call and procedure return (*qpt2* does not handle exceptions and interprocedural jumps). A *CFG* is a rooted directed graph *G = (V, E)* that corresponds to a procedure in the following way: Each vertex in *V* represents a basic block of instruction and each edge in *E* represents the transfer of control from one basic block to another. A *CFG* is created for each *routine* in the program. For slow profiling, the algorithm is straightforward in that each basic block for the *CFG* is instrumented to contain the profiling code.

For the optimal profiling instrumentation, a weighting *W* of CFG *G* assigns a non-negative value to every edge, subject to Kirchoff's flow law: For each vertex *v*, the sum of the weights of the incoming edges to *v* should be equal to the sum of the weights of the outgoing edges of *v*. A spanning tree of a directed graph $G = (V, E)$, is a subgraph $H = (V, T)$, where $T \subseteq E$, such that every pair of vertices in *V* is connected by a unique path (i.e., *H* connects all the vertices in *V*, and there are no cycles in *H*). The tool uses a simple heuristic of giving edges that are more deeply nested in conditional control structures lower weight, as these areas will be less frequently executed. A maximum spanning tree of this weighted graph is constructed such that the cost of the tree edges is maximum. QPT2 then places instrumentation code on the control-flow graph edges not in the spanning tree. More details of the algorithm can be found in [6].

CHAPTER III

PROGRAM DETAILS

This chapter explains the instrumentation details of the tools and the modifications to enable residual test coverage analysis of compiled binary programs. The source code for EEL and *qpt2* is distributed under license [20]. We built the executables using *g++ 2.95* [8] and *binutils* [9] distribution that contains the GNU *bfd* library (a library used by the GNU assembler, linker and the debugger *gdb*) that is used to instrument executables.

EEL Details

Initially EEL reads the program's symbol table and eliminates all duplicate, temporary, and debugging labels in the text segment. Reference [1] and [18] gives details of the UNIX program layout. It also discards labels that are not aligned on an instruction boundary or that are the target of a branch or a jump from the preceding routine. The remaining labels form the initial set of routines. If the executable has no symbol table, the initial set contains only the program's entry point and the first address of the text segment. EEL then makes an extra pass over the program's instructions to find direct subroutine calls. It then examines instructions to find jumps out of a routine or calls on routines not in the initial set. With all this information, EEL then constructs a routine's control flow graph. A reachable but invalid instruction in a CFG leads EEL to assume that the routine contains data. However, unreachable instructions at the end of a routine comprise another routine. CFG's allow EEL to represent the control flow in an architecture-independent fashion. It explicitly represents instructions' control flow in a CFG, so that internal and external control flow are treated uniformly and instructions appear to have no control flow. This is important because now a tool can add foreign code before or after almost any instruction without considering how the code interacts with local control flow, in other words the tool need not be aware of architectural details. A tool edits a routine's CFG by deleting instructions, adding new code before or after an instruction or adding code along a control-flow graph edge. A *snippet* contains the new code. EEL accumulates the edits without changing the CFG and uses callbacks to modify an edit. EEL's instructions are abstractions of RISC-like machine instructions and are divided into functional categories like memory references, control transfers and computations. The instruction interfaces provide several functions to query the information about the instruction. Code snippets are machine specific in that they are typically written in assembly. The registers manipulated within the snippets are placeholders to the actual registers. EEL performs live-register analysis and maps these place holder registers to the actual physical registers.

Figure 3 shows the snippet for incrementing the profile counter for SPARC.

```
3*     mov 0x1, %g1          ! SYS_exit
      ta %g0                ! exit syscall

!!
!! INCR_COUNT records a basic block or edge by incrementing its counter in
!! the count array.
!!
!! Modifies: %g5, %g6 (undefined by SPARC ABI).

      .seg "text"
      .global incr_count

incr_count:

1*     sethi 0x1, %g6        ! upper bits of &counter
2*     ld [%lo(0x1) + %g6], %g5 ! load counter
      add %g5, 1, %g5       ! increment
3*     st %g5, [%lo(0x1) + %g6] ! store counter


      class incr_count_snippet : public tagged_code_snippet
      {
      public:

       incr_count_snippet(addr counter_address)
         : tagged_code_snippet(incr_count_code,
                     sizeof(incr_count_code),
                     NULL,
                     NULL,
                     incr_count_offsets,
                     sizeof(incr_count_offsets))
         {
          SET_SETHI_HI(*find_inst(1), counter_address);
          SET_SETHI_LOW(*find_inst(2), counter_address);
          SET_SETHI_LOW(*find_inst(3), counter_address);
         }
      };
```

FIGURE 3. Sample Snippet Code

Beneath the machine-independent portions of EEL are system- and architecture-specific components that manipulate executable files and machine instructions. Apart from reading and writing Unix executable files using the GNU *bfd* library, it uses a tool called *spawn* to transform a file of annotated C++ functions and a machine description into machine-specific code for analyzing and manipulating binary instructions. More details of this tool are available in [12].

QPT2 and QPT2_STATS

This section discusses the program details of *qpt2*, *qpt2_stats* as well as the modifications we made for the current work. QPT2 is a tool written using the EEL framework. QPT2 inserts profiling code and runs in two modes. Running it with a *-s* option makes it insert profiling instrumentation in every basic block of the executable. It uses the snippet specified in Figure 3 to do this. Running it with a *-q* profiling option makes it insert optimal profiling instrumentation along the edges in a CFG as described in the previous chapter. Both the instrumented executables create a profile file that has a *QCounts* extension. The program *qpt2_stats* analyzes the executable run by reading both the original executable and the *QCounts* file. It can report hierarchical procedural statistics like the unix tool *gprof*, draw CFG's of routines, percentage of the different types of instructions that were executed, register scavenging information etc.

We modified *qpt2* such that when it is run with a *-s* option, it firsts reads a file with the filename of the executable and an extension *QCumulative*. This file has the cumulative coverage statistics from the different executable runs. For the first run, since there has been no coverage, the cumulative coverage file does not exist and *qpt2* instruments every basic block. For subsequent *qpt2* runs, as usual a counter is assigned to every basic block of the executable while the executable is analyzing the CFG of a routine. However, if the corresponding counter in the cumulative coverage file has its count greater than zero, the edit for that particular basic block is skipped. This is done for every basic block in the routine and iteratively done for all the routines in the executable. The executable file is then re-written to give us an executable that carries only residual instrumentation.

Running the instrumented executable produces the *QCounts* file as usual. This file has the record of every executed basic block during the current run. We modified *qpt2_stats* such that it reads both the *QCumulative* and the *QCounts* file and then updates the *QCumulative* file to record the overall coverage. Figure 1 shows the details of the instrumentation process. All in all we had to only make minor modifications to the two executables, *qpt2* and *qpt2_stats*. Also, since the tool was originally written using g++ 2.6, we had to make modifications to resolve compile errors for g++ 2.95, the current version of g++.

CHAPTER IV

EXPERIMENTS

This chapter discusses the details of the various experiments run and their analyses.

Standard Performance Evaluation Corporation Benchmarks

SPEC, the Standard Performance Evaluation Corporation, is a non-profit corporation formed to establish, maintain and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers [7]. We decided to use the SPEC95 although SPEC2000 was available mainly due to memory and execution time constraints. SPEC2000 has high memory requirements in that it recommends the use of 256MB RAM machines. Most of the computers available to the author had 128MB RAM, which was more suitable to run the SPEC95 benchmarks. These benchmarks are designed to provide comparable measures of performance for comparing compute-intensive workloads on different computer systems. We used these benchmarks for comparing the performance of the different versions of the instrumented executables by testing on the same computer system. The CINT95 benchmarks that were used do not stress other computer components such as I/O (disk drives), networking or graphics. These benchmarks emphasize the performance of the computer's processor, the memory architecture and the compiler.

CINT95 consists of 8 moderately large C programs. We compiled these programs with only a level one optimization *(-O)*, so as not to confuse the program's structure and introduce complications in constructing the program's control flow graph. This is the same as the benchmarks run in the base mode to indicate conservative optimizations. Also, these benchmarks are classified as speed benchmarks in that they measure the speed and not the throughput of computation. Timings were run on a UltraSparc SunOS 5.7 sun4u sparc with local disk and 128MB of main memory. We used the UNIX *time* command and the times given in the tables and graphs are the *user time* component of the time. We used the *user time* instead of the usual wall time because this component is not dependent on the system load and additionally the time that is spent in system processing, the *system time* component is low for SPEC benchmarks since they use few system commands. We have documented all the three components, and there is typically little difference (in the order of .05 seconds) between the system and user times. Each benchmark was run twice, and the lowest run time was recorded. The workstation was networked and there was only a single user process running on it. Also, each run was started when the load average on the computer was below 0.05 as measured by the UNIX *uptime* command.

The reference data for each of the CINT95 benchmarks, under the sub-directory *data/ref/input* for each benchmark, was split into 2 or 3 data sets depending on how the benchmark data was organized. The first data set was run with the uninstrumented program, and then with an optimally instrumented program and then finally with a program with full instrumentation. Times and executable sizes were recorded and the feedback files *QCounts* and *QCumulative* were used to re-instrument the executable to obtain the residual instrumented executable. The original, optimal and residual instrumented executables were then run against the second data set and the times, executable sizes and the number of basic blocks instrumented were recorded. This cycle was repeated for a third data set if one was available.

Experiment Results

The following sections give details of each of the benchmarks and how the input data sets were organized.

Results for 124.m88ksim

This benchmark simulates the Motorola 88100 processor running Dhrystone and a memory test program. We used the Dhrystone program as one benchmark data set and the memory test program as the other data set. The results for the performance runs are shown in Table 1. Figure 4 shows the performance results of running the 124.m88ksim benchmark.

TABLE 1. 124.m88ksim Performance Results

Executable	Memory Test (s)	Dhrystone(s)
Original	116.08	171.18
Optimal	166.11	292.26
Residual	239.15	189.03

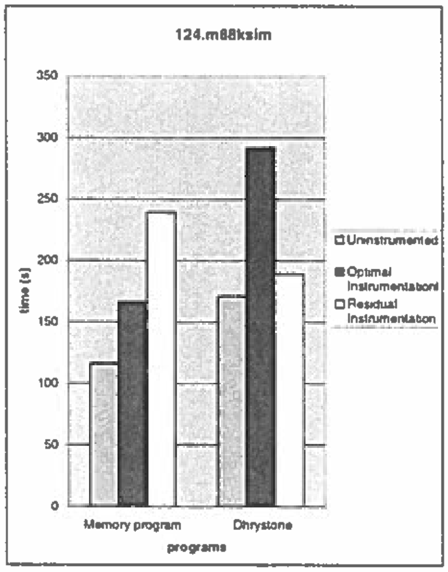

FIGURE 4. 124.m88ksim Performance Graph

A comparison of the sizes of executables created on each run is shown in Table 2. The number of basic blocks instrumented is given in brackets for the residual instrumented executable.

TABLE 2. 124.m88ksim Executable Size Comparison

Executable	Memory Test (Bytes)	Dhrystone(Bytes)
Original	260072	260072
Optimal	540093	540093
Residual	6465589(8377)	622013(6901)

Results for 126.gcc

The gnu C compiler compiles pre-processed source into optimized SPARC assembly code. The benchmark data set consisted of pre-processed source files, which we split into 3 data sets in the following fashion:

- amptjp.i, c-decl-s.i, cccp.i, cp-decl.i, dbxout.i
- explow.i, expr.i, insn-recog.i, integrate.i, protoize.i
- recog.i, reload1.i, stmt-protoize.i, stmt.i ,toplev.i, varasm.i

The run times recorded for each data set were obtained by compiling each file thrice (this was because the benchmark specified the running of 3 copies of each pre-processed source file). Due to the large number of source files, the files that had the longest compile times were used for determining the coverage for each data set. These were *cp-decl.i*, and *expr.i* respectively. The results of the performance runs are shown in Table 3. Figure 5 shows the performance results of running the 126.gcc benchmark in a graphical format.

TABLE 3. 126.gcc Performance Results

Executable	Program Set1(s)	Program Set2(s)	Program Set3(s)
Original	22.68	11.51	14.55
Optimal	31.14	15.54	20.03
Residual	52.05	13.49	16.77

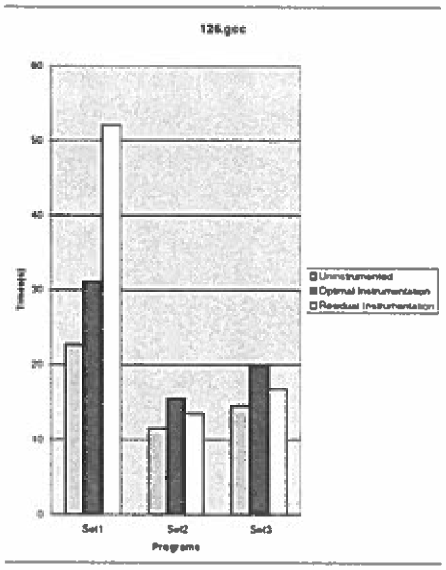

FIGURE 5. 126.gcc Performance Graph

A comparison of the sizes of executables created on each run is shown in Table 4. The number of basic blocks instrumented is given in brackets for the residual instrumented executable.

TABLE 4. 126.gcc Executable Size Comparison

Executable	Program Set1(Bytes)	Program Set2(Bytes)	Program Set3(Bytes)
Original	1663660	1663660	1663660
Optimal	4039932	4039932	4039932
Residual	5432572(11540)	4976337(96785)	4943569(95293)

Results for 129.compress

Compress is a compression program that compresses large text files (about 16MB) using adaptive Limpel-Ziv coding. The amount of compression obtained depends upon the size of the input, the number of bits per character, and the distribution of common substrings. This was the only benchmark for which only one reasonably sized data set existed. Hence, the author ran a smaller version of the provided data-set, and called it small-test and then ran the big-test which was the input data set provided by SPEC. The following were the contents of the small and big tests.

SmallTest.in

1900000 e 2231

BigTest.in

14000000 e 2231

The results are shown in Table 5. Figure 6 shows the performance results of running the 129.compress benchmark in a graphical format.

TABLE 5. 129.compress Performance Results

Executable	Small Test (s)	Big Test (s)
Original	33.30	234.97
Optimal	39.54	283.68
Residual	59.23	235.88

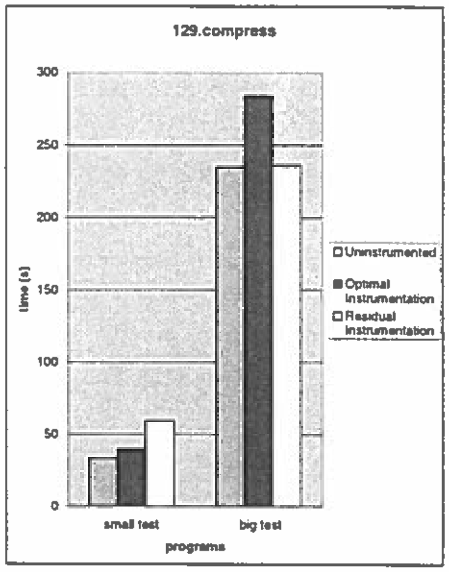

FIGURE 6. 129.compress Performance Graph

A comparison of the executable sizes is shown in Table 6. The number of basic blocks instrumented is given in brackets for the residual instrumented executable.

TABLE 6. 129.compress Executable Size Comparison

Executable	Small Test (Bytes)	Big Test (Bytes)
Original	83904	83904
Optimal	185906	185906
Residual	185906(379)	185906(104)

Results for 130.li

This program is a Lisp interpreter. The benchmark runs several lisp programs and records the run-time for the programs. We split the runs into 3 sets that were divided as follows:

- au.lsp, boyer.lsp, browse.lsp, ctak.lsp, dderiv.lsp, deriv.lsp, xit.lsp
- au.lsp, destru0.lsp, destru1.lsp, destru2.lsp, destrum0.lsp, destrum1.lsp, destrum2.lsp, xit.lsp
- au.lsp, tak0.lsp, tak1.lsp, tak2.lsp, takr.lsp, triang.lsp, xit.lsp

We ignored the programs puzzle0.lsp and puzzle1.lsp as they couldn't run in the isolated data sets. The results of the performance runs are shown in Table 7. Figure 7 shows the performance results of running the 130.li benchmark in a graphical format.

TABLE 7. 130.li Performance Results

Executable	Program Set1(s)	Program Set2(s)	Program Set3(s)
Original	14.38	74.98	105.61
Optimal	17.47	93.39	127.86
Residual	29.08	72.3	107.63

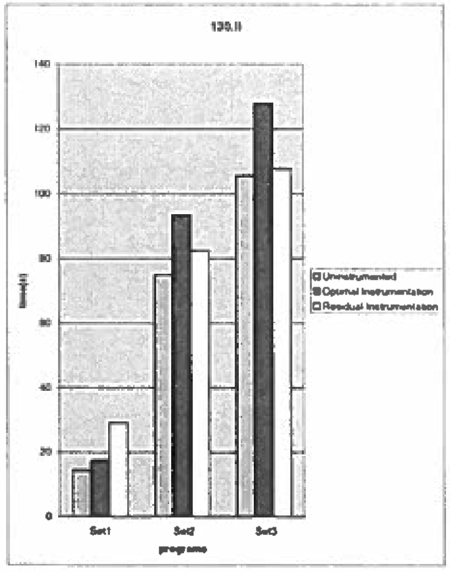

FIGURE 7. 130.li Performance Graph

A comparison of the size of the executables is shown in Table 8. The number of basic blocks instrumented is given in brackets for the residual instrumented executable.

TABLE 8. 130.li Executable Size Comparison

Executable	Program Set1(Bytes)	Program Set2(Bytes)	Program Set3(Bytes)
Original	140884	140884	140884
Optimal	354433	354433	354433
Residual	419969(5734)	395393(4152)	395393(4078)

Results for 132.ijpeg

This program performs jpeg image compression and decompression with various parameters. The results of the performance runs are shown in Table 9. Figure 8 shows the performance results of running the 132.ijpeg benchmark in a graphical format.

TABLE 9. 132.jpeg Performance Results

Executable	Penguin(s)	Specmun(s)	Vigo(s)
Original	142.00	129.44	147.33
Optimal	155.02	140.04	160.9
Residual	198.34	130.93	148.58

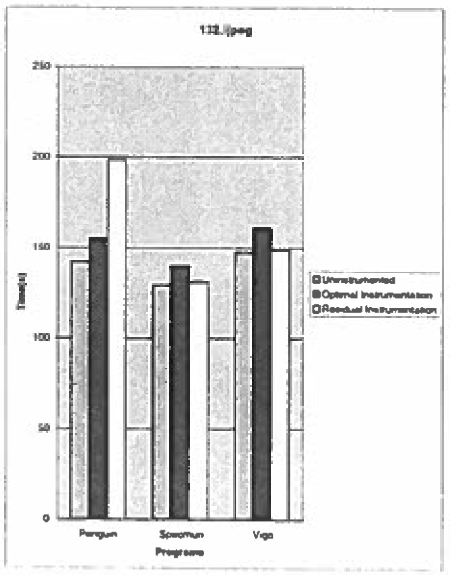

FIGURE 8. 132.ijpeg Performance Graph

A comparison of the size of the executables is shown in Table 10. The number of basic blocks instrumented is given in brackets for the residual instrumented executable.

TABLE 10. 132.jpeg Executable Size Comparison

Executable	Penguin(Bytes)	Specmun(Bytes)	Vigo(Bytes)
Original	232608	232608	232608
Optimal	545558	545558	545558
Residual	660246(9060)	619286(6556)	619286(6556)

Results for 134.perl

This Perl interpreter benchmark performs text and numeric manipulations (anagrams and prime number factoring). The three programs *Primes*, *Scrabble* and *Jumble* were used as the data sets. The results of the performance runs are shown in Table 11.

TABLE 11. 134.perl Performance Results

Executable	Primes(s)	Scrabble(s)	Jumble(s)
Original	71.88	120.88	15.94
Optimal	121.10	219.29	16.91
Residual	277.82	220.48	17.35

A comparison of the size of the executables is shown in Table 12. The number of basic blocks instrumented is given in brackets for the residual instrumented executable.

TABLE 12. 134.perl Executable Size Comparison

Executable	Primes(Bytes)	Scrabble(Bytes)	Jumble(Bytes)
Original	407204	407204	407204
Optimal	994067	994067	994067
Residual	1280787(23544)	1223443(20630	1198867(18961)

Figure 9 shows the performance results of running the 134.perl benchmark in a graphical format.

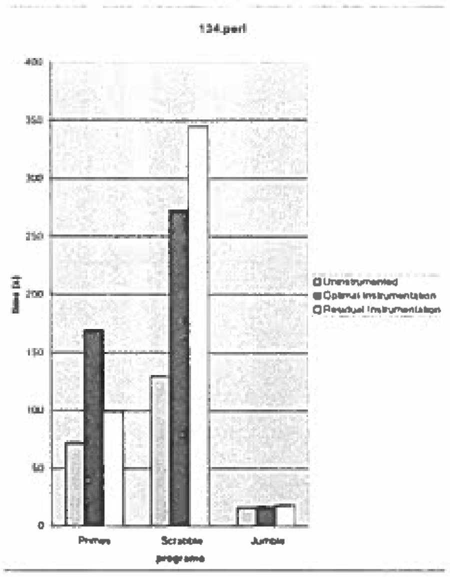

FIGURE 9. 134.perl Performance Graph

We also ran the experiments by changing the order of the Perl programs and the Table 13 shows the performance results. Figure 10 shows the performance results of running the 134.perl benchmark with the changed order of the data sets in a graphical format.

TABLE 13: 134.perl Performance Results with Different Execution Order

Executable	Primes(s)	Scrabble(s)	Jumble(s)
Original	71.99	129.62	15.93
Optimal	169.5	271.18	17.07
Residual	99.27	344.47	18.06

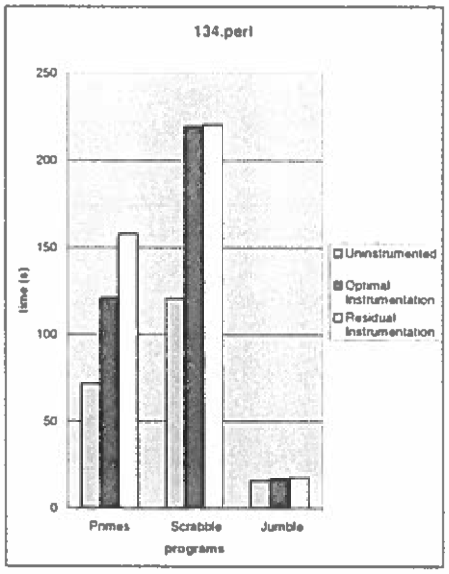

FIGURE 10. Alternate 134.perl Performance Results Graph

A comparison of the size of the executables is shown in Table 14. The number of basic blocks instrumented is given in brackets for the residual instrumented executable.

TABLE 14. 134.perl Executable Size Comparison Using Different Order

Executable	Primes(Bytes)	Scrabble(Bytes)	Jumble(Bytes)
Original	407204	407204	407204
Optimal	994067	994067	994067
Residual	1207059(19499)	1280787(23544)	1198867(18961)

Results for 147.vortex and 099.go

Vortex builds and manipulates three interrelated databases and Go is a game playing software in the area of artificial intelligence. The author had compilation problems with Vortex and hence was unable to run this benchmark. With Go, the author was unable to partition its data sets so as to run the necessary experiments.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

This chapter discusses conclusions of this work and suggestions for future work.

Conclusions

The following were the major contributions of this work:

- We demonstrated through empirical study that residual coverage monitoring is viable for compiled programs, whereas previously we had only data for Java programs.
- We demonstrated that residual coverage monitoring performed quite well when compared to the best of the currently available optimized program profilers.

Previous work had shown that the run time impact of residual coverage monitoring for Java programs was reasonably low after a few iterations of instrumentations and runs [14]. The results from [15] of the two biggest executables analyzed are shown in Table 15 and Table 16 for reference.

TABLE 15. Elevator Program Execution in Seconds

Executable	Test1(s)	Test2(s)	Test3(s)
Original	5.5	6.1	6.5
Instrumented	6.3	6.8	6.6
Blocks	323	240	119

TABLE 16. Instrumentation Program Execution in Seconds

Executable	Test1(s)	Test2(s)	Test3(s)
Original	4.3	1.7	4.4
Instrumented	4.7	1.8	4.4
Blocks	1000	614	547

For the SPEC95 benchmarks, which are all written in C, we have observed that the performance impact of running instrumented executables with residues after running just two data sets decreased considerably. In other words, performance impact of residual coverage monitoring is reasonably low just as for Java programs.

The executables with residues did quite well relative to the executables with optimally placed instrumentation code. Larus [6] had shown that vertex profiling with edge counters was cheaper than vertex profiling with vertex counters when the counters are placed optimally. Our work compared the performances of vertex profiling using edge counters with that of the placing counters in every non-executed basic block, and the latter performed better for a majority of the SPEC95 benchmarks that we ran. The

Perl benchmark was the only exception, where the executable with residual instrumentation did only slightly worse than the optimally instrumented executable.

We are not advocating that residues be used instead of these optimized profiling techniques, only that for structural coverage information, residues appear to remove the performance concerns that users might have while running instrumented executables and hence instrumented executables can very well be used in beta or deployed situations. Also, this work helps us validate the residual approach in that if we had concluded that the optimally instrumented executables had better performance numbers, we would have suggested that these profiling techniques were indeed superior to residual coverage monitoring both in terms of performance as well as in the amount of information that could be derived from the profiles.

We noticed that the executable size did not shrink significantly when the instrumentation from the already executed basic blocks was removed. Also, for each re-instrumenting cycle, we were still re-instrumenting a considerable percentage of the basic blocks. Since the SPEC95 benchmarks are positive benchmarks, i.e. they are guaranteed to run without errors, we believe that there must be several basic blocks of error-handling code that never gets executed and hence these basic blocks will always get instrumented. Though we cannot claim that the benchmarks we used are truly representative of a variety of typical applications, we come close as one of the criteria for the selection of programs as SPEC benchmarks is diverse application areas. Also, we studied only the base rate performance, as we did not want to pursue aggressive optimizations that could confuse the instrumentation tools when they try to determine program structure. In other words, we are limited in our ability of studying residues to the extent of the instrumentation tool's ability to understand any optimized program structure. We inherited other limitations of the executable editing tools that include:

- Ability to run only on the SPARC platform and instrument only SPARC executables. We don't believe this is a serious limitation as the platform specific work is isolated in the *snippets* and the rest of the instrumentation code is portable.
- Inability to deal with dynamically linked executables. The executables we studied were statically linked.
- The tool does not understand semantic optimizations that are typical in current day compilers.

Future Work

In this work we have only tried to address the performance concerns that users might have while using the instrumented versions of the executables. However, the program size of instrumented executables is about 2-3 times that of the uninstrumented ones. The current work has not determined the cause of this, though we suspect that the non-executed error processing code for the SPEC95 benchmarks is partly responsible for this. We believe that this component will be eliminated if testers run negative test cases to cover error scenarios. Tools that address privacy and security concerns for users while providing feedback to developers and testers, allowing them to refine their test data as well as their test cases, are candidates for future work in this area.

Work in the area of program coverage and test case analysis [2] can be extended to incorporate feedback from residual test coverage monitoring. There is currently quite a bit of work being done on path profiling [11,19,4], though hot-paths is the primary focus of this work. The work here could be extended to coverage monitoring so that we could selectively remove instrumentation already executed paths instead of basic-blocks.

BIBLIOGRAPHY

[1] A. Aho, R. Sethi and J. Ullman. *Compilers: Principles Techniques and Tools*. Addison Wesley, April 1986.

[2] Hiralal Agarwal. Dominators, Super Blocks, and Program Coverage. In *Proceedings of the 21st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25-34. ACM, 1994.

[3] Andy Podgurski and Wassim Masri et. al. Estimation of software reliability by stratified sampling. In *ACM Transactions of Software Engineering Methodologies*, volume 8, pages 263-283. ACM, July 1999.

[4] Thomas Ball and James Larus. Programs Follow Paths. *Technical Report MSR-TR-99-01, Bell Laboratories, Lucent Technologies Technical Report BL0113590-990106-01*, January 1999.

[5] Thomas Ball and James Larus. Optimally profiling and tracing programs. In *SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 59-70. ACM, 1992.

[6] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Language Systems*, 16(4):1319-1360, July, 1994.

[7] Standard Performance Evaluation Corporation. SPEC CINT95 distribution. World Wide Web, http://www.spec.org, 1995.

[8] Open Software Foundation. GNU C Compiler Home Page. World Wide Web, http://www.gnu.org/software/gcc/gcc.html.

[9] Open Software Foundation. GNU Project - Free Software Foundation. World Wide Web, http://www.gnu.org/software/binutils/binutils.html

[10] Glenn Ammons, Thomas Ball and James Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85-96. ACM, 1997.

[11] James Larus. Whole Program Paths. In *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, pages 259-169. ACM, 1999.

[12] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 291-300, La Jolla, CA, 1995.

[13] P. J. Maker. Gnu Nana: Improving support for assertions and logging in C and C++. World Wide Web, http://www.cs.ntu.edu.au/homepages/pjm/nanahome.

[14] C. Pavlopoulou. Residual Coverage Monitoring of Java Programs. Master's thesis, Purdue University, 1998.

[15] Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *Proceedings of the 1999 international conference on Software engineering*, pages 277-284. ACM, 1999.

[16] S. Sankar and R. Hayes. Specifying and testing software components using ADL. *Technical Report SMLI TR-94-23*, Sun Microsystems Laboratories, April 1994.

[17] A. Srivastava and A. Eustace. ATOM A System for Building Customized Program Analysis Tools. In *Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 196-205, Orlando, FL, July 1994.

[18] Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley, 1992.

[19] Thomas Ball, Peter Mataga and Mooly Sagiv. Edge profiling versus path profiling: the showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages Software engineering*, pages 134-148. ACM, 1998.

[20] University of Wisconsin, Madison. EEL and QPT distribution. World Wide Web, http://www.cs.uwisc.edu/warts/EEL, 1998.