

Rewriting Executable Files to Measure Program Behavior

James R. Larus and Thomas Ball*

larus@cs.wisc.edu

Computer Sciences Department
University of Wisconsin–Madison

1210 West Dayton Street

Madison, WI 53706 USA

608-262-9519

October 24, 1994

Abstract

Inserting instrumentation code in a program is an effective technique for detecting, recording, and measuring many aspects of a program's performance. Instrumentation code can be added at any stage of the compilation process by specially-modified system tools such as a compiler or linker or by new tools from a measurement system. For several reasons, adding instrumentation code after the compilation process—by rewriting the executable file—presents fewer complications and leads to more complete measurements.

This paper describes the difficulties in adding code to executable files that arose in developing the profiling and tracing tools `qp` and `qpt`. The techniques used by these tools to instrument programs on MIPS and SPARC processors are applicable in other instrumentation systems running on many processors and operating systems. In addition, many difficulties could have been avoided with minor changes to compilers and executable file formats. These changes would simplify this approach to measuring program performance and make it more generally useful.

Keywords: performance, measurement, instrumentation, control-flow graph, executable file

1 Introduction

A program's behavior and performance often is too complex to be understood without the assistance of tools such as an execution-time profiler, memory-leak detector, or cache performance profiler. Parallel programs' performance, moreover, is typically incomprehensible

*This work was supported in part by the National Science Foundation under grants CCR-8958530 and CCR-9101035 and by the Wisconsin Alumni Research Foundation.

without the aid of specialized tools. Performance tools base their results on measurements collected either by hardware monitors, code incorporated into the operating system, or code inserted into a program. The choice of a measurement technique depends on the nature of the desired data, the acceptable level of program perturbation, and the availability of hardware and software resources. This paper discusses techniques for implementing the last alternative, which is widely used because it requires no hardware or operating system modifications and has proven both flexible and effective in many systems.

To measure a program, a tool modifies the program by adding small bits of code (known as *instrumentation code*) that record program events or collect other data.¹ This code can be added during any stage of the compilation process. Adding the code to executable files—at the end of the process—requires no changes to system tools such as compilers or linkers and exposes the complete program (including libraries). Instrumenting executables permits complete and detailed measurements and simplifies the use of a measurement tool since programs need not be compiled or recompiled in a different manner to measure them.

Rewriting an executable file has three stages. The first, measurement tool-independent stage *analyzes* the original program to find its control structure and identify its data references. In the second stage, the measurement tool uses this analysis to select instrumentation code and decide where to insert it. In the third stage, the program is *instrumented* by inserting the measurement code and modifying existing instruction and data references to preserve the program's behavior.

Analyzing and instrumenting executables is straight-forward in the abstract. However, machine instruction sets and executable file formats are designed to be executed, not analyzed and modified. In systems, such as the one described by Johnson [2], in which the compiler and linker are developed in conjunction with the measurement tools, minor changes in the other parts of the system considerably facilitated instrumentation. An opportunity to start afresh is unfortunately rare and most tools will manipulate executables produced by unmodified compilers in existing formats.

This paper discusses our techniques for implementing the first and third stages.² These techniques are very general. Other instrumentation systems, running on different processors and operating systems, can use them to analyze and modify executables. We developed these techniques for a program profiler, `qp`, and program tracer, `qpt`. `qp` is a basic block execution profiler similar to MIPS's `pixie`. In addition to the naive approach of placing counters in each basic block, it implements a sophisticated algorithm for placing counting code that reduces profiling overhead by a factor of four or more [3]. `qpt` is an improved version of `qp` that use abstract execution [4] to efficiently trace a program's instruction and data references. The discussion below only mentions `qpt`, but many of the techniques originated in `qp`. Abstract execution dramatically reduces the cost of program tracing and the size of trace files by factors of 50 or more or simple techniques that record every reference. `qpt` is a second generation tracing system. The earlier version, `AE`, was a modification to the Gnu C compiler `gcc` that inserted tracing code while compiling a C program. Note that `qpt`, because it operates on executable files is relatively language-independent and has been

¹This approach can dynamically check many program properties, as well. For example, *Purify* adds instrumentation that detects memory leaks [1].

²Our second-stage techniques for profiling and tracing are described elsewhere [3, 4, 5].

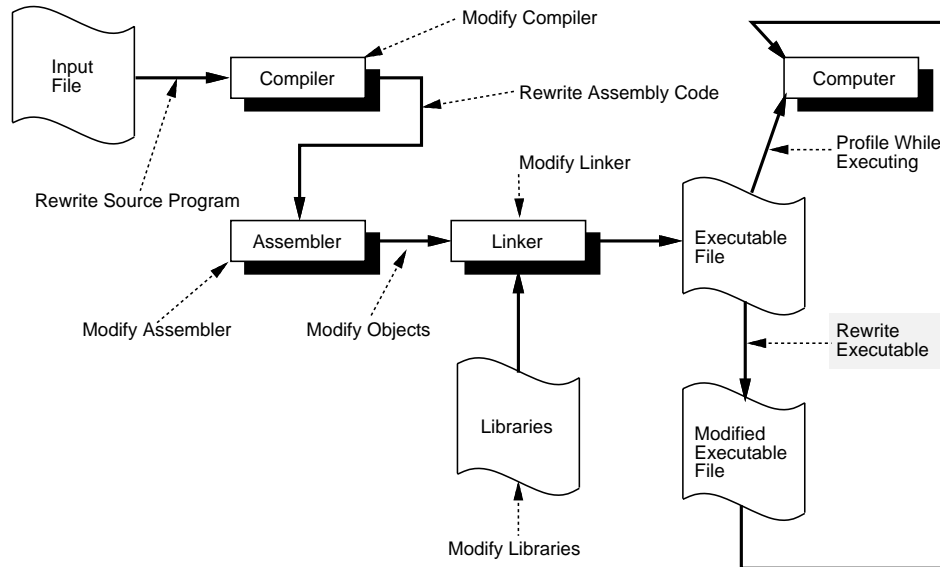


Figure 1: Options for inserting instrumentation code into a computer program. This code can be inserted at any stage in the compilation process. However, adding the code to executables, at the end of the process, permits complete and detailed measurements and simplifies use of tool since programs need not be compiled or recompiled in a different manner to measure them.

used extensively on unoptimized and optimized C, C++, Modula-3, and Fortran programs.

The paper divides into three parts. The first part (Sections 1.1 and 2) discuss related work and provides a general overview of adding code to executables. The second part (Section 3) details the problems caused by existing executable files, compilers, and linkers and outlines our solutions to them. Finally, Section 4 suggests simple changes to compilers and executables that would simplify the technique of program instrumentation.

1.1 Related Work

Code modification is a widely-used technique for measuring program performance. Tools can add instrumentation code during any stage of the compilation process (see Figure 1).

Even before compilation, a source-to-source program transformation can add measurement code directly into a program’s text. This type of instrumentation has been used to: measure source-level characteristics, such as the type of statements executed [6]; estimate potential parallelism [7]; and monitor program behavior in functional languages [8]. However, the high-level language source of a program is often far removed from the instructions executed by a computer, so this approach is unable to measure details of the instruction or memory reference patterns. Nor can it be applied to code, such as libraries, for which source text is unavailable.

During compilation, a specially-modified compiler can insert instrumentation code into a program as it is compiled. The Unix `prof` and `gprof` profilers [9] use this approach, as do other measurement systems [4]. When a compiler’s source is available, this approach

enables the measurement system to exploit the compiler’s analyses to reduce the cost of measurement by intelligently placing instrumentation code [3, 4]. In addition, a compiler’s mappings between syntactic and semantic structures aid in placing instrumentation code and in reporting measurements in understandable terms, such as procedure and variable names and line numbers.

However, modifying compilers has disadvantages. Commercial compilers’ source code is almost always unavailable. In addition, programs can be written in several, different languages, each of which has its own compiler, and linked to libraries that are supplied pre-compiled. Another problem is that compilers typically defer code generation until the final stage, so the instrumentation code only manipulates the compiler’s intermediate representation, which blurs the association between instrumentation and particular instructions. Finally, a program must be recompiled to measure it, which increases the time and effort to study the program and which prevents measurement of precompiled programs.

After compilation, the effects of the compiler’s optimization and code generation are visible and can be directly measured. One possibility, which addresses many problems in the previous approach, is to change the assembly language produced by a compiler before passing it to the assembler [10]. On some systems, however, generating assembly language changes the compiler’s behavior. For example, on MIPS systems, assembly code contains no debugging information. In addition, MIPS assembly language does not correspond directly to binary instructions. The MIPS assembler generates instructions that translate assembler pseudo instructions into machine instructions and also reorganizes instructions to minimize pipeline delays.

Similarly, if sources of the assembler or linker are available, either tool can insert the instrumentation code. The linker processes the entire program, including libraries, so it can ensure that all modules are measured and can compute interprocedural information [11, 12]. By contrast, a program instrumented earlier in the compilation process requires specially-instrumented libraries. The linker also has relocation information and binds addresses, which greatly simplifies the process of inserting additional code.

A clever approach, which effectively uses this relocation information, is to modify object files between the steps of assembly and linking [1]. This approach, however, requires relinking an entire program to measure it and it cannot be applied to programs whose object files are unavailable.

A very different approach is to dynamically measure programs with either the process-control mechanism used by debuggers [13] or other, faster breakpoints [14]. The cost of the debugger’s process-control mechanism, which entails two context switches per interaction, is far too high for this approach to be competitive. Kessler’s fast breakpoints dynamically patch a running executable to jump to the instrumentation code. Their overhead is comparable to the approach described in this paper.

The final alternative, advocated by this paper, is to insert instrumentation code by rewriting an executable file (`a.out`) to incorporate the code. MIPS’s profiling and tracing tool `pixie` has long used this approach [15]. In addition, Johnson used the process, which he called “postloading,” to both measure and optimize programs [2]. Unlike the system described in this paper, Johnson’s postloader operated in conjunction with a specially-modified compiler and linker that simplified the process by eschewing difficult optimizations and by

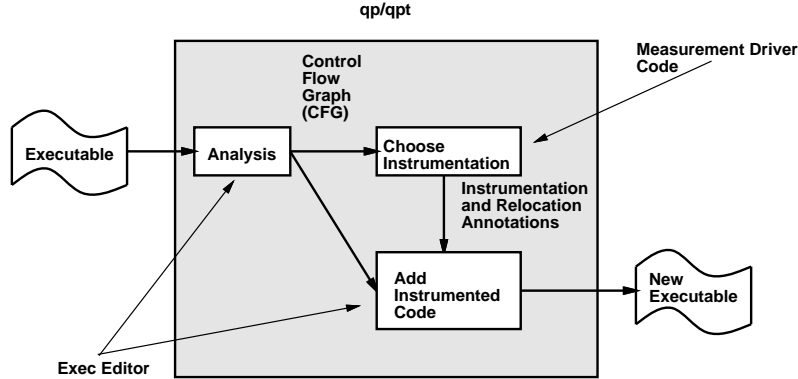


Figure 2: Overview of instrumentation process. Program analysis constructs a control-flow graph (CFG), which is used by the measurement driver code to decide where to place instrumentation and by the exec editor to update instruction addresses to reflect the additional code.

passing along additional information. Wall also used the approach, which he called “late code modification,” in systems that measure and optimize programs [16]. A related application, which requires many of the same techniques, is to translate executable files so they run on a different architecture or operating system [17, 18].

For convenience, we call the technique of adding code to an executable *rewriting an executable file* and the portion of a measurement tool that performs it an *exec editor*. The rest of the tool that chooses the code is called the *measurement driver code*. This alternative has many advantages. An exec editor is largely independent of earlier stages in the compilation process. It requires no sources for, or modifications to compilers, assemblers, or linkers. Also, it works for programs written in different source languages and compiled with non-standard (e.g., `gcc`) compilers. In addition, executable rewriting manipulates entire programs, including precompiled libraries whose source is unavailable.

2 Overview of Exec Editing

This section is an overview of the instrumentation process in `qp`. It outlines the major steps in the process, but does not describe the complications, which are explored in the next section. Figure 2 shows the process. The first step is program analysis, which constructs a control-flow graph for each procedure in the executable. These graphs, which identify potential paths through a procedure, are the basic program representation used by both the measurement driver code and the exec editor. The measurement driver code in `qp` computes the best locations for instrumentation and passes the location and instructions for each instrumentation sequence to the exec editor portion of `qp`. This part of the program decides how to modify the program and how to adjust jump and branch offsets to accommodate the profiling or tracing code. Finally, the exec editor writes a new executable file containing instrumented routines and an updated symbol table.

In producing this new executable, an exec editor can modify a program’s instructions but not its data. As described below, jumps and branches can be analyzed. However, data

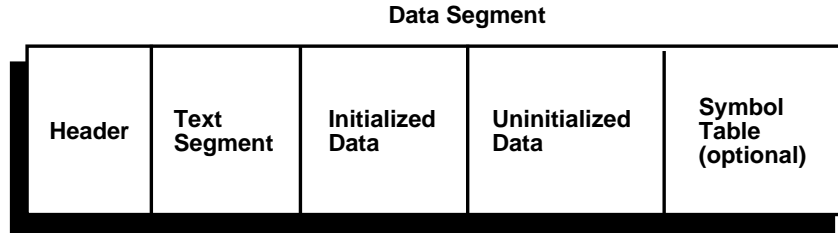


Figure 3: Organization of a Unix executable file. The file header identifies the pieces of the file. The text segment primarily contains a program’s instructions. The data segment is divided into initialized and uninitialized data. Finally, a file may contain a symbol table to aid in debugging.

references are more difficult to track accurately because of indirect references, so data in the modified program must remain in the same place. This constraint, fortunately, does not prevent an exec editor from adding new data after the existing data.

`qpt` processes each routine in an executable separately. Although interprocedural analysis would aid program analysis and simplify instrumentation in a few cases (described below), we avoided it because of its potentially large cost in time and space. In retrospect, this was a bad choice because the problems caused by the lack of information about interprocedural jumps and branches could have been avoided by an quick initial examination of the program that collected limited interprocedural information.

A Unix executable file (`a.out`) typically contains five major parts (see Figure 3). The first is a header that records the size and locations of the other pieces. The next is the *text segment*, which primarily contains a program’s executable code. The *data segment* contains statically-allocated data and is composed of *initialized data* and *uninitialized data* (also known as *bss*). The latter is not explicitly represented in an executable. Instead, the file’s header records the size of *bss* segment and its zero-filled space is allocated when the program is loaded into memory. Following the program is an optional symbol table that maps names and line numbers in the source program to instruction and data addresses. The detail and quality of information in this table varies widely, depending on the format of the executable and the optimization and debugging level at which a program was compiled.

`qpt` first reads an executable file’s header and symbol table. To analyze a program, `qpt` only needs the starting address of each procedure, but it also extracts additional information for the measurement driver code, so as to avoid a second pass over the table. If a file is stripped (i.e., its symbol table was removed to save space), `qpt` quits. As discussed below, `qpt` could process stripped files since its control-flow analysis can identify the procedures. However, the lack of routine names makes it difficult to meaningfully report profiling information, so `qpt`, reflecting its origins, does not analyze these files.

`qpt` next processes each procedure individually by building its *control-flow graph (CFG)*, which is a common compiler data structure that concisely represents flow of control in the procedure. Nodes in a CFG are called *basic blocks*. They delimit straight-line, single-entry, single-exit sequences of instructions. Edges represent jumps between blocks or fall-throughs between consecutive blocks. `qpt` constructs a CFG in two passes over a routine’s instructions. The first pass examines each instruction to identify control transfers (branches and jumps)

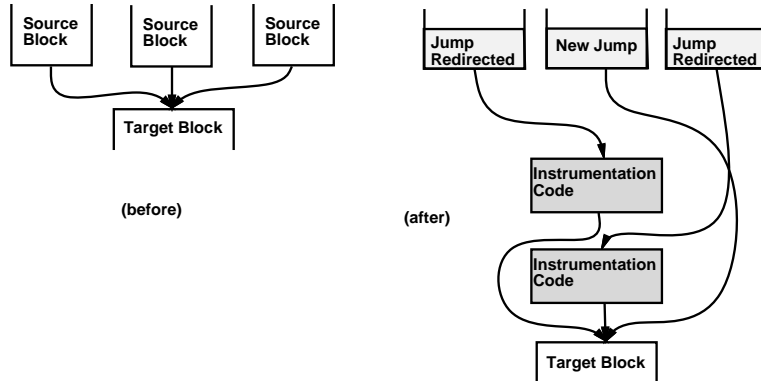


Figure 4: Instrumenting edges. The instrumentation code for an edge is prepended to the edge’s target block. Putting code on edges may sometimes requires additional jumps in the source blocks and at the end of the instrumentation code.

and, from them determines the first and last instruction in each basic block. The second pass records edges connecting the blocks. The code to build a CFG is machine-independent and relies upon a small collection of machine-specific routines that categorize an instruction and determine destination addresses for jumps.

The measurement driver code in `qpt` uses the CFG to place instrumentation code in optimal locations [3]. The first step computes a weighting that assigns a likely execution frequency to each CFG edge. A weighting is either computed by a heuristic based on the structure of the CFG or is derived from a previous profile of the program. After computing the weighting, `qpt` computes a maximum weight spanning tree of the CFG. This set of edges is the largest and costliest (i.e., most frequently executed) subgraph of the CFG that need not be instrumented. All edges not in the tree must be instrumented to record either the number of times that they execute or the sequence in which they execute, depending on whether a program is profiled or traced. The information recorded along these edges is sufficient to reconstruct a full profile or trace.

Instrumenting edges, as opposed to basic blocks, is one of `qpt`’s innovations. We show elsewhere [3] that instrumentation code on edges is always as good, and frequently much better, than code limited to blocks. In practical terms, placing code along edges is only slightly more complex. Instrumentation code for a target block is placed immediately before the block’s code (see Figure 4). The difficulty comes in redirecting control flow so as to land on the appropriate instrumentation without introducing unnecessary jumps.

Figure 5 describes the algorithm for laying out the instrumentation code for a block. The last instruction in a predecessor block either jumps to the target block or falls through from the previous block. In the former case, the jump can be redirected to either land on the instrumentation code or, if the edge is not measured, to bypass the instrumentation and go directly to the target block. In the latter case, control falls through to instrumentation code, if the fall-through edge’s code is placed first in the block’s sequence of instrumentation code. If the fall-through edge is not instrumented, but other edges are, it requires an additional jump to avoid the instrumentation code. In the best case of a measured fall-through edge and other edges uninstrumented, this algorithm introduces no additional jumps. In `qpt`,

```

count  $\leftarrow$  0;
foreach  $p \in \text{pred}(B)$  do
  if  $\text{is\_instrumented}(p \rightarrow B)$  and  $\text{is\_fall\_thru\_edge}(p \rightarrow B)$  then
    begin
       $\text{output\_instrumentation\_code}(p \rightarrow B)$ ;
      count  $\leftarrow$  count + 1;
      if  $\text{number\_instrumented\_edges}(\text{pred}(B)) > 1$  then
         $\text{output\_jump}(B)$ ;
    end

  foreach  $p \in \text{pred}(B)$  do
    if  $\text{is\_instrumented}(p \rightarrow B)$  and not  $\text{is\_fall\_thru\_edge}(p \rightarrow B)$  then
      begin
         $\text{output\_instrumentation\_code}(p \rightarrow B)$ ;
        count  $\leftarrow$  count + 1;
        if count  $\neq$   $\text{number\_instrumented\_edges}(\text{pred}(B))$  then
           $\text{output\_jump}(B)$ ;
      end

```

Figure 5: An algorithm for laying out instrumentation code for incoming edges to block B . It first outputs instrumentation for the fall-through edge, if one exists, to avoid introducing an unnecessary jump. It then produces the instrumentation for the other edges, all of which, except the last one, require a jump over later instrumentation code.

approximately half of the measured edges fall in this category.

`qpt` also inserts instrumentation code *within* blocks to trace address references. Memory references arise from either explicit load or store instructions or implicit references to memory-resident operands. The set of instructions that compute an address for a memory reference is called its *address slice*. `qpt` finds an address slice by computing the transitive closure of the instructions whose value is used to compute an address. Instructions whose results cannot be recomputed when the trace is regenerated require instrumentation code to record their value [4]. Memory-reference instrumentation code is similar to control-tracing code, but since this code is contained within a block, it does not require additional jumps.

Inserting either form of instrumentation code changes the distance between instructions and requires adjustment of conditional branches and unconditional jumps. Direct branches and jumps, which specify the address or offset of their target, are easily modified to accommodate the instrumentation code. Indirect jumps that do not go through a jump table are more difficult since the target address is unknown until run time. The exec editor in `qpt` adds code to translate the target address immediately before an indirect jump executes. This code uses a complete map between the original and new address of each instruction (see Section 3.2) that is stored in the instrumented executable.

The final stage in processing a routine is to write the modified code to a new executable file. At this point, `qpt` has no direct representation of the instrumented routine. Instead, annotations on both flow graph edges and blocks and in auxiliary tables describe the instrumentation code and changes to existing instructions. `qpt` writes the instrumented routine to a temporary file. After processing all routines, `qpt` goes back and copies this code into a new executable file. This two step process is necessary in order to patch forward calls (i.e., to a routine later in the executable), whose eventual address is unknown when the call is first encountered.

While creating the new executable file, `qpt` updates the symbol table information to reflect the new memory locations of routines. Fixing this information permits source-level debugging of instrumented executables. This feature is particularly useful to the developer of an instrumentation tool! In general, updating symbol table information entails a straight-forward replacement of addresses in the original executable with addresses in the new executable. However, certain compact information encodings, for example, MIPS systems' line number table, complicate the process since the mapping between old and new information may not be one-to-one.

3 Complications in Exec Editing

Rewriting an executable is not as simple as the description above suggests. Quirks in computer instruction sets and missing information in executable file formats complicate the analysis and insertion of new code. The latter problem could be mitigated by simple changes in the file formats (discussed in Section 4). However, until these changes become widespread, exec editors have to live with existing machines and file formats. The techniques discussed in this section, which were developed to instrument programs on two popular RISC architecture (MIPS [19] and SPARC [20]), are useful on many other systems. Other computers and operating systems share similar problems, many of which can be resolved by these techniques.

Both the MIPS and SPARC processors are reduced instruction set computers (RISCs). This type of processor instruction set offers many advantages for exec editing. Most important are a small instruction set and fixed-length instructions, both of which simplify the decoding of binary instructions when constructing the CFG. On the other hand, other characteristics of RISCs, most notably the exposed pipeline (i.e., delayed loads and branches), complicate exec editing and are discussed below.

MIPS and SPARC systems use different executable file formats. MIPS uses a proprietary format called ECOFF, which is an extension to the COFF format, while SunOS uses a derivative of the BSD `a.out` format.³ ECOFF is a more complex format that contains significantly more precise debugging information. The additional complexity provides few benefits for `qpt`, which extracts only three pieces of information from the MIPS symbol table: a procedure's starting addresses, whether the procedure comes from an assembly-language file, and the register that holds the procedure's return address. Because of SPARC systems' register-use conventions and less aggressive compilers, the second and third piece of information are unnecessary on that machine. On the other hand, ECOFF's complexity does not adversely affect `qpt`, mainly because of MIPS's `ldfcn` library, which hides many details of the data structures.

The discussion below is organized by problem. Each subsection describes how a hardware or software decision complicates analysis or instrumentation and explains how `qpt` handles the difficulties. The first two problems are caused by branch instructions, which complicate program analysis and modification. The third problem comes from the condition codes on SPARC processors. The next two problems are caused by compiler conventions that also complicate these two tasks. The final group of problems are primarily due to shortcomings in information contained in an executable.

3.1 Delayed Transfers

Delayed control-transfer instructions (e.g., delayed branches, jumps, and subroutine calls) cause a few problems in constructing flow graphs and adding instrumentation code. A delayed instruction does not execute immediately. Instead, its effect is delayed until the following instruction executes [21]. For example, a delayed branch executes in two stages: the first performs the conditional test and the second jumps to the target. The second stage overlaps execution of the instruction immediately following the branch. An *annulled delayed branch* only executes the instruction in the delay slot if the test succeeds and prevents execution if the test fails. Delayed instructions cause difficulties in instrumenting programs. The first occurs when an instruction in a delay slot belongs to two basic blocks. The other arises when it is necessary to add some code immediately after an instruction in a delay slot.

Annulled branches complicate construction of the control-flow graph (CFG). A CFG is poorly suited to represent out-of-order instruction execution. In a conventional CFG, a basic block is a straight-line (executed in order) code sequence that commonly ends at a branch or jump. A non-annulled branch slightly complicates the representation since the block ends after the branch's delay slot, but the instructions still execute in serial order. In an annulled

³Sun's new operating system Solaris uses a different format that resolves some problem, as discussed below.

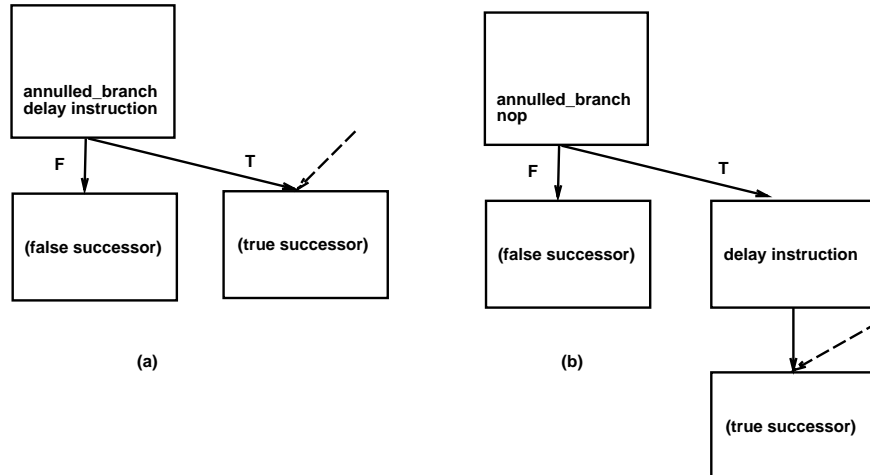


Figure 6: Representation of annulled branches in a CFG. Part (a) shows the CFG of a block ending with an annulled branch. The delayed instruction only executes if control passes to the block labeled “true successor.” Part (b) shows a better representation, where the delayed instruction is moved to its own block along the correct control-flow edge.

branch, however, the instruction in the delay slot executes conditionally and so a block is no longer straight-line code. `qpt` currently builds the same, conventional CFG, representation for annulled and non-annulled branches and provides predicates to distinguish blocks that end with each type of branch. This solution works, but complicates the measurement driver code, which must be aware of the different branches.

A better representation is a CFG in which the instruction from an annulled branch’s delay slot is put in its own block along the true edge of the conditional branch’s block (see Figure 6). With this approach, the measurement driver code sees a conventional CFG with straight-line code in each block. This change in representation does not imply a change in the program’s instructions. The exec editor can easily retain the annulled branch in the instrumented executable code.

Another problem is caused by an instruction scheduling optimization that puts the same instruction in two basic blocks. The instruction in a delay slot is part of two blocks if it is also the target of a jump (see Figure 7a). The instruction is both the last instruction of one block and the first instruction of the subsequent block. The overlap can complicate adding instrumentation since code cannot fit between the blocks. The obvious solution, which works well in practice, is to undo this minor optimization by duplicating the instruction so each block has its own copy.

Delayed branches cause more serious instrumentation problems when the instruction in a delay slot produces a value that must be recorded (see Figure 7b). The instrumentation code that records the value cannot be added after the instruction since the branch transfers control immediately after the instruction executes. In `qpt`, traced values originate either from load instructions or function calls. A call from within a delay slot is ineffectual, so `qpt` only encounters memory loads in delay slots. An alternative is to move the load and instrumentation code immediately before the delayed branch. On MIPS processors, the

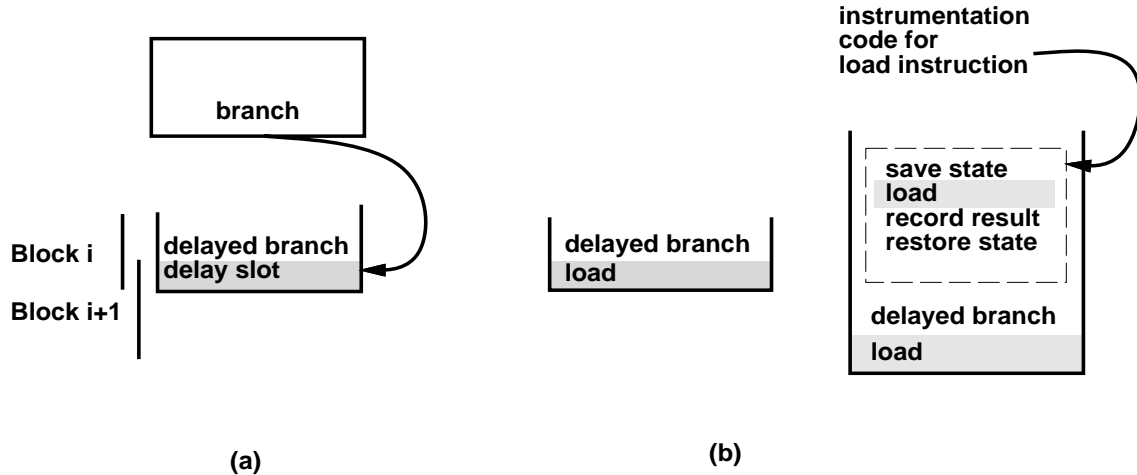


Figure 7: Complications caused by delayed branches. Part (a) shows the problem that occurs when two basic blocks overlap because the last instruction in one block (in its delay slot) is the first instruction of the next block. Part (b) shows how a load instruction in a delay slot can be traced. The tracing code must be placed before the delayed branch and must save and restore state to avoid affecting the outcome of a conditional branch.

moved load, however, may modify a register used by the subsequent conditional branch. The general solution is to execute the load as part of the instrumentation code that records its value, before the conditional branch. This code sequence masks the load's effect, by saving and restoring its target register, so as not to affect the subsequent branch or original load, which executes in the delay slot and modifies the program's state. Another solution, which works for delayed branches, but not delayed procedure calls, is to move the delayed instruction to both the target and fall-through blocks and trace it there.

3.2 Indirect Jumps

Indirect jumps are commonly perceived to be a serious impediment to constructing CFGs and instrumenting programs. The perception is incorrect because compiled code almost always uses these jumps in a stylized manner that can be analyzed. Hand-written assembly code can use indirect jumps in an uncontrolled manner that makes control-flow analysis impossible. Fortunately, this problem is rare. Aside from threads packages, `setjmp`'s and `longjmp`'s, and procedure returns, indirect jumps occur only in `switch` statements in which the alternative targets are collected into a jump table. No other commonly-used constructs in higher-level languages have an obvious analogue or implementation with indirect jumps (as opposed to indirect calls).⁴

Indirect jumps through jump tables are innocuous, since, when the table can be found (see Section 3.4), it contains the jump's destination addresses. These locations demarcate

⁴Exceptions to this rule are continuations in Scheme, ML, and other functional languages, which are just indirect jumps on a larger scale, and assigned goto's in Fortran. Both constructs are very difficult to analyze precisely [22], though this problem has not prevented `qpt` from being ported to trace ML programs.

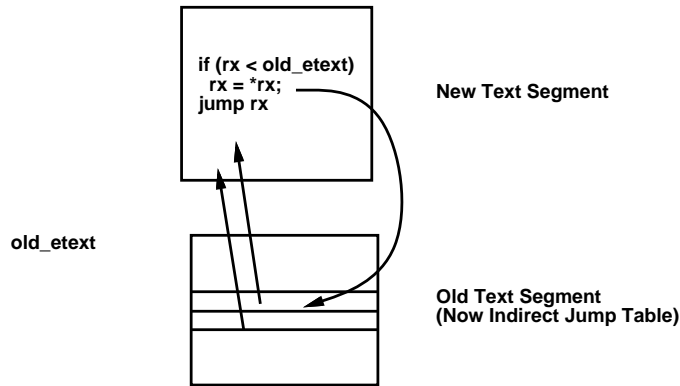


Figure 8: Translation table for indirect jumps. `qpt` stores addresses in a program’s original text segment, so it can dynamically translate an unknown address at an indirect jump. If jump’s target address is in the old text segment (below `old_etext`), it is dereferenced to obtain the new location of the target instruction.

CFG edges. In addition, the table entries can be easily updated to redirect the jump to the new locations of its target blocks.

On the other hand, if a jump table cannot be found or if an indirect jump is not part of a `switch` statement, the instrumented code must ensure that the jump lands on the relocated, not original, address of its target block. This complication arises in the rare hand-written `switch` statement and in control transfers in the `setjmp` routine and threads packages. The exec editor cannot redirect these jumps because the target address is a data value that is difficult to detect and translate statically. Without special attention, the instrumented code would fail at these indirect jumps because their destination is an address in the uninstrumented program.

To avoid this error, `qpt` uses a program’s original text segment as a translation table that dynamically maps from addresses in the original program to addresses in the new program (see Figure 8). Location i in the old text segment contains the address, in the modified program, of the instruction that original was stored at i . Before an indirect jump to an unknown destination, a small amount of exec editing code compares the jump address against the end of the old text segment. If the address is lower (i.e., it is a jump into the original address space), the code dereferences the translation table to find a new target for the jump. The same mechanism enables `qpt` to trace programs that use signals—which MIPS’s `pixie` does not permit. The `sigvec` system call, which establishes a signal handler, requires the address of a function to invoke at a signal. This address is a literal value that is difficult to recognize and translate in a program. However, `qpt` easily recognizes a `sys_sigvec` system call and modifies it to translate the function’s address.

The drawback of a translation table is extra memory. The table doubles the size of the original text segment, while instrumentation code expands it further by 70%–200%, so the translation table requires about one-third of the instrumented text space.

3.3 Condition Codes

The SPARC processor uses four condition code registers in conjunction with conditional branch instructions. Variants of the arithmetic instructions set the condition code flags, which affect subsequent conditional branches. Condition codes complicate instrumentation, since the additional code must not affect a live condition code (one whose value will subsequently be used). On one hand, the SPARC's instruction set is good because each arithmetic instruction has a variant that does not affect the condition codes (unlike computers such as the VAX, in which every arithmetic instruction affects the condition codes). On the other hand, the SPARC's instruction set does not include an instruction that allows a user-mode program to save and restore the condition codes.

Most of `qpt`'s profiling and tracing instrumentation code has no effect on condition codes. The one exception is the check for a full trace buffer that needs to be flushed to disk. `qpt` works around its inability to save and restore condition codes in two ways. First, it tries to place buffer checks at places in which the condition codes are not live. In general, this is successful because a value typically is tested immediately before a branch and so the condition codes are dead in most of the program. However, in small, highly-optimized loops, such as:

```
loop: ld [%10], %11
      add %10, %10, 4
      add %11, %12, %12
      bne loop           ! branch on condition codes
      cmp %10, 100      ! set condition codes for next iteration
```

the condition codes are live throughout the loop. `qpt` must insert a buffer check in the loop, so it uses a more expensive code sequence that has no effect on condition codes. This code subtracts the current buffer pointer from the buffer's end and adds the scaled sign bit of the difference to the current PC to compute two different instruction address depending on whether the buffer is full. This code executes twice as many instructions, but does not change the condition codes.

A more expensive, but general solution, which was unnecessary in `qpt`, is to save and restore the condition codes with a sequence of branch or arithmetic operations that indirectly access or modify the condition codes. For example, the *Z* bit indicates if the last ALU operation produced a result of zero. It can be tested with the `be` and `bne` instructions and set by `add %0, %0, %0`.

3.4 Code and Data

Some compilers store read-only data in a program's text space. This has several benefits: it shares a single copy of the data among multiple, concurrently-executing processes (since text segments are usually shared); it reduces the distance between instructions and data, thereby permitting more efficient addressing; and it ensures that values are not modified (since text segments are usually read-only). However, mixing code and data greatly complicates constructing CFGs because distinguishing data from instructions is often difficult.

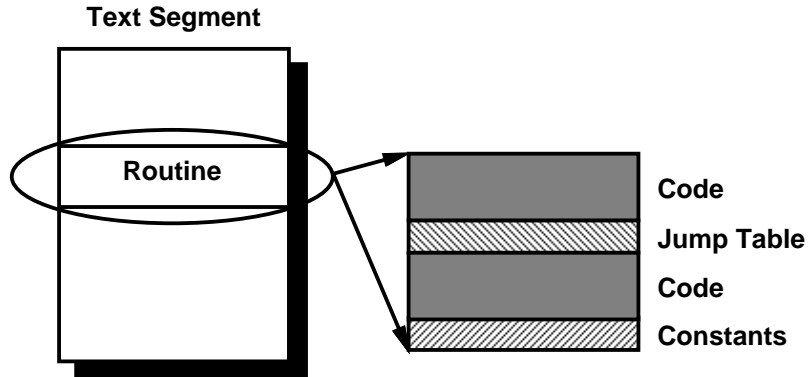


Figure 9: Data in the text segment. Data in the text segment occurs in two places. The first is jump tables (e.g., for `switch` statements). The second is literal constants.

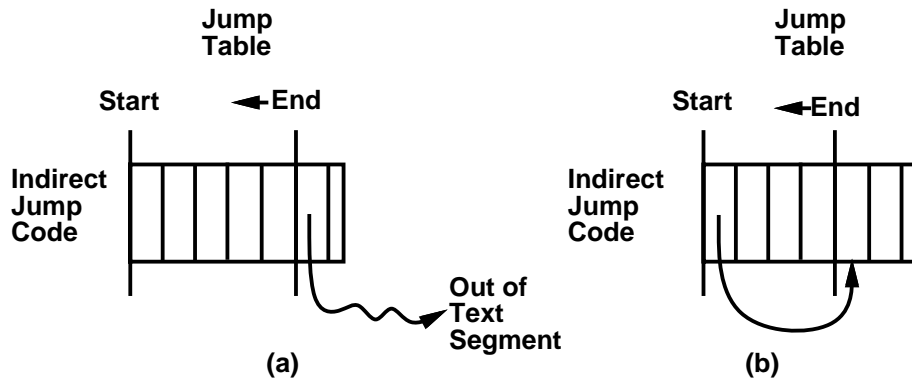


Figure 10: Finding end of a jump table. When the last address in a jump table cannot be determined by examining instructions, it can be found by examining table entries. An entry that is not a valid instruction address clearly marks the table's end. Conversely, valid addresses in the table point to instructions beyond the table.

A compiler puts data in the text segment for two reasons (see Figure 9). The first is jump tables for `switch` statements. MIPS compilers store these tables in the data segment. On the other hand, SPARC compilers store them in the text segment, immediately following the indirect jump that uses them. Either approach, if consistently applied, enables an exec editor to find the first and last locations in a jump table easily.

On SPARC systems, a jump table usually starts immediately after its indirect jump instruction's delay slot. With MIPS compilers, the base address of a jump table is found by examining the instructions immediately before the indirect jump. These instructions load the address, which is determined by the linker, into a register, where it is added to the offset in the table.

A jump table's extent can be found in one of two ways. The first is to examine instructions before the indirect jump to find a comparison that checks if the index expression is within the table's bounds. This comparison contains the table's size. This test can be found if a compiler generates stylized code for `switch` statements and if the instruction scheduler does

not greatly reorder the code. This approach consistently works on MIPS systems.

The other way to find the end of a table in text space is to scan it. An entry containing an invalid address in the program's text space is an instruction beyond the table (see Figure 10a). In compiled code, the restriction on addresses can be tightened to legal addresses in the current procedure. Call the table found this way, the *preliminary table*. An entry in this table could be both a legal instruction and legal address. The two can be distinguished, and the end of the table determined, by finding a jump address in the table (or a jump or branch instruction in the procedure) that lands on a preliminary table entry (see Figure 10b). This entry is the first block beyond the end of the table.

The other form of data in a text segment are literal constants. They typically appear immediately after a routine's instructions. These values may be difficult to distinguish from instructions and can cause the control-flow analyzer to append incorrect basic blocks to the CFG. However, these constants can be segregated from instructions when building the CFG. The analyzer constructing the CFG detects the last instruction in a routine, which typically is a return, and stops scanning at that point. However, a routine can contain more than one return instruction or can end with an unconditional backward branch, so the analyzer must examine the partially-constructed CFG to determine if control passes to an instruction after the last return or unconditional jump. If not, the instruction marks the end of the routine and everything that follows is data.

A special case is a "routine" consisting entirely of data (frequently a table of constants). This confusion arises because symbol tables record identifiers in the text segment and do not distinguish procedures from tables. In this case, an analyzer cannot even begin constructing a CFG, so the previous technique does not work. However, the control-flow analyzer can identify tables because: their name is known, their first words are invalid instructions, or they lack an essential attribute of a routine, such as a return or an interprocedural jump. The second test is facilitated by instruction set encodings, such as SPARC's, in which small integers correspond to invalid instructions.

3.5 Register Allocation

At high optimization levels, some compilers allocate registers interprocedurally, which violates the normal register-use conventions. In general, `qpt` is unconcerned with register-use conventions since instrumentation code saves and restores registers and does not affect a program's state. However, the cost of pushing and popping registers on the stack frequently exceeds the cost of instrumentation. `qpt` reduces this overhead on MIPS systems with *register scavenging*.⁵ While scanning instructions to construct a CFG, `qpt` notes the unused caller-saved registers in a procedure. These registers can be used by instrumentation code, without preserving their values, since the procedure's callers expect these registers to be modified.⁶

⁵Register scavenging is unnecessary on SPARC, since three global registers are deliberately left unused by the SPARC ABI (application binary interface). `qpt` checks that a program follows the SPARC register-use convention before using these registers.

⁶A more sophisticated version of this technique would look for caller-saved registers that are dead at each instrumentation point.

However, register scavenging depends on a program obeying the caller-save register-use convention. Interprocedural register-allocated code and some hand-written routines violate this convention by keeping a live value in caller-saved register across a call in which the callee does not modify the register. `qpt` must detect these violations and fall back on the more general code sequences that save and restore state. The MIPS symbol table provides part of the information necessary to detect violations. It records the source language of the file containing a procedure, so assembly code can be identified and treated as suspect. However the symbol table does not record a file's optimization level, so interprocedurally optimized code is difficult to detect. This omission is particularly frustrating as the symbol table records a file's debugging level. `qpt` resolves this problem with a command-line argument to indicate that a program was compiled at `-O3` or `-O4` (and so is interprocedurally register allocated).

3.6 Hidden Procedures and Multiple Entries

Symbol tables frequently do not record every procedure in a program. Local procedures are typically omitted. This omission complicates exec editing since the symbol table entries identify points at which to start construction CFGs. These *hidden routines* are discovered in two ways. The first is a call to a routine that is not in the symbol table. The other is when a procedure's CFG does not account for all of its space and its last instruction is followed by valid instructions. `qpt` adds hidden routines to its internal symbol table and instruments them like conventional routines. However, they lack meaningful names and their assigned names can complicate the process of reporting measurements.

Hand-written routines and Fortran code with `ENTRY` statements can contain multiple entry points. This practice is common for complementary numeric routines (such as *sine* and *cosine*), where one routine is a short stub that transforms its arguments and jumps into the other routine. Symbol tables do not record the alternative entry point, so `qpt`'s instrumentation code is typically slightly inaccurate for these routines. The problem could be corrected with an additional pass over a program to detect multiple entry points. `qpt` currently does not do this, but it will in the future.

3.7 Shared Libraries and Position-Independent Code

SunOS and other operating systems use shared libraries to reduce the size of executable files and the amount of memory consumed by multiple copies of library routines [23]. When a program begins execution, it dynamically loads the shared libraries into its address space. The unresolved references and missing code in a dynamically-linked program prevent `qpt` from instrumenting it. Conceptually, at least, it would not be difficult to instrument a dynamically-linked program by statically linking the libraries with `/lib/ld.so` and then rewriting the resulting complete program.

However, shared libraries rely on position-independent code (PIC) to reduce the work required for dynamic linking. `qpt` processes this code, even though it does not instrument dynamically-linked programs, since PIC libraries are used by statically-linked programs. PIC introduces three complications:

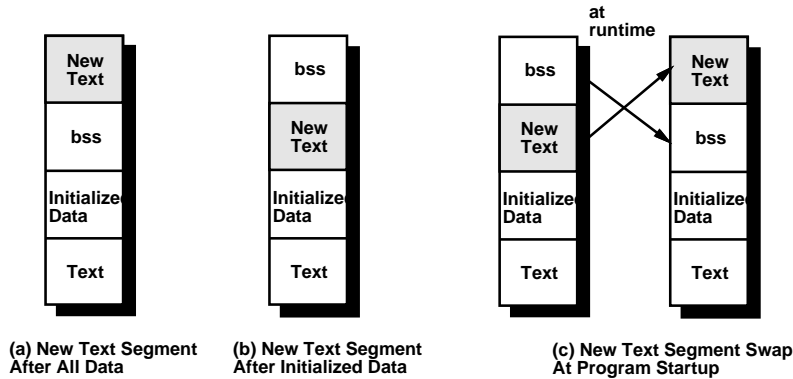


Figure 11: Location of instrumented text segment. The instrumented program (new text) should be placed after existing data (a). However, this placement requires the bss segment to be explicitly represented in the executable file. Placing the new text after the initialized data (b), minimizes the file’s size, but requires relocating all uninitialized data. `qpt`’s solution (c) is to use the approach (b) in the executable file, but to swap the new text and bss segments before the program begins so it appears to the program as if approach (a) was used.

- It frequently invokes a `call` instruction to compute an absolute address. The call’s return address determines the address of the call instruction, which can be used to compute other addresses. Fortunately, these calls have a stylized form:

```
call .+8
```

that is easily recognized. `qpt` disregards them since they do not alter control flow.

- On the SPARC systems, code for `switch` tables look different because the code contains a PC-relative address for the jump table. The solution is to modify `qpt` to look for the alternative code sequence and parse it correctly.
- PIC code uses trampolines to invoke dynamic linked routines. A *trampoline* is a short code sequence containing a relocatable jump to a routine. An application calls the trampoline, whose address is known at static-link time, and it, in turn, jumps to a dynamic routine. Since trampolines are stored in the data segment, they can easily be distinguished from true procedures. In addition, when constructing a call graph to report profile statistics or to generate the trace regeneration code, `qpt` must follow calls through trampolines to find the routines that are actually invoked.

3.8 Abutting Text and Data

On MIPS systems, text and data segments are widely separated in a program’s address space, so `qpt` can extend a program’s text space without running into its data segment addresses. On SPARC processors, the two segments abut so the text segment cannot expand. On these machines, the instrumented code must be placed in another part of the address space.

A logical place is immediately following the data segment. However, the format of SunOS executable files complicates this placement.⁷ In these files, a data segment is divided into initialized data, which is directly represented in a file and `bss` or uninitialized data (see Figure 3) `bss` is implicitly represented by its length in the executable file. If `qpt` places the new code after the initialized data (Figure 11b), where it minimizes the size of the resulting executable file, it resides in addresses previously occupied by uninitialized data and requires substantial, and perhaps impossible, relocation of a program's data addresses. On the other hand, if `qpt` places the new code after the uninitialized data (Figure 11a), it forces the `bss` data to be represented explicitly in the executable, which can increase its size by an order of magnitude or more.

A practical solution combines the two approaches. `qpt` places the instrumented code after the initialized data in the executable, but as soon as the program starts executing, it copies the new code to locations above the `bss` data and clears the memory it previously occupied (Figure 11c). This process works well, but greatly complicates debugging since breakpoints cannot be set until the code and `bss` segments flip.

3.9 Startup and Termination

`qpt` adds instrumentation code that performs actions immediately before a program starts executing and just after it finishes running. `qpt`'s startup code allocates a buffer on top of the stack by moving the program's arguments and environment down the stack. The code then jumps to the normal startup routine, which invokes the program. New startup code is easy to add since executable files explicitly identify a program's entry point. `qpt` simply changes the entry point to be its new routine.

`qpt` termination code, which writes out the instrumentation buffer after a program finishes, is more difficult to install. A program terminates either because of an exception or an exit system call. Unix does not provide an exception-handling mechanism that would permit `qpt` to gain control reliably at errors. Consequently, programs that terminate abnormally cannot be fully profiled or traced. Normally, however, terminating programs invoke an exit system call. `qpt` installs a call on its termination routine immediately before an exit system call. `qpt` can statically determine whether a system call is `exit` by examining the instructions before the call to find the system call number loaded into an argument register. At calls for which this test is ambiguous, `qpt` inserts a short code sequence to check dynamically if the call is `exit`.

3.10 Back-Tracing

Some program measurements, such as quick program profiling, require a back-trace of all routines active at a point in the program's execution. In `qpt` this back-trace identifies basic blocks that do not satisfy Kirchoff's flow law because the count of their entry arcs is one greater than their exit arcs. This imbalance, which affects the quick profiling algorithm, can easily be rectified with a list of call sites active at exit (i.e., a back-trace). On SPARC

⁷This is not a problem in the Solaris OS, which uses the ELF file format, which allows any number of segment to be stored in the executable file in any order and mapped to any memory locations.

systems, a short code sequence, invoked immediately before the exit system call, collects and records a back-trace by walking up the stack.

On MIPS systems, it is extremely difficult to produce a back-trace from within a running program because each procedure stores its return address at a different offset in its stack frame. The symbol table records these offsets, but the table is unavailable during program execution. Instead, `qpt`'s exit code sequence dumps the active stack, so other tools can examine it in conjunction with the symbol table to compute the back-trace. In general, this approach works well since exit is rarely invoked from within deeply-nested procedures in a normally-terminating program. However, routines with large local variables (as frequently happens with Fortran programs) cause extraneous information to be dumped at a large cost in time and disk space. A better alternative would be to include the return address stack offset information in the instrumented program, so the PCs can be found at run time.

4 Recommendations

`qpt` and a number of other exec editing tools demonstrate that rewriting executables is both a practical and effective means of measuring program behavior. However, the discussion above shows that choices made by operating systems and compilers can inadvertently complicate the process. Fortunately, a few simple changes to compiler and executable file formats could greatly simplify the process of rewriting executables, at little or no cost to the rest of the system. The rest of this section briefly lists a few important changes of this type.

4.1 Separate Code and Data

The first change would be to separate instructions clearly from data, or at least, ensure that the two are clearly distinguishable. From the perspective of an exec editor, instructions belong in the text segment, data in the data segment. However, this distinction is not always practical for the reasons discussed above. In this case, data should be distinguished from instructions by labeling it with a symbol table entry that clearly identifies it as data, not instructions.

4.2 Executable File Library

MIPS systems provides a library of routines (`ldfcn`) to access information in an ECOFF executable file. This library hides much of the complexity of the ECOFF file format, which compactly stores detailed debugging information in a collection of interlinked symbol tables. Other vendors should emulate this library and provide a higher-level interface to their executable files than the common standard of providing only a description of its data structures.

For exec editing, however, `ldfcn` has two shortcomings. An easily correctable problem is that `ldfcn` provides no access to line number data information from an executable file. A more fundamental issue is that the library is oriented to programs that extract information from a file, rather than those that create a copy of the file with some modified information. The `ldfcn` iterators do not guarantee that objects are traversed in the order in which they

appear in a file, so an exec editor must traverse the symbol tables itself, to ensure that each record is processed in the correct order.

A promising step in this direction is the GNU project's BFD library, which is a machine and operating system independent library for reading and writing executable files [24]. This library is used in the GNU project's assemblers, linkers, and debuggers, and provides more than enough functionality for an exec editor.

4.3 Executable File Improvements

A few minor changes to an executable file's symbol table would eliminate many problems at little or no cost. The text and data segments should be separated in memory, so the text segment can expand without running into data. Alternatively, the approach taken in ELF of allowing a segment to be stored in non-contiguous pieces also alleviates the problem. The symbol table should record for every indirect jump whether it is part of a `switch` statement and, if so, the location of its jump table. In addition, authors of compilers and libraries should ensure that all procedures are recorded in the symbol table. The symbol table should also identify multiple entry points into a procedure. Finally, symbol tables should record the optimization level of each file. These changes, except the first, require only a minor expansion of the quantity of information in a table.

5 Conclusions

Instrumenting a program with small pieces of code is an effective way of monitoring program behavior or measuring program performance. Although instrumentation code can be added at many points during compilation, waiting until the end of the process and rewriting the executable file reduces the cost of measuring the program and exposes its entire code to instrumentation. Some tools, including MIP's `pixie` and the authors' `qp` and `qpt`, successfully use this approach to profile and trace programs. The cost of instrumenting an executable is inconsequential (less than 20 seconds on a modern workstation for most executables). The run-time overhead is dominated by the instrumentation code, not the additional checks and branches inserted by the exec editor.

However, rewriting an executable file requires that an exec editor find all procedures in the file and build accurate control-flow graphs. A few design decisions in operating systems and compilers inadvertently complicate this process. These choices include intermixing code and data, not identifying jump tables for `switch` statements, omitting procedure entries from the symbol table, violating register-use conventions without recording the fact, using stack formats that prevent run-time back tracing, and placing the data segment immediately after the text segment. Many of these decisions can be changed in a way that simplifies exec editing without affecting programs' execution cost or significantly increasing the size of executables.

Until then, measurement tools must deal with executables that have these problems. This paper described a number of techniques that permit exec editing of existing programs. These work-arounds are not complex and work properly in the vast majority of cases, but they complicate an exec editor, and since some are based on heuristics, they are not always guaranteed to work with future compilers. Making a few changes to executable file formats

and compiler would encourage the widespread use of this approach to program measurement and would lead to new tools that provide deeper insight into program performance.

Acknowledgements

Tony Laundrie's profiling program `bbp` identified many of the problems discussed above and provided the idea of using the old text segment as an indirect jump table. Jeff Hollingsworth and Brian Johnson provided many helpful comments on this paper.

References

- [1] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter Usenix Conference*, pages 1–12, January 1992.
- [2] S. C. Johnson. Postloading for fun and profit. In *Proceedings of the Winter 1990 USENIX Conference*, pages 325–330, January 1990.
- [3] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.
- [4] James R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software Practice & Experience*, 20(12):1241–1258, December 1990. UW CS TR 912.
- [5] James R. Larus. Efficient program tracing. *IEEE Computer*, 26(5):52–61, May 1993.
- [6] Donald E. Knuth. An empirical study of fortran programs. *Software Practice & Experience*, 1(2):105–133, 1971.
- [7] Manoj Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37(9):1088–1098, September 1988.
- [8] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, pages 338–352, June 1991.
- [9] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An execution profiler for modular programs. *Software Practice & Experience*, 13:671–685, 1983.
- [10] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, April 1990.
- [11] David W. Wall. Global register allocation at link time. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 264–275, June 1986.

- [12] David W. Wall and Michael L. Powell. The Mahler experience: Using an intermediate language as the machine description. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 100–109, October 1987.
- [13] Matt Bishop. Profiling under unix by patching. *Software Practice & Experience*, 17(10):729–739, October 1987.
- [14] Peter B. Kessler. Fast breakpoints: Design and implementation. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation (PLDI)*, pages 78–84, June 1990.
- [15] MIPS Computer Systems, Inc. *RISCompiler Languages Programmer's Guide*, December 1988.
- [16] David W. Wall. Systems for late code modification. Technical Report WRL Technical Note TN-19, Digital Equipment Corporation, Western Research Laboratory, 1990.
- [17] Gabriel M. Silberman and Kemal Ebcioglu. An architectural framework for supporting heterogeneous instruction-set architectures. *IEEE Computer*, 26(6):39–56, June 1993.
- [18] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.
- [19] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.
- [20] Ben J. Catanzaro, editor. *The SPARC Technical Papers*. Springer-Verlag, 1991.
- [21] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.
- [22] Olin Shivers. Control flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pages 164–174, June 1988.
- [23] Robert A. Gingell, Meng Lee, Xuong T. Dang, and Mark K. Weeks. Shared libraries in sunos, n.d.
- [24] Steve Chamberlain. *libbfd: The Binary File Descriptor Library*. Cygnus Support, bfd version 3.0 edition, April 1991.