Open access • Journal Article • DOI:10.1002/SPE.4380240204

Rewriting executable files to measure program behavior — Source link

James R. Larus, Thomas Ball

Institutions: University of Wisconsin-Madison

Published on: 01 Feb 1994 - Software - Practice and Experience (Wiley)

Topics: Executable, Instrumentation (computer programming), Compiler and Profiling (computer programming)

Rewriting Executable Files to Measure Program Behavior

James R. Larus
Thomas Ball

Technical Report #1083

March 1992

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

March 25, 1992

Abstract

Inserting instrumentation code in a program is an effective technique for measuring many aspects of program performance. The instrumentation code can be added at any stage of the compilation process by system tools such as the compiler or linker or by external tools that are part of a measurement system. For a variety of reasons, adding this code after the compilation process—by rewriting the executable file—is simpler and more generally useful than adding it earlier.

This paper describes the problems that arose in writing the tools qp and qpt, which add profiling and tracing code to programs on the MIPS and SPARC processors. Many of these problems could have been avoided with minor changes to compilers and executable files' symbol tables. These changes would simplify this technique for measuring program performance and make it more generally useful.

A program's performance often cannot be understood without the assistance of tools such as execution time profiler or cache simulators. Parallel programs' performance is even less comprehensible without the assistance of performance tools. Most performance tools make minor additions and modifications to the measured program. These changes add small bits of code (known as *instrumentation code*) to record the execution of program events or to collect and record data.

Tools can add instrumentation code at any stage during compilation (see Figure 1). Even before the process begins, a source-to-source transformer can add measurement code directly into a program's source [5, 6, 7]. Instrumentation of this type is appropriate for measuring source-level characteristics, such as the type of statements used.

During the first step in the process, a modified compiler can insert instrumentation code while compiling a program [4, 8]. If the source of a compiler is available, this approach can

*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.

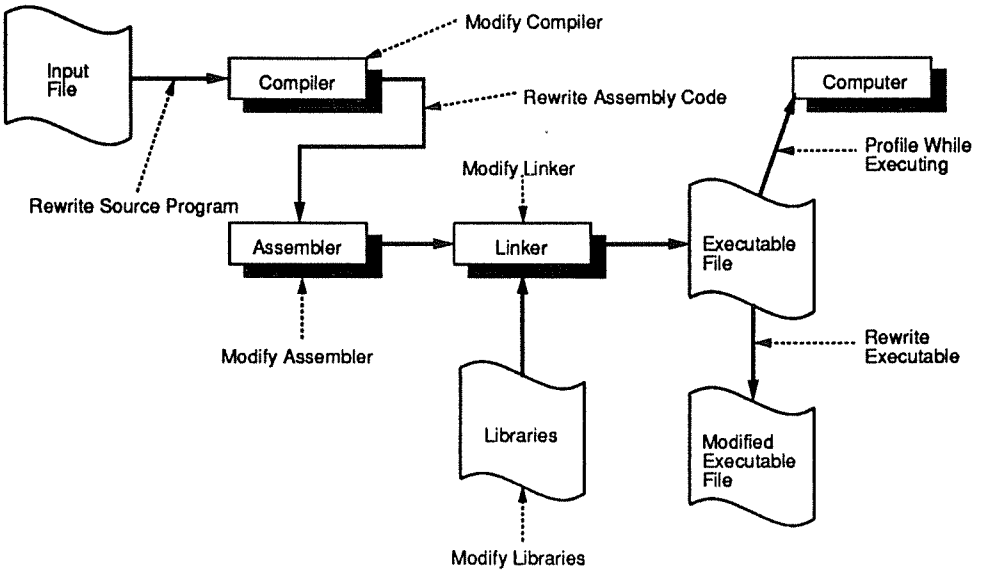

Figure 1: Options for instrumenting a computer program.

take advantage of compiler analysis to reduce measurement cost by placing instrumentation code intelligently [1, 8]. In addition, a compiler maintains mappings between syntactic and semantic structures and compiled code that are useful in placing instrumentation code and interpreting its measurements. However, modifying compilers has many disadvantages. The source of high-quality compilers typically is unavailable to third parties. To compound the problem, programs are written in a variety of languages, each of which has its own compiler, and use libraries that are supplied compiled. In addition, because compilers typically defer code generation until their final stage, an instrumenter only sees a compiler's intermediate representation, which blurs the association between instrumentation and particular instructions. The final disadvantage is the necessity of recompiling a program, which greatly increases the cost of measuring it.

After compilation, the effects of compiler optimization and code generation are visible and directly measurable. One possibility, which addresses most problems in the previous approach, is to rewrite the assembly language produced by a compiler before passing it to the assembler [9]. However, on some systems (most notably, MIPS), generating assembly language changes the compiler's behavior and prevents it from producing debugging information.

Similarly, if sources to the assembler or linker are available, either tool can insert instrumentation code. The linker has the additional advantage of processing an entire program, including libraries, so it can ensure that all modules are measured and can use interprocedural information [11]. By contrast, if a program is instrumented earlier, specially instrumented libraries may be necessary. Another advantage of using a linker is that some assemblers reorganize code to schedule

instructions—for example, the MIPS assembler. This optimization is not visible until the linking stage.

Programs can be dynamically instrumented with the process-control mechanism used by debuggers on subordinate processes [2]. However, the cost of this mechanism, which requires two process context switches for each interaction, is far too high to make this approach competitive.

The final alternative, which this paper advocates, is to insert instrumentation code following compilation by rewriting an executable file (`a.out`) to add instrumentation code. MIPS's tool `pixie` has used this approach with great success [10]. For convenience, we will call the process *rewriting an executable file* and the tool performing it an *exec editor*. This alternative has many advantages. An exec editor is largely independent of earlier stages in the compilation process. It does not require 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, rewriting executable files shares link-time instrumentation's advantages because it manipulates entire programs, including libraries.

Our experience in using this approach in `qp` and `qpt`, however, illustrates why the technique is not widely applied. `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 [1]. `qpt` is a modified version of `qp` that also traces a program's instruction and data references using the technique of abstract execution [8]. Abstract execution dramatically reduces the cost of program tracing and the size of trace files by factors of 50 or more. `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.

Rewriting an executable file is complicated by three outside factors: inappropriate compiler conventions, inadequate symbol table information, and awkward `a.out` structure. None of these factors is fatal. All can be ameliorated at the cost of additional time and complexity. The rest of the paper describes these problems and the techniques necessary to instrument `a.out` files on MIPS and SPARC systems. The paper divides into three parts. The first explains more details of `qp` and `qpt`, to provide a firm basis for understanding the instrumentation process. The second part describe problems caused by these factors and outlines our solutions to them. The final section suggests simple changes to compilers and `a.out` files that would greatly simplify this method of program instrumentation.

1 Description of QP and QPT

`qp` and `qpt` process each compiled routine from an `a.out` file in turn. They construct a control-flow graph for each routine, which identifies potential paths through the procedure. Next, they

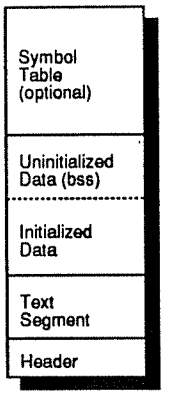

Figure 2: Organization of a Unix executable file.

compute the best locations for the instrumentation code and modify the existing code by inserting additional instructions and adjusting jump and branch offsets to accommodate new code. Finally, they write a new `a.out` file containing instrumented routines and an updated symbol table.

A Unix executable (`a.out`) file is composed of five major parts (see Figure 2). The first is a header that records the other pieces' size and locations within the file. The next is the *text segment*, which contains a program's executable code. The *data segment*, which contains statically-allocated data, follows the text segment. The data segment is composed of *initialized data* and *uninitialized data* (also known as *bss*). The latter is not explicitly represented in an `a.out` file. Instead, the file's header records the size of *bss* and its space is allocated when the program begins execution. Following the program is an optional symbol table that maps source program names and line numbers to instruction and data addresses. The detail and quality of information in this table varies widely, depending on the format of the `a.out` file and the optimization and debugging level at which a program was compiled.

`qp/qpt` first reads an `a.out` file's header and symbol table. At this stage, `qp/qpt` only must know the starting address of each procedure, but it also extracts information for later stages of the instrumentation process from the symbol table, to eliminate the need for a second pass over the table. If the file has been stripped (i.e., the symbol table removed), `qp/qpt` cannot process it and quits.

With procedure entry information, `qp/qpt` can construct a CFG (control-flow graph) for each procedure in a text segment. A *control-flow graph* is a common compiler data structure that concisely represents flow of control among instructions in a procedure. The nodes in a CFG are called *basic blocks*. They delimit straight-line, single-entry, single-exit sequences of instructions. The edges represent jumps between blocks or fall-throughs between consecutive blocks. `qp/qpt` constructs the CFG in two passes over a routine's instructions. The first pass examines each instruction to find jumps and, from them, the first and last instruction in each basic block. The

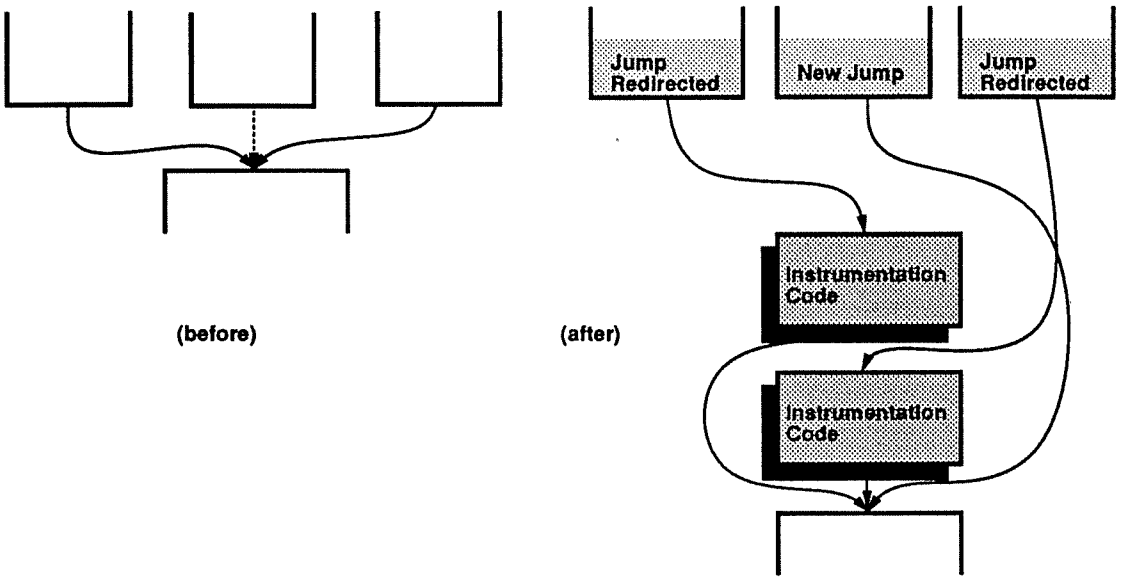

Figure 3: Instrumenting edges.

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 to categorize an instruction and to determine destination addresses for jumps.

`qp/qpt` uses the CFG to place instrumentation code in optimal locations [1]. The first step is to compute a weighting that assigns a likely execution frequency to each 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 a weighting, `qp/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 either record the number of times that they executed or the sequence in which they execute, depending on whether a program is being 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 `qp/qpt`'s innovations. We show elsewhere [1] that instrumentation code on edges is always as good, and frequently much better, than code limited to a block. In practical terms, it is only slightly more complex to place code along edges. The instrumentation code for edges coming into a block is placed immediately before a block (see Figure 3). The source of an incoming edge either jumps to the block or falls through from the previous block. In the former case, a jump can be redirected to either land on the instrumentation code or, if the edge is not measured, to bypass the code and go directly to the block. In the latter case, control can still fall through to instrumentation code, if the code for a

fall-through edge is placed first in the sequence of instrumentation code segments. If a fall-through edge is not instrumented, but other incoming edges are, a new jump is necessary to jump over the instrumentation code.

Obviously, inserting code into a compiled program changes the distance between instructions and requires adjustment of jumps. `qp/qpt` builds a map between the original and new location of each instruction for other purposes (see Section 2.2). The map itself is constructed in another pass over a routine by noting the size of the instrumentation code inserted between each pair of instructions. This map can be used to adjust branch offsets and jump target addresses when a modified routine is written out. The control-flow graph is not modified since it represents the original program.

`qpt` inserts instrumentation code within blocks to collect information necessary for tracing address references. This information takes two forms. The first is a value loaded from memory and the second is a result from a function call. This instrumentation code causes similar relocation problems that can also be resolved by the address map.

The final stage in processing a routine is to write it out. At this point, `qp/qpt` does not have a direct representation of an instrumented routine. Instead, annotations on both flow graph edges and blocks and in auxiliary tables describe the additional instrumentation code and the changes to existing instructions. `qp/qpt` writes an instrumented routine to a temporary file. After processing all routines, `qp/qpt` goes back and copies this file into a new copy of the executable file. At this point, `qp/qpt` knows the new address of every routine and is able to patch forward calls (i.e., to a later routine in the executable file) that were left unresolved in the temporary file.

While writing a new executable file, `qp/qpt` is able to use its mappings to update symbol table information. Fixing this information facilitates debugging the instrumented executable files, particularly while developing the tool that performs the instrumentation. If the symbol table is properly updated, an instrumented program can still be debugged with symbolic debuggers.

2 Rewriting Executable Files

The process of rewriting an executable file is not as simple as the description above suggests. Quirks in every instruction set and `a.out` file format complicate the process. This section discusses complications that arose while instrumenting programs for the two popular RISC computers, MIPS and SPARC. Other computers have different problems, many of which can be solved by the same techniques used for these computers.

The MIPS and SPARC processors are both reduced instruction set computers (RISC). This type of processor offers many advantages for exec editing. The most important are a small instruction set and fixed-length instructions, both of which greatly simplify decoding instructions in a routine. Other characteristics of RISCs, most notably the exposed pipeline (i.e., delayed loads

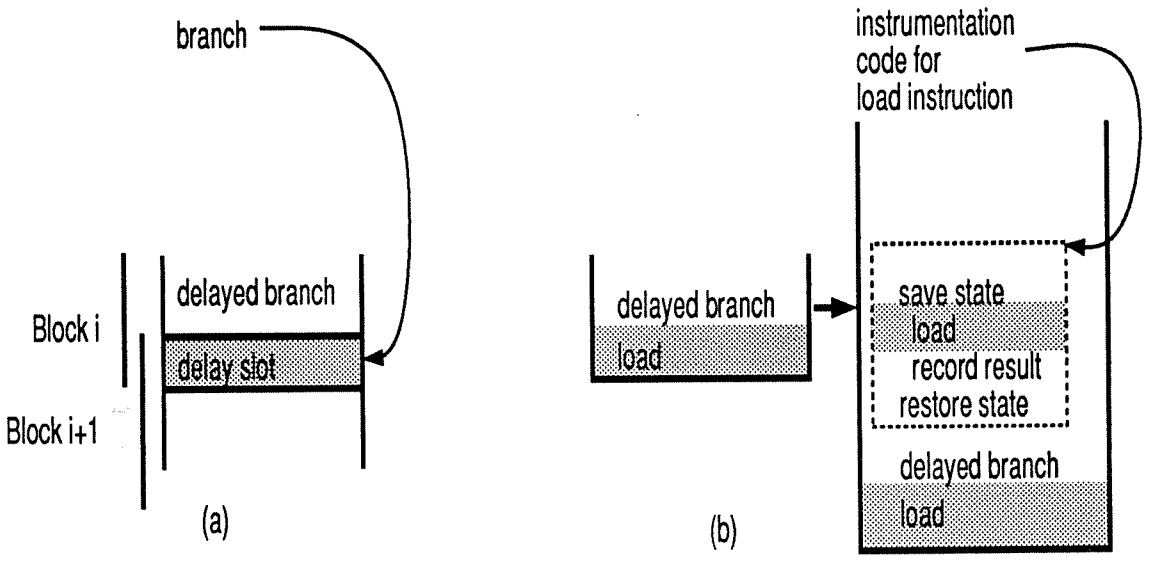

Figure 4: Control transfer to a delay slot.

and branches), complicate exec editing and are discussed below.

MIPS and SPARC processors uses different `a.out` file formats. MIPS uses a proprietary format called ECOFF, while Sun uses a derivative of the BSD `a.out` format for the SPARC. ECOFF is a more complex file format that contains significantly more precise debugging information. The additional complexity provides few benefits for `qp`, as it extract only three pieces of information from the MIPS symbol table: procedures' starting addresses, whether a procedure comes from an assembly-language file, and the register that holds a procedure's return address. `qpt` also uses the line number map. On the other hand, ECOFF's complexity does not adversely affect `qp`/`qpt`, mainly because of MIPS's `ldfcn` library, which hides many details of the data structures.

2.1 Delayed Instructions

In general, delayed control-transfer instructions (e.g., delayed branches and subroutine calls) cause no problems in constructing flow graphs. However, two uses of delayed instructions cause difficulties in instrumenting programs. The first occurs when an instruction in a *delay slot* (the window of instructions that execute after the jump, but before control transfers) belongs to two basic blocks. The other arises when recording the result of an instruction in a delay slot.

An instruction in a delay slot belongs to two basic blocks if it is the target of a jump (see Figure 4a). The delay slot instruction is both the last instruction of one block and the first instruction of the subsequent block. A compiler optimizer saves an instruction by fusing two blocks when the last instruction of a block is identical to the first instruction of the succeeding

block. The overlap complicates instrumentation since instrumentation code cannot fit between the blocks. The obvious solution, which works well, is to undo the minor optimization by duplicating the overlap instruction so each block has its own copy.

Unfortunately, delayed branches cause more serious problems when an instruction in a delay slot produces a value that must be recorded for an address trace (see Figure 4b). Traced values originate from either load instructions or function calls. A call from within a delay slot is ineffectual, so only delayed memory loads are traced. Instrumentation code to record the loaded value will not fit in the delay slot. An alternative is to move the load and instrumentation code immediately before the delayed branch. The load instruction, 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.

2.2 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 uses these jumps in a stylized manner that can be analyzed. Hand-written assembly code can use indirect jumps in a less controlled manner that makes control-flow analysis impossible. 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 constructs in commonly-used higher-level languages have an obvious analogue or implementation with indirect jumps (as opposed to indirect calls).[1]

Indirect jumps through jump tables are innocuous, since, when the table can be found (see Section 2.3), it contains the addresses of the jump's destinations. These addresses demarcate CFG edges. In addition, the table entries can be updated to redirect the jump to the new locations of the 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 when a `switch` statement is either hand-written or highly optimized, so the location of the jump table cannot be extracted from the instruction sequence. Without special attention, instrumented code would fail at the indirect jump because its destination is an address in the original, not instrumented, program.

[1] The exception to this rule is continuations in Scheme, ML, and other functional languages, which are just indirect jumps on a larger scale.

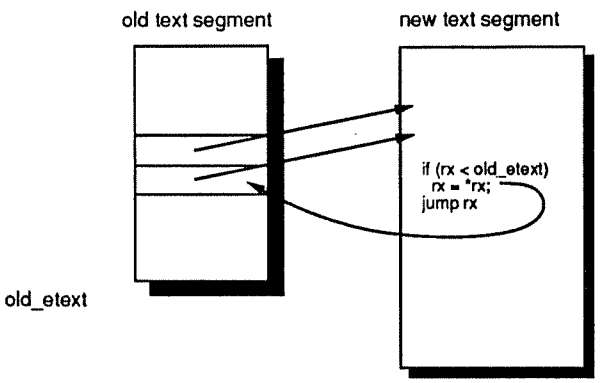

Figure 5: Translation table for indirect jumps.

To avoid this error, `qp/qpt` uses a program's original text segment as a translation table to map from addresses in the original program to addresses in the new program (see Figure 5). Before an indirect jump with an unknown destination, a small amount of code compares the jump address against the end of the old text segment. If the address is lower, the code dereferences the translation table to find the new target for the jump. The same mechanism permits `qp/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 for a signal. This address is a literal value that is difficult to recognize or translate in `qp/qpt`. However, `qp/qpt` easily recognizes a `sys_sigvec` system call and adds code before it to translate the function's address. The cost of a translation table is the extra memory required to store it. A translation table doubles the size of the original text segment, while instrumentation code expands it by 70%–200%, so the translation table requires about one-third of the instrumented text space.

2.3 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 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 construction of CFGs because distinguishing instructions from data is often difficult.

Data in a text segment occurs in two places (see Figure 6). The first is jump tables for `switch` statements. On MIPS, compilers place these tables in the data segment. On SPARC, they occur in the text segment immediately following the indirect jump that uses them. Either approach, if consistently applied, is acceptable, though placing the tables in the data segment

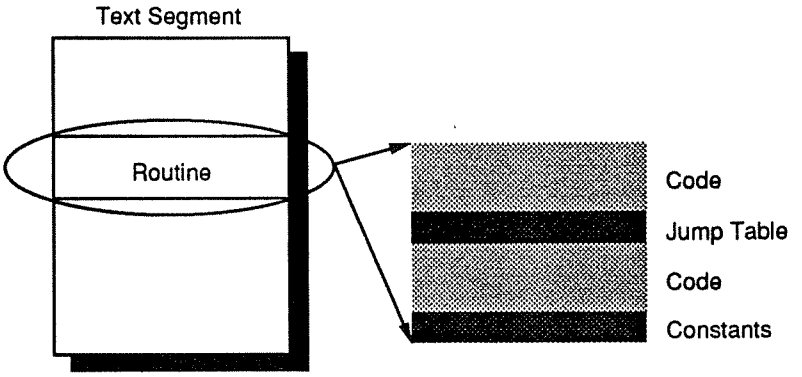

Figure 6: Data in text segment.

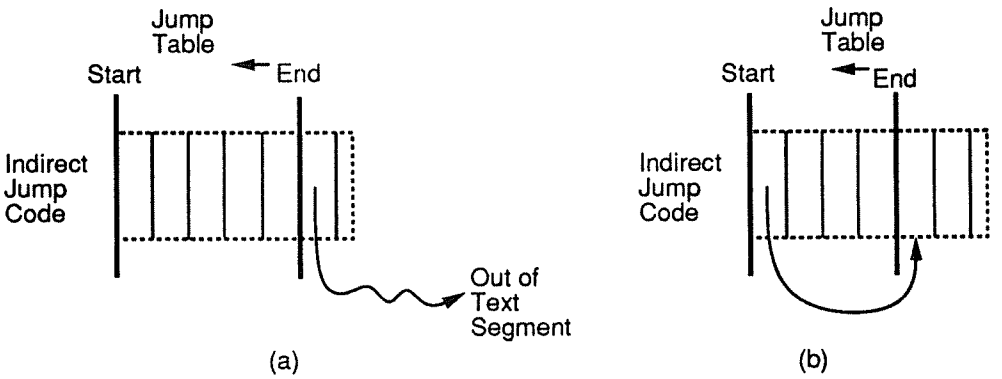

Figure 7: Finding end of a jump table.

slightly simplifies an exec editor. Finding the beginning of a consistently located table is easy. The table's extent can be found in one of two ways. The safest 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. It can be found if a compiler generates stylized code for `switch` statements and if the compiler's optimizer does not severely reorder the code. This approach consistently works for the MIPS.

The other way to find the end of a jump table in text space is to scan it. An entry that does not contain a legal address in the text space must be an instruction beyond the table (see Figure 7a). However, entries can be both legal instructions and legal instruction addresses. The two can be distinguished, and the actual end of the table determined, by finding a jump address in the table (or a jump instruction's target from the program) that lands on a table entry (see Figure 7b).

The other form of data in a text segment are literal constants. They typically appear immediately after a routine's instructions. These values are difficult to distinguish from instructions

and can cause a control-flow analyzer to construct incorrect basic blocks that contain invalid instructions. Constants can be segregated from instructions by carefully identifying basic blocks in a routine. The analyzer constructing these blocks must detect the last instruction in a routine, which typically is a return, and stop scanning at that point. However, as a routine can contain more than one return instruction or can end with an unconditional backward branch, the analyzer must examine the partially-constructed CFG to determine if control passes to an instruction following a return or unconditional jump. If not, the instruction marks the end of the routine.

A special case is a "routine" consisting entirely of data (frequently a table of constants). This confusion arises because symbol tables record names in the text segment and contain too little information to distinguish procedures from tables. In this case, an analyzer cannot even begin constructing a CFG, so the previous technique does not work. However, the analyzer can identify a table either because of its name's stylized form or because its first word is an invalid instruction. The latter test is facilitated by instruction set encodings, such as SPARC's, that make small integers correspond to invalid instructions.

2.4 Hidden Procedures and Entries

SPARC symbol tables occasionally do not record all procedure entry points in a library. Hand-written libraries contain internal routines omitted from the symbol table. These *hidden routines* are discovered in two ways. The first is at calls to routines that are not in the symbol table. The other is when a procedure's CFG does not account for all of its space and the last instruction is followed by valid instructions. `qp/qpt` adds hidden routines to its internal symbol table and instruments them like conventional routines. However, they lack meaningful names and their assigned names complicate reporting results.

Hand-written routines can also have multiple entry points. This practice is common for complementary numeric routines (such as sin 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 `qp/qpt`'s instrumentation code is typically slightly inaccurate for these routines. The problem could be corrected by making an additional pass over a program to detect these entries, but the expense is not worth the small benefit.

2.5 Register Allocation

At high optimization levels, some compilers allocate registers interprocedurally, which violates the register-use conventions of most programs. In general, `qp/qpt` is unconcerned with register-use conventions since instrumentation code sequences save and restore registers and do not affect a program's state. However, the cost of pushing and popping registers on the stack frequently exceeds

the instrumentation cost itself. qp/qpt reduces this overhead on MIPS with *register scavenging*.[2] While scanning instructions to construct a CFG, qp/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.[3]

However, register scavenging depends on a program obeying the caller-save register-use convention. Interprocedurally register allocated code and some hand-written routines violate this convention by holding a value in caller-saved register across a call in which the callee does not modify the register. qp/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 assembler 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. qp/qpt resolves this problem with a command-line argument that indicates whether a program was compiled at -O3 or -O4 (and is interprocedurally register allocated).

Interprocedural register allocation complicates address tracing in another way. Registers containing global variables are implicit parameters to routines and their values must be recorded upon function entry. Fortunately, register-allocated global variables appear as upwardly-exposed uses in the program slices used to compute address traces [8] and can be treated identically to other function arguments.

2.6 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 [3]. When a program begins execution, it dynamically loads the shared libraries into its address space. Currently, the unresolved references and missing code in a dynamically-linked program prevent qp/qpt from instrumenting it. Conceptually, at least, it would not be difficult to instrument a dynamically-linked program by 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. qp/qpt processes this code, even though they do not instrument dynamically-linked programs, since PIC libraries are used by statically-linked programs. On the SPARC, PIC

[2] Register scavenging is unnecessary on SPARC, since three global registers are deliberately left unused by the SPARC ABI (application binary interface). qp/qpt ensures that a program follows this convention before using these registers.

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

introduces three complications:

- It frequently invokes a `call` instruction to compute an absolute address. The return address for the call 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. `qp/qpt` disregards them since they do not alter control flow.

- Code sequences for switch tables look different because the code does not know the absolute address of the jump table. The solution is to modify `qp/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, `qp/qpt` must follow calls through trampolines to find the routines that are actually invoked.

2.7 Back-Tracing

Quick program profiling requires a back-trace of all routines active at an exit system call. This back-trace identifies basic blocks that do not satisfy Kirchoff's flow law because their entry arcs have a count 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 systems, a short code sequence, invoked immediately before the exit system call, collects and records a back-trace.

On the other hand, on MIPS systems, it is extremely difficult to produce a back-trace from within a running program because each stack frame stores its return address at a different offset. The symbol table records these offsets, but the table is unavailable during program execution. Instead, the exit code sequence dumps all active stack frames, so other tools can examine them 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. An alternative would be to record the return address information in the instrumented program, so it could write a back-trace at runtime.

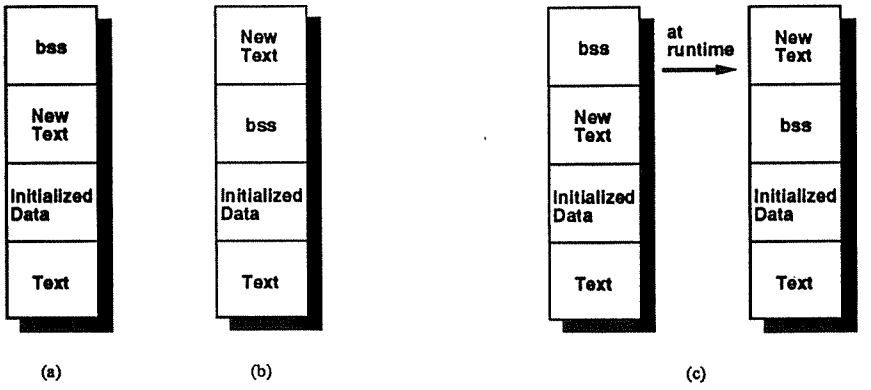

Figure 8: Location of new text segment.

2.8 Abutting Text and Data

On MIPS processors, text and data segments are widely separated in a program's address space, so `qp/qpt` can expand 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 (Figure 8a). However, the format of `a.out` files complicates this placement. 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 2) `bss` is implicitly represented by its length in the `a.out` file. If `qp/qpt` placed the new code after the initialized data, where it minimizes the size of the resulting `a.out` file, it resides in addresses previously occupied by uninitialized data and requires substantial, and perhaps impossible, modification to a program's data addresses. On the other hand, if `qp/qpt` placed the new code after the uninitialized data (Figure 8b), it forces the `bss` data to be represented explicitly in the `a.out` file, which can increase its size by an order of magnitude or more.

A practical solution combines the two approaches. `qp/qpt` places the instrumented code after the initialized data in the `a.out` file, 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 8c). This process works well, but greatly complicates debugging since breakpoints cannot be set until the code and `bss` segments flip.

2.9 Startup and Termination

`qp/qpt` adds instrumentation code that performs actions immediately before a program starts executing and just after it finishes running. `qp/qpt`'s startup code allocates a buffer on top of the stack by copying 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 `a.out` files explicitly identify a program's entry point. `qp/qpt` simply changes the entry point to be its new routine.

`qp/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 permits `qp/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. `qp/qpt` installs a call on its termination routine immediately before an exit system call. `qp/qpt` can 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 in which this test is ambiguous, `qp/qpt` inserts a short code sequence to check dynamically if the call is `exit`.

3 Recommendations

`pixie`, `qp`, and `qpt` demonstrate that rewriting executable files is both a practical and effective means of measuring program behavior. However, the discussion above also shows that choices inadvertently made by operating systems and compilers can significantly complicate the process. Fortunately, a few simple changes to compiler and `a.out` file formats would greatly simplify the process of rewriting executable files, 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.

3.1 Separate Code and Data

Perhaps the most important 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 reasons discussed above. In this case, data should be stored separately from instructions, either by placing it at the end of the text segment or strictly after each procedure's code. Intermixing instructions and jump tables complicates instrumentation unnecessarily. However, if data is stored in the text segment, the symbol table must be extended slightly to identify the portions of the text segment that contain data.

3.2 a.out Library

MIPS 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 `a.out` files than the common standard of providing only a description of its data structures.

For an exec editor, however, `ldfcn` has two shortcomings. An easily correctable problem is that `ldfcn` does not provide access to line number data information in an `a.out` file. A more fundamental problem 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.

3.3 a.out Improvements

A few minor changes to an `a.out` file's symbol table would eliminate many of the problems discussed above 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. 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 libraries should take care to ensure that all procedures in a library are recorded in its symbol table and that procedures with multiple entry points are distinguished by the table. Finally, symbol tables should record the optimization level of each file. These changes, at most, require a minor expansion of the quantity of information in a table.

4 Conclusions

Instrumenting a program with small amounts of monitoring code is an effective way of measuring a program's performance. Although instrumentation code can be added at many points during compilation, waiting until the end of the process and rewriting the resulting 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`, have successfully used this approach to profile and trace programs.

However, rewriting an executable file requires that an exec editor find all procedures in the file and build their control-flow graphs. A few design decisions inadvertently made by operating systems and compilers significantly 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 runtime back-tracing, and placing the data segment immediately after the text segment. Most of these decisions can be changed in a way that simplifies exec editing without affecting programs' execution cost or significantly increasing the size of `a.out` files. Making these

changes 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 provided many helpful comments on this paper.

References

[1] Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[2] Matt Bishop. Profiling Under UNIX by Patching. *Software Practice & Experience*, 17(10):729–739, October 1987.

[3] Robert A. Gingell, Meng Lee, Xuong T. Dang, and Mark K. Weeks. Shared Libraries in SunOS, n.d.

[4] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An Execution Profiler for Modular Programs. *Software Practice & Experience*, 13:671–685, 1983.

[5] 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*, pages 338–352, June 1991.

[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] James R. Larus. Abstract Execution: A Technique for Efficiently Tracing Programs. *Software Practice & Experience*, 20(12):1241–1258, December 1990.

[9] 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.

[10] MIPS Computer Systems, Inc. *RISCompiler Languages Programmer's Guide*, December 1988.

[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.