
INSTRUMENTATION TOOLS

Jim Pierce*, **Michael D. Smith[†]**, **Trevor Mudge***

**Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor*

*[†]Division of Applied Sciences
Harvard University, Massachusetts*

1 INTRODUCTION

The instrumentation of applications to generate run-time information and statistics is an important enabling technology for the development of tools that support the fast and accurate simulation of computer architectures. In addition, instrumentation tools play an equally important role in the optimization of applications, in the evaluation of new compilation algorithms, and in the analysis of operating system overhead. An instrumentation tool is capable of modifying a program under study so that essential dynamic information of interest is recorded while the program executes. The instrumentation process should not affect the original functionality of the test program, although it will slow down its operation. In a typical situation, a computer architect uses an instrumentation tool to produce an instruction or data trace of an application. The architect then feeds that trace to a trace-driven simulation program. The usefulness of instrumentation tools is obvious from a quick glance at current research publications in the area, where a significant number of authors use traces generated by two of the most popular instrumentation tools: *pixie* [23] and *spixtools* [6]. These tools are popular because of their applicability to many architectures and programs, their relatively low overhead, and their simplicity of use.

This chapter's focus is the design of instrumentation tools. Section 1 describes how instrumentation tools fit into the broad range of techniques available for the collection of run-time information. Section 2 lists the points in the compilation process at which we can instrument an application. It goes on to discuss the advantages and disadvantages of performing instrumentation at these points, noting that the basic structure of an instrumentation tool and the problems

faced are common to all of the approaches. Section 3 then discusses the specifics of instrumentation tool design, and Section 4 presents the important characteristics of some existing instrumentation tools. Finally, an appendix is included to illustrate the use of two existing instrumentation tools.

1.1 Methods for collecting run-time information

Before we discuss the design of instrumentation tools in detail, we first describe other approaches that provide the ability to collect run-time information. In general, we can classify a run-time data collection method as either a hardware-assisted or a software-only collection scheme. Each type of approach has advantages and disadvantages to consider.

A hardware-assisted collection scheme involves the use of hardware devices that are added to a system solely for the purpose of data collection. These monitoring devices are not necessary for the proper functioning of the computer system under test. Many different hardware methods exist for unobtrusively monitoring system-wide events. They include: 1) specially designed hardware boards, such as the BACH system [9], which observe and record bus activity; 2) off-computer logic analyzers, such as the University of Michigan's Monster system [21], that monitor the activity of the system bus; and 3) special on-chip logic, such as the performance monitoring counters on the DEC ALPHA 21064 microprocessor chip, which summarize specific run-time events [7]. The two main advantages of a hardware-assisted collection scheme are that one can build hardware to capture almost any type of event and that a hardware monitor can theoretically collect dynamic information without slowing down the application under test. Unfortunately, there are a number of disadvantages to these schemes too. First, since a huge amount of data can be gathered in a short time, the monitoring hardware is built either to summarize events (e.g., a counter that only counts the number of cache misses and not their addresses) or to record disjoint segments of program operation (e.g., a hardware monitor with a large memory that accepts the run-time information at the full execution rate and then later dumps this data to a backing store). In either case, less than the full amount of information is gathered which could lead to a distorted picture. To minimize the amount of unwanted data collected, researchers have combined hardware-assisted approaches with software instrumentation of applications to signal when the hardware should start and stop monitoring [21]—another compelling reason to understand software instrumentation methods. Finally, hardware-assisted collection schemes are costly and highly dependent

upon the characteristics of the monitored machine; thus, they are not a practical alternative for many users.

Software-only collection schemes, on the other hand, are relatively inexpensive and more portable than hardware-assisted collection schemes because the software schemes use only the existing hardware to gather the desired run-time information. In general, we can divide the software-only schemes into two approaches: 1) those which simulate, emulate, or translate the application code and 2) those which instrument the application code. Chapter 1 presented a detailed description of the first approach. Briefly, a code emulation tool is a program that simulates the hardware execution of the test program by fetching, decoding, and emulating the operation of each instruction in the test program. SPIM [12] and Shade [5] are examples of tools in this category. One of the major advantages of emulation tools is that they support cross-simulation and the ability to execute code on hardware that may not yet exist. Compared to instrumentation tools though, an emulated binary, even with sophisticated techniques such as dynamic cross-compilation [5], is noticeably slower than an instrumented binary when capturing the same run-time information.

An instrumentation tool works by rewriting the program that is the target of the study so that the desired run-time information is collected during its execution. The logical behavior of the target program is the same as it was without instrumentation, and the native hardware of the original application still executes the program, but data collection routines are invoked at the appropriate points in the target program's execution to record run-time information. Overall, researchers have proposed the following three distinct mechanisms to invoke the run-time data collection routines: microcode instrumentation, operating system (OS) trapping, and code instrumentation.

Agarwal, Sites, and Horowitz [1] describe a microcode-instrumentation technique called ATUM (Address Tracing Using Microcode) that supports the capture of application, operating system, interrupt routine, and multiprogramming address activity. Instead of instrumenting the individual applications, their technique "instruments" the microcode of the underlying machine so that the microcode routines record, in a reserved portion of main memory, each memory address touched by the processor. This approach is effective because, typically, only a small number of the microcode routines are responsible for the generation of all memory references. This approach is general because it is independent of the compiler, object code format, and operating system— as Agarwal states [2], ATUM is "tracing below the operating system." In fact, any information visible to the microcode can be instrumented. Agarwal, Sites, and Horowitz report that the overhead of this approach causes applications to

run about ten times slower than normal when used to collect address traces [1]. Of course, microcode instrumentation is only applicable to hardware platforms using microcode and even then the user must have the ability to modify the code. Furthermore, since most processors today have hardwired control, this approach has limited applicability.

A more widely applicable approach is to collect run-time information using OS traps. For instance, data address traces can be collected by replacing each memory operation in the target program with a breakpoint instruction which traps to a routine that records the effective address. A disadvantage of using OS traps is that, if many events must be recorded, the cumulative OS overhead of handling all the traps is significant. However, there are a number of exception mechanisms in operating systems that can be utilized to improve the efficiency of this method. Tapeworm II [28] is an example of an efficient software-based tool that drives cache and TLB simulations using information from kernel traps. It utilizes low-overhead exceptions and traps of relatively few events. The applicability and efficiency of the OS-trap approach depends upon the accessibility of certain OS primitives. With proprietary operating systems, this can be a problem.

The most generally applicable approach is the direct modification of the program's code. This approach, called instrumentation, inserts extra instructions into the target program to collect the desired run-time information. Data collection occurs with minimal overhead because the application runs in native mode with, at most, the overhead of a procedure call to invoke a data collection routine. Most instrumentation tools can create instrumented binaries that run at less than a ten-times slowdown in execution time when collecting an address trace. Some instrumentation tools such as QPT [18] rely on sophisticated analysis routines and post-processing tools to reduce this overhead even more. This approach is generally applicable because it is independent of the operating system and underlying hardware, it has been implemented on systems ranging from Intel architectures [22] to the DEC ALPHA architecture [18][24]. Furthermore, most code instrumentation tools require only the executables, not the sources files, so a user can instrument a wide range of programs.

There are a number of shortcomings to code instrumentation, however. It is most suited to the instrumentation of application programs. Furthermore, most code instrumentation tools only instrument single-process programs; kernel code references and multiple process interactions are not typically included. Therefore, address traces generated by these tools are often incomplete and of limited utility for TLB or cache simulations that require the monitoring of

system-wide events. Recently however, there have been tools written that do instrument kernel code and multitasking applications [4][8][19].

Overall, software-only collection schemes are less expensive to implement and easier to port from system to system than hardware-assisted schemes. Software-only schemes, however, do impose some overhead on the system under test and often are restricted in the type of runtime information that they can gather. Even so, the robustness and simplicity of code instrumentation tools makes them a popular choice by today's computer architects. The remainder of this chapter focuses on the design of code instrumentation tools.

2 WHEN TO INSTRUMENT CODE

Code instrumentation can be performed at any one of three points in the compilation process: after the executable is generated, during object linking, or during some stage of the source compilation process. Although different problems arise depending upon when the code is instrumented, the general procedure of instrumentation is the same at all levels. In general, code instrumentation involves four steps:

1. preparing the code for instrumentation - code extraction, disassembly, and/or structure analysis,
2. adding instrumentation code - selecting instrumentation points and inserting code to perform the run-time data collection,
3. updating the original code to reflect new code addition - reassembly, relocation information update, or control instruction target translation,
4. constructing the new executable.

We now turn to the issues involved in instrumenting code at the different stages in the compilation process.

2.1 Executable instrumentation

Instrumenting the executable or late code modification is of the greatest utility to the user. However, it is also the most difficult for the instrumentation

tool since code structure information is not available. The tool is responsible for recognizing and disassembling the code sections, instrumenting the code, and then relocating the code while rebuilding the executable. The missing information affects the tool's ability to perform all three actions. Without the structure information, the tool must invoke compiler knowledge or code structure heuristics to accomplish the tasks which can result in both performance and reliability problems. When the tool cannot accurately predict code behavior statically, runtime overhead is incurred to adjust the behavior during execution. In addition, instrumentation can fail or worse, produce incorrect code, due to invalid code structure assumptions. These issues will be discussed more fully in the next section.

Sophisticated tools which can overcome these obstacles present many advantages to the user such as the following:

- Source code independence - This makes to a wide range of programs available for tracing.
- Program generation independence - Binaries produced by different compilers of various languages can be instrumented.
- Automatic library module instrumentation - Full tracing of user-level execution is easy since statically linked library code is included in the executable.
- Fast and efficient - No source code recompilation or assembly is required. The user is not required to maintain instrumented library modules.
- Code creation details hidden - The user need not be familiar with the compile-assembly-link process needed to create the application. In particular, details such as the necessary library modules or flags, non-standard linking directives, or intermediate assembly code generation are of no concern.

Late code modification tools have various requirements for the information necessary in the binary file. The most general tools can instrument a stripped binary-a binary without a symbol table. At the other extreme are tools which require the compiler to include additional symbol table information. These tools usually require the source to have been compiled with the `-g` option which includes profile and debugging information in the symbol table. Late code modification tools exist for many microprocessors and several of them are discussed in Section 4.

2.2 Link-time instrumentation

If one is willing to give up source code independence, a convenient time to instrument a program is after the objects have been compiled but before the single executable has been created and the relocation and module information has been removed. Instrumentation can be done by a sophisticated linker which includes an object rewriter. During the linking process each object is passed to the rewriter which performs the necessary code modifications. It handles code and data relocation by just noting location changes in the object's relocation dictionary and symbol table. The modified objects are then passed back to the linker proper and are combined into one executable in the normal manner. Recompile of the source code is unnecessary. The presence of the relocation information and the symbol table make relocation straightforward. Postponing modification until the executable stage when this information is missing makes relocation much more difficult and sometimes impossible.

There are several tools which perform link-time modification. Mahler is a back-end code generator and linker for Titan, a DECWRL experimental workstation [30]. The module rewrite linker can perform intermodule register allocation, instruction pipeline scheduling, and the insertion of code for basic block counting and address trace generation. Code and data relocation is done as described above. Another tool, epoxie, relies on incremental linking which produces an executable containing a combined relocation dictionary and symbol table [29]. Its advantages over Mahler are that the standard linker can be used and data sections remain fixed so data relocation is not necessary. Epoxie produces address traces and block statistics. An extension of epoxie has been created by Chen which can instrument kernel-level code [4]. It is described in Section 4.8.

Link-time instrumentation is not automatic like late code modification and requires input from the user. The user must have the application object files and know the application's linking requirements. In addition, the source files are probably necessary to generate the object files.

2.3 Source code modification

The earliest time to instrument the code is while it is being compiled. This is also perhaps the most straightforward time since the tool has maximal knowledge about the code. Unfortunately, it has several drawbacks from the user perspective:

- Source files are required.
- Compiler limited - Most tools are either incorporated into one compiler or based upon a particular language or intermediate level generated by one compiler. This further restricts the traceable applications.
- Instrumentation speed - Each time the application is instrumented the source must be recompiled. This also implies that the user must be familiar with the application's compilation procedure.
- Limited code instrumentation - Library modules are not instrumented automatically because they are not included in the source files. It is possible to create separate instrumented copies of all library modules and link them to the instrumented source objects but this requires obtaining the module source code and maintaining multiple versions of modules. Kernel code is difficult if not impossible to instrument with this method.

A major advantage of source-level instrumentation is that the binary creation phase of the instrumentation is greatly simplified. Often the unmodified system assembler and linker can be used to create the binary. Furthermore, the large amount of information available at this stage permits types of instrumentation to be done which are not feasible at later times. For instance, most source-level tools take advantage of compiler control-flow knowledge to reduce the amount of instrumentation code. This reduces both the execution time and resulting trace size.

AE (Abstract Execution) is a tracing system developed by Larus and Ball which is incorporated as part of the Gnu C compiler [3]. Its goal is to generate very small traces which can be saved and then reused for multiple simulation runs. The modified compiler actually produces two executable programs. The first is the modified application. In addition to normal compilation, the compiler uses the notion of abstract execution to insert tracing code in the application code. Abstract execution is based upon control-flow tracing to reduce the amount of trace code necessary. The resulting trace produced by the modified application is only a tiny part of the full trace. This allows traces representing long execution runs to be saved on disk. The compiler also produces an application specific trace regeneration program. The regeneration program is a post-processing tool which accepts the compacted trace and outputs the full execution trace. The tracing overhead, including the cost of saving the compacted trace to disk, is 1-12 times the unmodified program's execution time [17].

MPtrace is a source-level instrumentation tool developed by Eggers et al. to generate shared-memory multiprocessor traces [8]. Their goals were to develop a tool which was highly portable, caused minimal trace dilation, and generated accurate traces, i.e., complete traces which closely resemble those gathered using non-intrusive techniques. Dilation describes the increases in execution time that result from code expansion due to instrumentation. Minimizing program dilation is critical in multiprocessor tracing since a change in execution time effects the coordination of multiple processes and thus the overall execution behavior of the program. Source-level instrumentation allows MPtrace to achieve those goals. MPtrace is more closely tied to a parallel C compiler than to an architecture. Thus, its portability depends upon the compiler's portability. MPtrace was initially created for Sequent ix86-based shared-memory systems and only twenty five percent of the tracing system was machine dependent, most of that being a description of the instruction set.

MPtrace attempts to limit execution time dilation by employing compiler flow analysis techniques to reduce the amount of added instrumentation code. It instruments the code by adding assembly instructions to the assembly-level output of the compiler which will produce a skeletal trace. At the same time, program details are encoded in a roadmap file used for later trace expansion. The modified assembly-level sources are assembled and linked using the respective unmodified system tools. A compacted trace is produced upon the execution of the instrumented application. Using a post-processing program and the roadmap file, the full multiprocessor trace can later be generated. MPtrace can achieve a time dilation of less than a factor of 3 but the usual execution time increase is around a factor of 10 [8]. Library module code is not traced.

In summary, there are three times at which code instrumentation can take place. Late code modification does not require source files, library code is automatically instrumented, and the binary creation details are hidden from the user. However, due to the lack of information available in the binary file, late code modification tools are the most complex and the resulting binaries can suffer performance and reliability problems. Link-time modification takes advantage of some code information to simplify binary creation. It retains use of the system linker, can instrument module code, but the application source is likely to be required. Finally, source-level instrumentation utilizes substantial code information to simplify the code instrumentation process and to produce complex traces. It requires application sources and usually more information from the user. Library module code is not easily instrumented. The remainder of the chapter will focus on late code modification tools.

3 HOW LATE CODE MODIFICATION TOOLS ARE BUILT

An instrumentation tool must insert tracing instructions into the executable without altering the logical behavior of the program. At no point can the added instructions change the program state. For trace generation, the events which need to be recorded are the execution of basic blocks and all data memory references. With this information, an execution profile, memory reference, or full execution trace can efficiently be produced. The usual way these events are recorded is by adding code segments prior to each event. The code stores the information in a trace buffer which is periodically checked during program execution and flushed to backing store when full. The four tasks of the instrumentation tool are to:

1. Find the section(s) of the executable file which contain code and disassemble them to obtain program structure information,
2. Insert instructions to record events thereby expanding the original code section,
3. Translate all addresses which were changed because of the code expansion,
4. Put parts back together to make a new executable.

The next four subsections describe the problems faced and the specific actions required of the tool during each of the above stages. The final subsection discusses some architectural properties which facilitate or frustrate late code instrumentation. To assist in describing problems and the methods used to overcome them, we use several existing instrumentation tools as examples: IDtrace for the Intel architecture, pixie for the MIPS architecture, and QPT for both MIPS and SPARC architectures. These tools will be discussed in detail in Section 4. IDtrace is used most often as an example due to the authors' familiarity with the tool. However, it should be noted that all late code instrumentation tools encounter similar instrumentation problems and rely on similar solutions.

3.1 Code extraction and disassembly

The first steps of the instrumentation tool are to locate and then disassemble the code sections of the executable. Unix executables come in a variety

of flavors: ELF, COFF, ECOFF and the BSD a.out format [10][14], but their structure is basically the same. They all begin with tables containing information such as the number, type and location of sections in the file, if and where the sections are to be loaded into memory, and where to begin program execution. Most executables contain one text section, one data section, and one BSS section. The text section contains code. The BSS section allocates space for uninitialized data and is actually empty in the file. Once the text section is located, it must be disassembled. During disassembly the code is split into basic blocks and a relocation table is created which stores the locations of these blocks. This table will be needed later to instrument the code and update the target addresses of control instructions. Since instructions will be inserted into the code, almost all instructions will have their location shifted in memory and so the branch and jump instruction targets must be translated to reflect this. For most instructions this is straightforward since the targets are known at instrumentation time. For data objects, however, address translation is difficult, and without the symbol table, impossible. It is important that all data locations remain unchanged during instrumentation. Therefore, data sections are not modified and are loaded into memory in their original positions.

In some cases, data can be found within the code segment, and this can present several problems for disassembly. There are two reasons a compiler might put non-instruction bytes in the text section. One is to insure constant data cannot be written and to allow the data to be shared by multiple processes. The other source of non-instruction bytes are in-lined indirect jump tables which are created by the compiler for switch or case statements. The obvious problem associated with data in the text section is that, without additional information, the disassembler treats the data words as instructions and tries to disassemble them. These “non-instructions” could mistakenly define basic blocks, be instrumented, or even be modified. Even if the data were not mistakenly modified by instrumentation, earlier code expansion would cause it to be moved within the section. As stated before, data addresses cannot be relocated so this cannot be allowed to happen. The solution is to create a new text section which contains the instrumented code and to treat the complete original text section as a data section. It might be thought that modifying or adding erroneous instructions would lead to incorrect execution. This will not happen because these “bogus” instructions will never be executed. Since control was never passed to data in the text section in the original program, control will not pass to the instrumented data in the new program.

Another, more subtle, problem is more serious and affects ISAs with variable-length instruction. It is highly likely that after a disassembler blindly disassembles through non-instruction bytes, it will be out of alignment with the

following real instruction bytes. For instance, suppose a disassembler creates meaningless instructions from a block of constant data and it needs one byte past the end of the data block to complete the last instruction. Again, these bogus instructions are of no concern because they will never get executed. However, because of the one byte used earlier, disassembly will be out of alignment with the beginning of the true instruction bytes after the constant data and will continue to generate meaningless instructions. To combat this problem, the disassembler must know where non-instruction bytes are located in the text section and skip over them. Constant data locations can be found in the symbol table but locations and sizes of jump tables can only be deduced by knowing compiler code generation behavior. Thus, instrumentation tools like IDtrace which run on ISAs with variable-length instruction must be compiler dependent and could require the executable to contain the symbol table to assist in disassembly. Fortunately for IDtrace, most compilers for the Intel architecture put constant data in the data section and the symbol table is not necessary. However, IDtrace's disassembler is compiler dependent and will not properly instrument programs with unrecognizable jump table code.

3.2 Code insertion

Once the code is disassembled, the instrumentation code is added in binary form since there is no later assembly phase. Actual code insertion is not difficult. The only requirement is that the added code cannot alter the functionality of the program. Most instrumentation tools add short code sequences at the beginning of each basic block. If a memory reference trace is required, instruction sequences are also added prior to each memory referencing instruction.

For instance, during profile instrumentation, IDtrace labels each basic block with a unique number. Instrumentation produces two new files: the new executable and a `.blk` file. The latter holds information about each block such as its size, beginning address, and label number. During runtime, an array exists in memory which holds the execution count of each block. A code sequence is inserted before each basic block which will increment the proper array position for that block. When the program exits, this array is dumped to the `.cnt` file. Figure 1 is an example of IDtrace basic block instrumentation code. The block count array variable that is incremented is checked for overflow, and the trace buffer is checked and emptied if close to full. Even though each count array entry is a 32-bit unsigned integer value, it could still overflow if the program were sufficiently long. Using a command line option, IDtrace adds code to check for

```

push status_flag_reg      ; save status flag register
push temp_reg             ; save temp register
temp_reg <- block_number  ; put block label in register
M[ctab+(4*temp_reg)]      ; update basic block execution
    <- M[ctab+(4*temp_reg)] + 1 ; count table
temp_reg <- tbuf_ptr
(temp_reg > tbuf_near_full) ; check if trace buffer is
if not goto END           ; nearly full
call flush_buffer         ; if full, flush trace buffer

END: pop temp_reg          ; restore temp register
     pop status_flag_reg   ; restore status flag register

```

Figure 1 Instrumentation code inserted before each basic block by IDtrace in profile mode.

overflow and do sequential saves to the `.cnt` file. This adds extra instructions to each basic block sequence and will slow execution.

Original Instruction

```
reg1 <- reg1 + M[reg2+100]
```

Instrumented Instruction

```

push status_flag_reg      ; save status flag register
push temp_reg1            ; save temp registers
push temp_reg2
temp_reg1 <- reg2+100      ; compute effective address
temp_reg2 <- trc_buf_ptr   ; load trace buffer pointer
M[temp_reg2] <- load_tag   ; record reference type tag
M[temp_reg2+1] <- temp_reg1 ; record reference address
trc_buf_ptr <- trc_buf_ptr + 5 ; step trace buffer pointer
pop temp_reg2             ; restore registers
pop temp_reg1
pop status_flag_reg       ; restore status flag register
reg1 <- reg1 + M[reg2+100] ; original instruction

```

Figure 2 Instrumentation code inserted before a data reference instruction by IDtrace in memory reference mode.

Memory reference code is similar. It calculates the effective address of the data reference and sends it to a trace buffer. Figure 2 shows the code added by IDtrace to record a data reference.

3.3 Address translation

As the new code is added to the instrumented text section, the control instruction targets must be translated. This is easy for conditional branches and most jump and call instructions because they contain either the absolute target address or its relative offset. Most tools create a relocation table to perform address translations during instrumentation. The table holds the original and corresponding new addresses of all control instructions and their targets. IDtrace accomplishes address translation using two code passes. During the first pass through the code, the original locations of all control instructions and their targets are entered into the table. During the second pass, instrumentation instructions are inserted in the code and the new addresses of the targets are added in the table. When a control instruction is encountered and the new location of target is already in the table (this would occur for a backward branch), the new relative distance can be calculated and entered in the instrumented code immediately. When a forward branch is encountered the new location of the target will not be in the table and the new location of the branch must be noted in the table. Later, when the target instruction is instrumented and its new location is known, the relative offset in the earlier branch instruction is adjusted.

Unfortunately, there are some control instructions for which the target cannot be calculated at instrumentation time. The most difficult ones to handle are indirect call instructions where the target address is found in a register or memory location. Since the data values are unknown during instrumentation, the target cannot be calculated. Furthermore, instrumentation does not affect data values, so execution of the unaltered instruction will produce the original target address rather than the new address. To maintain correct program behavior the address translation must be performed at runtime. As the code is being instrumented, a *translation table* is created which is a list of original and new address pairs corresponding to the beginning of each procedure. This table is included in the instrumented file and is loaded into memory at runtime. Each indirect call instruction is replaced by a group of instructions that computes the original target address and then passes this address to a *call-handling routine*. This routine performs a translation table lookup using the original target address to find the associated new address. If a target translation is found,

control is passed to the translated address and the indirect call works as intended. If, however, the target is not found, an error message is reported and execution halts.

Without the use of the symbol table, some heuristic is necessary to detect procedure beginnings. For example, IDtrace marks all instructions following a return or nop instruction as potential procedure beginnings. If the code contains procedures with other instructions endings or if the target of an indirect call is the middle of a procedure, the table lookup scheme will fail. Even if execution progresses correctly, this method incurs substantial runtime overhead for each indirect call executed and significant memory space is required to hold the table.

Indirect jump instructions also pose a translation problem but can be handled in a similar manner to indirect calls. The jump instruction is replaced by code which computes the original target address and passes the address to the runtime lookup routine. This scheme has two drawbacks however. One disadvantage is the increase in overhead due to more runtime translations. The other is that the translation table requires more entries. Not only procedure beginning addresses but all basic block beginning addresses must be included in the table. This increased table size requires more space and increases address lookup time.

If instrumentation can be based upon compiler code generation knowledge, indirect jumps can be handled in a more efficient manner. In compiled code, indirect jumps are used in two situations. One is in conjunction with a jump table produced for switch or case statements. A jump table is a list of absolute addresses and the target of the indirect jump is found by using a register value as an index into the table. If the jump table can be identified, the absolute addresses can be translated at instrumentation time and the unaltered indirect jump instruction will work correctly at runtime. IDtrace translates the jump table addresses during instrumentation since it can find the location and size of the jump tables during disassembly. The other use of indirect jumps is for procedure returns in many RISC processors, such as the MIPS and ALPHA architectures. These too can be translated during instrumentation if assumptions about the compiler are utilized. The discussion of nixie in Section 4.2 describes how this can be done.

QPT cleverly stores the translation table in the location of the original text section [18]. Instead of being an opcode, the word at the original instruction address is the translated address. This allows QPT to load a complete translation table, one which holds the translation for every original instruction

address, without using any additional memory or file space. This succeeds only because 1) there is not constant data in the text section, and 2) instructions are a fixed 4-byte length.

A final issue in branch translation is branch target distances. Some ISAs such as the Intel architecture, include both short and long target length branch instructions. Usually, code expansion moves the targets out of range of the original short branch instructions. The simplest solution is to convert all short branches to long branches. In MIPS code, all branches targets are 24 bit long but it is still possible for code expansion to push target distances beyond this distance. Pixie can adjust for this if `-branchcounts` is given as a command-line option.

3.4 Rebuilding the executable

After the code has been instrumented and target translation is completed, the file sections must be combined to make a new executable. There are now the original text, data, and BSS sections, a new text section, and some tables and buffer space. The original sections must be loaded into memory in their original locations since they contain data. The optimal solution would be to either extend the text section to include the new text and translation table or to create a new text section. Space would be added to the BSS section to include the trace and block count buffers. The executable file format tables would be updated to reflect these changes and to point to the new text section as the location to begin execution. For various reasons, the optimal solution is not possible on many platforms.

The main problem encountered is that many OS loaders do not make full use of the information found in the load format tables. Most formats allow the user to specify the number of text and data sections, the location of where they are to be loaded into memory, and at what address execution is to begin. Unfortunately, to facilitate faster loading, most OS loaders load an application's sections into memory in the same positions in which they reside in the file, ignoring the position information in the format tables. Furthermore, Unix System V loaders only accept one file structure. It must contain one text section, one data section, and one BSS section, in that order. Execution must begin at a fixed address in the text section. The data section must immediately follow the text section. Obviously, special tricks are required to create the new, instrumented binary.

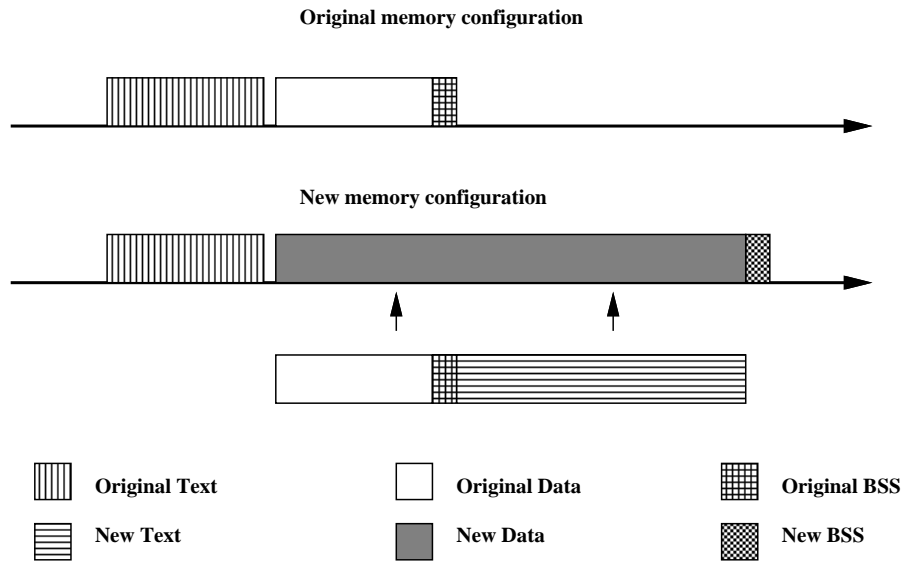


Figure 3 Original and new binary file configuration.

The solution used by IDtrace is illustrated in Figure 3. It combines the original data and the zero-filled BSS sections along with the new text section, trace buffer, and other tables into one big data section. Execution must begin in the original text section so the first few instructions there are modified to transfer control to the beginning of the new code found in the middle of the expanded data section. Another dummy BSS section is added to the end to satisfy the loader's requirement of one BSS section. Note that if the first instructions of the text section were not changed the program would run exactly as before since the original text and data sections are unmodified.

QPT has similar problems on SPARC processors because the text and data sections abut one another leaving no room to expand the text section. In this case, the QPT designers had two choices: add a new text section after the BSS section, which would require explicitly represent zero-filled data in the binary file; or, add a new text section between the data and BSS sections, which would create relocation problems with BSS data since the addresses of BSS data would then point to new text code. They compromised. The new text section is added between the data and BSS sections. Then, immediately upon execution, the new text copies itself to a location above the BSS data and zero fills the uninitialized memory space.

Rebuilding methods which expand the data space must allow for correct dynamic memory allocation (i.e., `malloc`). For example, on Intel platforms, the last address of the data space is stored in the `_curbrk` variable found in the application program. It is accessed by `sbrk`, a routine called by C's `malloc` function to position dynamically allocated memory. The `_curbrk` value must be updated with the last address of the expanded data space so that memory will not be allocated over top of the new code. IDtrace must know the location of `_curbrk` to make this change. Since IDtrace does not depend upon the symbol table, it finds the location of `_curbrk` by pattern matching disassembled instructions with the known `sbrk` instruction sequence. From those instructions, it extracts the location and updates `_curbrk` to reflect the data section's expanded size. If IDtrace cannot find `_curbrk` a warning message is produced. This is not always an error, however, since `_curbrk` is not included in all programs.

There are several other small issues which must be handled before the new binary will run correctly. First, the `exit` call must be modified so that the trace and basic block count buffers can be dumped to a file before control is returned to the OS. Most instrumentation tools modify the `exit` routine to call a new routine which performs these cleanup functions and then exits. The address of the exit procedure can be found in several ways:

- Lookup up the address in the symbol table. This method, of course, requires the binary to contain the symbol table.
- Pattern matching the disassembled code for the known sequence of exit procedure instructions. This method relies upon code knowledge.
- Knowing the location of a call to the `exit` procedure in the program and extracting the address from the instruction bytes. This is not too difficult because the initialization library code, `crt0.o`, contains an exit call and this code is always positioned at the beginning of the text section. This method also relies upon code knowledge.

The start code must also be modified to initialize instrumentation buffers and perhaps open trace files. If the OS loader cannot be told to begin execution at a non-default location, the original start code must also jump to the beginning of the new code section.

As the above sections have described, many problems are encountered when trying to modify an application at the executable stage. Actually inserting the

trace code is not nearly as difficult as translating control instruction targets and rebuilding the binary. Some tools rely on compiler-based assumptions to overcome these problems. Others require significant information in the symbol table. Still other tools, such as *pixie*, sacrifice execution efficiency in order to be almost compiler independent.

3.5 ISA properties

Some inherent architectural features simplify instrumentation. Others pose difficulties or add complexity to the resulting code. Some of these properties are discussed below. In general, RISC processor code is more easily instrumented and the resulting code is shorter and faster. However, some instrumentation problems are unique to RISC code.

Load-store vs. Memory-to-memory architectures

The major factor in the size and consequently the execution time of a program instrumented to trace memory references is the number of instructions requiring tracing code. Thus an instruction set that includes memory-to-memory operations such as the Intel architecture will have many more instructions to instrument than does a load-store architecture which usually retrieves operands from the register file. Memory-to-memory architectures often have a smaller register set which forces local variables to be stored in memory locations. Furthermore, memory operands can often be used as a source and destination in the same instruction thereby generating two trace entries from one instruction. All of these properties of memory-to-memory architectures contribute to the large size and runtime dilation of instrumented code. The i486 has approximately 180 instructions which can address memory. In addition, many of these instructions can perform both a load and a store and some non-string instructions reference two different addresses [13]. In contrast, the MIPS R3000 has only 14 instructions which can reference memory. Each can only perform a single read or write and no instruction can access more than one memory address [15].

Multi-reference instructions

Some processor instruction sets such as the i486 and the RS/6000 include string operations which can perform an indeterminate number of references per instruction. One example, in the i486 ISA, is the *rep* instruction prefix which

can cause one string instruction to repeatedly access sequential memory addresses until a condition is satisfied. It is impossible to ascertain the number of iterations at instrumentation time. To record an accurate reference trace, the single instruction must be replaced by a sequence of instructions which output the reference, perform the string operation, check the condition, and loop back if the condition is not satisfied. This emulation code adds to the size and execution time of the instrumented binary.

Register allocation

As seen in the sample code in Figure 1 and Figure 2, registers used in the trace code segments must be first saved and then restored so that the inserted trace code will not alter the current state of the application. If the processor has a large register set, tricks can be performed to eliminate these time consuming operations. For instance, pixie scans the original code prior to instrumentation and utilizes the three least referenced registers as dedicated instrumentation registers. The original code instructions which referenced these registers are replaced with memory referencing instructions. Pixie then uses the registers exclusively as instrumentation registers holding buffer pointers and effective address calculations. They are used in instrumentation segments throughout the program without having to continually save and restore their values [29]. QPT relies on the caller-save procedure register convention to scavenge instrumentation registers. QPT finds registers which were saved by the calling procedure but unused in the current procedure. This assumes that the program obeys the calling convention, and QPT tries to use symbol table information and optional command-line arguments to verify this. If it cannot be assured, the register values are saved and restored as described earlier. Because their target processors have 32 registers, pixie and QPT are able to contain code expansion.

Condition codes

Condition code values are part of the state of the computer and so should not be altered by actions in the tracing code. The Intel architecture has special instructions which push and pop the status flag register and these instructions are used by IDtrace to hide any effect the tracing code might have on the flags. The SPARC processor has four condition code registers. While the processor does not have user mode instructions which save and restore the registers, two types of arithmetic instructions are implemented: one which affects condition codes and one which does not. QPT's tracing code uses the non-affecting arithmetic instructions in all places except for the trace buffer overflow check.

In this case, it either inserts the check instructions where the condition codes are not live or performs the check with a more expensive code sequence which does not affect the codes.

Variable instruction lengths

Variable length instructions in combination with data located within the text section can wreak havoc with code disassembly. The disassembler must use information in the symbol table to skip constant data and use compiler specific knowledge to recognize and pass over jump tables. This was an unexpected and serious problem with IDtrace. Instruction length also affects the length of the output trace. When instructions are of uniform length, the trace need not contain the address of each instruction in order to quickly derive an execution trace. It is sufficient only to output each executed basic block beginning and data reference addresses. The position of data references relative to instruction references can be denoted using only a small integer offset. The offset represents the number of instructions executed since the last basic block beginning or data reference.

Delayed branches

Delayed branches in some RISC processors necessitate careful instrumentation. An instruction in a delayed branch slot succeeds a branch instruction in assembly code order but will get executed regardless of the branch direction. It is important that no instrumentation code get inserted between the branch and the delay slot instruction. The easiest way to handle this situation is to move any delay slot instruction which requires instrumentation to a location prior to the branch. It must be verified that this movement does not affect the outcome of the branch.

Indirect addressing

Finally, ISAs with heavy dependence upon indirect addressing will suffer from the overhead caused by the runtime address translation. In the MIPS architecture for instance, procedure returns are done with the jump register instruction (`jr`). The call instruction stores the return address in a general purpose register (usually `r31`) and `jr` indirectly finds the return address in that register. Thus, every return causes an address table lookup thereby adding to the execution time of the instrumented program. A method to avoid this overhead which is based upon compiler knowledge is described in Section 4.2.

4 CURRENT INSTRUMENTATION TOOLS

Late code instrumentation tools can be found for most of the popular current microprocessor. The following is a description of a selection of tools for use on various platforms.

4.1 IDtrace

IDtrace is an instrumentation tool for Intel architecture Unix platforms [22]. It instruments SysV R4 ELF binaries compiled using Intel/AT&T C, USL CCS C, and gcc compilers. Currently, it cannot automatically process code compiled by Intel's Proton compiler developed for the Pentium. IDtrace can produce a variety of trace types including profile, memory reference, and full execution traces. Primitive post-processing tools which read output files, view traces, and compute basic profile data are included in the IDtrace package. IDtrace can instrument stripped binaries, i.e., the symbol table is not needed. However, the executable must be statically linked and kernel code references are not included in the trace. Using full execution trace instrumentation, IDtrace will produce a executable which is about 5 times larger and runs 10-12 times slower than the original.

Primarily due to the need to recognize jump table code for disassembly purposes, IDtrace is compiler-dependent. To help alleviate problems due to non-compiler generated code, IDtrace can accept hints from the user on how to instrument a binary. The location or size of a jump table or the location of the beginning of a procedure are examples of such hints. IDtrace reads the hint information from an input file and uses it to assist in disassembling the code and translating addresses. As an example, execution of an instrumented program might abort with a message stating that a particular indirect call target address could not be translated at runtime. This could occur if IDtrace did not recognize the address as a procedure beginning and add it to the runtime transition table. The user could add this address to the hint file and reinstrument the program. IDtrace will then include the address and its translation in the translation table so that runtime lookup can occur during re-execution. While this process is tedious, it does allow the execution of handwritten or other non-compiled assembly code.

4.2 pixie and nixie

Pixie was the first binary instrumentation tool which received widespread use. Pixie is a full execution trace generation tool which runs on MIPS R2000, R3000 and R4000 based systems [23]. The tool is included in the performance/debugging software package of most systems based upon the MIPS architecture. Versions are available which instrument ECOFF and ELF file formats. With newer versions of pixie, if pixified dynamic libraries exist, they can be linked into the instrumented application to generate traces of dynamically-linked as well as statically linked code. Pixie does not, however, record kernel activity.

The default instrumentation option is to record only basic block execution counts. An informative post-processing tool, `pixstats`, can interpret the output to present a wide-array of runtime statistics. Using command line arguments, pixie will also instrument the application to produce an instruction and/or data trace. The reference trace output is written to a file descriptor. Using another tool called `makepipe`, the trace can be piped directory to a trace consumer program such as a memory simulator. Program expansion and time dilation depend upon the type of instrumentation used. When tracing both instruction and data references, the new executable is roughly 3 times larger and 4 to 5 times slower. The time dilation does not count the time required to save or pipe the trace.

Pixie is virtually compiler-independent. Constant data in the text section does not cause disassembly problems because the MIPS architecture has fixed-length instructions. It avoids having to recognize and decipher jump tables by performing all indirect jump address translations at runtime. Thus, switch generated indirect jumps, procedure returns effected by jump-to-register-value instructions, and indirect calls, all incur the overhead of a runtime table lookup to perform the target address translation. While pixie is not as restrictive as `IDtrace`, it does have some limitations. Like, `IDtrace`, it must use some heuristic to decide upon basic block separation. These heuristics are based upon MIPS compiler generated code. Hand assembled code could cause errors in separation and lead to inaccurate results. In addition, pixie cannot trace past fork calls and will fail on some special library routines.

In an attempt to lower the runtime overhead of pixie, another tool called `nixie` was created [29]. At the cost of becoming compiler-dependent and operating on a smaller set of application binaries, it makes assumptions about the binary code structure in order reduce runtime address translations. One of the

main sources of these translations is the use of indirect jump instruction, `jr`, to perform procedure returns in MIPS code. The compiler convention for a procedure call is to use `jal` or `jalr` and put the return address in `r31`. The return code convention is to use `jr r31`. Nixie avoids the runtime translation for the return by translating during instrumentation the return address found in the `jal` instruction. Then, nixie assumes that `jr` via `r31` is a return and the value in `r31` has already been translated. The `jalr` instructions are treated as indirect calls and are translated using the runtime lookup table as before. When the new address is found, the new return address is put in `r31`. The remaining `jr` instructions (the ones not using `r31`) are assumed to be indirect jumps produced by case or switch statements. Nixie recognizes the code patterns the compiler uses to begin a jump table and deciphers the size and memory location of the jump table. The entries in the table are translated at instrumentation time so they do not require runtime translation. The developers found about two dozen places in standard library code where the above assumptions were incorrect. Fixes for these exceptions were built into nixie so that most code can be instrumented without error.

Because nixie makes compiler-based assumptions about code structure, it can only instrument a subset of the pixie instrumentable applications. However, results from benchmark tests showed that the runtime of nixie instrumented binaries were up to 30% faster than pixie-instrumented ones [29].

4.3 Goblin

Goblin is a trace generation tool which instruments IBM RS/6000 applications [26]. It annotates code on the basic block level, i.e., code is added prior to each basic block to report block execution. Goblin has characteristics of both a late code and link-time modification tool. It accepts as input an executable with a detailed symbol table yet performs instrumentation separately on each object. The instrumented objects are reassembled and linked into a new executable by the system's assembler and linker programs. Goblin's first step is to use the descriptive symbol table to separate and disassemble the executable into assembly code objects. It then annotates the assembly code, records static data about the blocks in the objects, and updates the symbol table to reflect the instrumentation changes in each object. The regular system assembler and linker are then used to create an instrumented executable from the instrumented objects. The profile routines are introduced at the link stage as a profile library to be included in the image. The user can select different kinds output traces by linking in different trace libraries. Several libraries exist. One generates a

complete basic block trace. Another allows the generation of a full memory reference trace. Finally, since storage of large traces is difficult, there is library which performs on-the-fly basic block statistic calculations so that the whole trace need not be saved.

4.4 SpixTools

SpixTools comprises several programs that implement late-code modification of SPARC application binaries to produce instruction-level statistics [6]. The two main tools in the SpixTools distribution are `spix` and `spixstats`. `Spix` accepts an executable program and generates an instrumented executable. When run, this instrumented executable produces, in addition to its normal output, information indicating the number of times that each basic block in the original program was executed. By default, this information is directed to file descriptor 3, but the user can change this default through the use of the `-fd` option in `spix`. Unlike `pixie`, `spix` does not generate instruction or data traces; it only generates basic block counts¹

`Spixstats` uses the basic block counts to summarize the behavior of the instrumented program. This tool creates tables of (static and dynamic) opcode usage, branch and delay slot statistics, register and addressing mode usage, distribution of constants in immediate and displacement fields, and `gprof`-like per-function information. The ranking of functions is based on the total number of instructions executed in that function and not on the total number of cycles spent in that function. Exact cycle counts would require specific pipeline and memory system information which is not available to `spixstats`.

`Spix` handles the problems with executable instrumentation in similar fashion to the tools already discussed. For instance, when `spix` cannot correctly identify the targets of a register-indirect jump instruction, it simply has the instrumented executable print a diagnostic message indicating the address of the undiscovered target instruction and then terminate abnormally. Through the use of the `-jaddr` option in `spix`, the user then re-instruments the executable with this extra piece of information. This method is not unlike the hint information in the `IDtrace` approach. Furthermore, like the previous tools, `spix` works only with static code (no support for self-modifying code or dynamic libraries), and it is not capable of instrumenting the kernel.

¹Older versions of `spix` were capable of generating instruction and data traces. These capabilities have been removed since other SPARC tools (such as *Shade*) replaced them.

For the SPEC89 benchmarks, spix roughly quadruples the size of the executables. For the integer benchmarks where the average basic block size is small, the spix-instrumented executables run approximately 2.5-times slower. On the floating-point intensive benchmarks where instrumentation code execution can be overlapped with long latency floating-point operations and the basic block size is larger, the spix-instrumented executables run anywhere from 5% to 50% slower [6].

4.5 QPT

Like its predecessor AE [16], the design goal of QPT is to produce compact traces which can be stored for later simulations [18]. The difference between the two tools is that QPT instruments the executable while AE is part of a C compiler. This allows QPT to be applicable to many applications created by various compilers. As noted in the last section, QPT must overcome the disassembly and relocation obstacles common to all late code modification tools. In addition, QPT performs control flow analysis to reduce the amount of inserted tracing code. Therefore, it must rely heavily on symbol table information and code structure knowledge in order to reconstruct the exact code structure. QPT processes the code on a procedure basis. The address of each procedure is found in the symbol table and a control flow graph (CFG) is constructed with a basic block at each node. Using heuristics to decide the likeliest execution path, optimal code insertion points are located on CFG edges rather than nodes (blocks) and trace instructions are added to the original code.

The trace regeneration process is another unique feature of QPT. The trace output by the instrumented program is a compact trace which needs expansion before it can be used by a trace consumer program. Most tools supply statically created information files which can be read by a post-processor program to expand the trace. The AE system creates an application-dependent trace regeneration tool for each instrumented application. In both these cases the expanded trace would then be piped to the consumer program. QPT instead creates a regeneration program object file which can be linked into the compiled consumer program. Thus, the consumer program can read the compacted trace directly from disk [17].

The performance of the abstract execution instrumentation depends upon the regularity of the program's control flow and memory reference patterns. Numeric programs with sequential access patterns and few conditional branches require less instrumentation and therefore produce a more compact trace than

do non-numeric programs with more irregular behavior. Statistics reported by Larus in [17] show that the runtime of traced programs ranges from 1.4 to 12.3 times that of the non-traced program. These numbers include the time to store the trace to disk. The compact traces are between 13 and 250 times smaller than the expanded full execution trace. Larus states that regeneration costs are insignificant since the regeneration routine can produce the full trace at a rate of 200,000 to 500,000 addresses per second while most memory simulators consume addresses at the rate of tens of thousands per second. QPT does not currently instrument dynamically-linked shared libraries but could be modified to do so.

4.6 ATOM

ATOM [24] is a tool that allows the user to build his/her own customized instrumentation and analysis tools. For example, using ATOM, a few small C routines can be written to emulate the functionality of *pixie* and *pixstats* on a DEC ALPHA machine. On the other hand, if the trace information generated by *pixie* is not adequate, ATOM can be directed to gather and analyze a customized set of trace information.

Within ATOM, the authors have defined a set of instrumentation primitives common to all instrumentation programs. These primitives separate the tool-specific part of an instrumentation program from the common infrastructure required by all instrumentation tools. As a user, you write C routines using ATOM's instrumentation library which indicate the parts of the application program that interest you. For instance, ATOM provides library routines that allow you to have access to each procedure in an application, each basic block in that procedure, and each instruction in that basic block. By appropriately indicating where instrumentation code should go (e.g., before or after a particular set of program structures) and by indicating the particular information to be gathered at this instrumentation point, you can use ATOM to access to all of the dynamic information in an application.

In addition to instrumentation routines, an ATOM user can also write analysis routines (e.g., cache simulation routines that use the instrumentation data) that become part of instrumented program. In this way, both the instrumented code and the analysis code run in the same address space and thus experience a lower communication overhead of a simple procedure call rather necessitating context switching, file piping, or inter-process communication. The ATOM system guarantees correct operation by ensuring that the instrumented routines

and the analysis routines do not share library procedures or data. Still, the incorporation of the analysis routines into a single executable with the instrumented application program can perturb the output trace. For instance, if an analysis routine dynamically allocates memory, the trace of the heap addresses in an instrumented application will be different from the addresses used in the uninstrumented version of that application. ATOM employs several techniques and urges the user to avoid certain programming constructs to make certain that the behavior of the application is unchanged by the instrumentation and analysis routines.

ATOM is implemented on top of a link-time modification system called OM [25]. ATOM works by translating an ALPHA executable into OM's RISC-like symbolic intermediate representation. Through some extensions to OM, ATOM inserts instrumentation procedure calls at the appropriate points in the application code, optimizes the instrumentation interface, and translates the symbolic intermediate representation back into an ALPHA executable.

Since ATOM starts with an executable file, it can be considered to be a late-code modification tool. It, however, is not as robust an approach as a system like *pixie* since ATOM requires relocation information in the executable image in order to work. This relocation information simplifies the work required to adjust branch targets due to the insertion of instrumentation code.

Another advantage of the ATOM approach is that the underlying OM system can efficiently support an approach that does not steal registers from the application program. ATOM (like QPT and unlike *pixie*) uses the typical register save and restore mechanisms of a procedure call at each instrumentation site. This approach is desirable because it means that ATOM works on programs that use signals and `setjmp-program` features which are difficult to correctly handle under an approach that steals registers. The downside of a procedure call approach is that it incurs a greater overhead for each instrumentation action, especially if one does not have exact information on the register requirements of the instrumentation routines. Since the instrumentation routines can be quite complex in the ATOM system (remember that ATOM allows the user to use the instrumentation information immediately in an analysis routine), ATOM relies on sophisticated heuristics and techniques to reduce the procedure call overhead.

The performance of ATOM is related to the granularity of instrumentation and the complexity of the analysis routines. Srivastava and Eustace [24] report performance numbers for several different analysis tools built with ATOM. To summarize, for an analysis tool that instruments each memory reference and

simulates a direct-mapped 8 kilobyte cache, Srivastava and Eustace found that it took an average of approximately 120 seconds to instrument each program in the SPEC92 benchmark suite and that each instrumented program ran an average of nearly 12-times slower than the uninstrumented version. On the other hand, for an analysis tool that simply instrumented each system call site and summarized this information, they found that it still took only 120 seconds on average to instrument the SPEC92 suite but each instrumented program now ran only 1.01-times slower. Overall, ATOM is a powerful tool for building customized analysis programs.

4.7 Spike

Spike is an instrumentation tool which, like AE, was built into a compiler (GNU CC) [11]. Unlike AE, it is optimized for on-the-fly trace consumption rather than trace storage. This is performed by linking the original program with an instrumentation library. The library contains a procedure that is invoked for every trace event. This procedure can implement any kind of simulator or trace collector. In many ways, this is similar to ATOM.

Spike can trace data, instruction addresses, and an instruction behavior trace used for processor simulation. This last kind of trace is a dynamic list of *abstract machine architecture instructions*, or *amai*. Each *amai* is described by a type (e.g., integer add, floating-point multiply), and a list of source and destination operands. Any memory accessing instruction includes the memory address as well. The format and content of the *amai* are based on the RTL intermediate code language of the GNU C compiler.

Spike causes execution time dilation from a factor of 3–9 times. Because Spike operates on the compiler’s intermediate representation of a program, it is largely machine-independent. Spike has been implemented for the Motorola 68000 family, the SPARC, and the HP PA-RISC instruction set architectures.

4.8 Multitasking and kernel tracing tools

Most code instrumentation tools simply record user-level events within a single thread of control. Recently though, researchers have implemented tracing systems that extend existing code instrumentation tools so that they are able to capture multitasking traces and kernel actions. We briefly describe two such systems that illustrate the key issues related to the gathering of an accurate

interleaving of application and operating system reference traces within a multitasking environment. As will be seen, one could further extend these tools so that they could record other types of dynamic information.

The basic action of any multitasking tool is the sequenced collection of trace data from each instrumented application into a single global trace buffer. Recall that the act of instrumenting an individual application involves the placement of instrumentation code around the points of interest in the program and the inclusion of extra support routines which provide initialization, trace buffer management, and other support functions. In general, the instrumentation of each program in a multitasking workload is identical to the instrumentation of a single program except that the support routines change to reflect the management of the shared trace buffer. On the other hand, the trace of a multitasking workload is slightly different than the trace produced by a single application because the multitasking trace must include extra process information to distinguish the trace items of one process from the trace items of another process. For efficiency and practicality reasons, the existing multitasking tracing tools add extra support code into the operating system kernel to help gather this process information and ensure the consistent writing of the global trace buffer.

For the most part, the operating system is just another instrumented application. However, the portions of the operating system that are required to support the tracing system must be runnable with tracing turned off. The dumping of the global trace buffer to disk, for instance, is not part of the normal operation of the system and thus should not be traced. Furthermore, several portions of the operating system are too delicate to instrument automatically. For example, standard basic block instrumentation techniques will fail to instrument properly an operating system routine which flushes the CPU write buffer.

Chen [4] describes one such multitasking tracing tool based on the epoxie instrumentation tool [29] that modifies executables prior to linking. Chen's modified epoxie tool instruments code written for the MIPS instruction set architecture and thus, like pixie [23], uses register scavenging to select registers for use by the instrumentation code. Ideally, one would like to share the pointer into the global trace buffer indicating where the last trace item was written among all of the instrumented applications. Unfortunately, register scavenging precludes the direct mapping of a single global buffer into each application, since it is impossible to guarantee that one single register is available in all instrumented applications at all times. As a result, Chen's system maintains a trace buffer for each traced process, and at every entry into the kernel, the kernel copies the contents of the current process's trace buffer into the global trace buffer.

The tracing of system activity is more sensitive to software trace distortion than the user-level tracing of a single application. Chen's tool illustrates how one can minimize the problems of memory and time dilation. Even though *epixie* creates instrumented executables with very little code expansion due to its link-time optimizations, these instrumented executables are approximately 2-times larger and run approximately 15-times slower than the uninstrumented versions of the executables [4].

Compensation for memory dilation in *epixie* is accomplished in two ways. First, traces are collected on a system with a large amount of physical memory so that page misses due to limited memory capacity do not occur, and second, the traces are used to simulate the TLB behavior of an uninstrumented system. Time dilation is only partially compensated for; in particular, the rate of the system clock interrupt is reduced by 1/15, and the idle activity—the time spent in the operating system idle loop—is scaled by a factor of 15. These rough compensations are adequate because the research focus is on memory system behavior, and Chen claims that memory system behavior is largely unaffected by errors in these areas. The other operating system entity affected by time dilation is the process scheduler, and the effects of time dilation on scheduler policy is minimized by focusing on single-process and client-server workloads where context switches are driven by the applications and not by the scheduler policy. Similar techniques were employed by Agarwal [2] and Mogul and Borg [20].

Mazieres and Smith [19] describe another multitasking tracing tool based on the QPT instrumentation tool [18] that performs late code modification. Unlike Chen [4], their research is interested in the analysis and evaluation of I/O-bound applications such as network applications. Therefore, they organized their multitasking tool to reduce the effects of time dilation. Essentially, Mazieres and Smith attack the problem of time dilation in two ways. First, they chose QPT as their base instrumentation tools since it uses abstract execution [3] to minimize the amount of instrumentation overhead that occurs during the execution of an instrumented application. Second, they implemented their tool on a SPARC architecture where they could take advantage of several unused registers that are reserved by the SPARC ABI [27]. They use one of these reserved registers as the single, global, register-based, trace-buffer pointer that is shared by all instrumented executables. This decision removes the need for the copying of the per-process trace buffers into the global trace buffer as seen in Chen's system. They also describe a few other optimizations that have the potential to further reduce instrumentation overhead.

Overall, the systems by Chen and by Mazieres and Smith prove that it is possible to gather useful multitasking traces using code instrumentation techniques. However, there are several problems that make the gathering of accurate multitasking traces significantly more difficult than the gathering of a single application trace.

Exercises

- 4.1 Different computer architectures will schedule the same event at differing times. One goal of simulation is to determine the bottlenecks in this schedule. Based on this observation, consider the following statement: hardware-collected traces are more valuable than software-collected traces for simulation. Is this correct? Why or why not?
- 4.2 Many architectures provide a “block move” multi-reference instruction that copies one block of memory to another. An example would be:

```
copy.w R1, R2, R3          ; copy R3 words from M[R1] to M[R2]
```

This instruction poses a serious problem when creating a data trace (as described in Section 3.5: *Multi-reference instructions*). Only the starting addresses are found in the registers specified by the `copy` instruction, but this single instruction accesses the data cache many times. This chapter proposed changing the `copy` into a small loop to solve the problem.

Suggest an instrumentation method that does not require replacing the `copy` instruction. Put your answer in the form of pseudo code, such as Figures 1 and 2. (*Hint: You may consider assigning some of the work to the simulator, instead of the instrumentation tool.*)

- 4.3 Instrumented code runs slower than non-instrumented code. The slowdown is due to many factors. One is the execution time of the additional instructions. Explain two other, additional reasons for slowdown.
- 4.4 There are many solutions to the `_curbrk` dynamic memory allocation problem that IDtrace must face. Describe another solution besides the one that the designers of IDtrace developed. Compare your solution with theirs.
- 4.5 There are several reasons that gathering a trace in a multitasking environment is more difficult than in a normal, single-threaded environment. List two such reasons. Give an example for each where the normal, single-threaded approach breaks down.
- 4.6 Should compiler-based tools such as AE and Spike use the same solution to the address translation problem (see Section 3.3) as do late code modification tools? Explain why or why not.
- 4.7 IDtrace labels each basic block with a unique number. Explain how these numbers can be used to generate a trace of instruction addresses.

- 4.8 Operating system calls reveal much about a program: its I/O behavior, its use of system resources, etc. One method to obtain a system call trace is by use of OS traps. It is also possible to use software techniques alone.
- (a) Develop a software-only instrumentation technique to record system call events. (One detail that may help: Unix I/O system calls return the number of bytes read/written by the call in a pre-specified register.)
 - (b) Using the trace obtained in part (a), along with a trace of data and instruction address references, describe a technique to measure all I/O activity generated by a program. Be sure to consider *all* activity. (For simplicity, you may assume that only one process is executing on the system at a given time.)
- 4.9 Obtaining a trace of a real-time application, such as an interactive database or the kernel, is difficult with late code modification instrumentation techniques. One reason is the slowdown that these techniques incur interferes with the time-critical nature of the application. Explain how trace sampling can be incorporated to solve these problems (see Chapter 6: *Sampling for cache and processor simulation*). Be specific about the modifications to inserted instrumentation code that are required to implement sampling.
- 4.10 This chapter concerns itself with tracing compiled languages such as C and FORTRAN. Interpreted languages such as LISP or BASIC can also be traced by instrumenting the interpreter. Unfortunately, the same program will have considerably different traces when used with different interpreters. Develop an instrumentation technique that measures the data references due to the interpreted-language program itself, without measuring the extra data references generated by the interpreter.

REFERENCES

- [1] A. Agarwal, R. Sites, and M. Horowitz, "ATUM: A new technique for capturing address traces using microcode," Proceedings of 13th Annual Symposium on Computer Architecture, (Tokyo, Japan), Jun. 1986, pp. 119-127.
- [2] A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming*. Kluwer Academic Publishers: Norwell, MA, 1989.
- [3] T. Ball and J. Larus, "Optimally profiling and tracing programs," Proceedings of the 19th Annual Symposium on Principles of Programming Languages, Jan. 1992.
- [4] J. Chen, "The Impact of Software Structure and Policy on CPU and Memory System Performance," Technical Report CMU-CS-94-145, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1994.
- [5] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," Proceedings of 1994 SIGMETRICS Conference on Measurement and Modeling of Computer Systems, (Nashville, TN), May 1994, pp. 128-137.
- [6] B. Cmelik, "SpixTools Introduction and User's Manual," Technical Report SMLI TR-93-6, Sun Microsystems Laboratory, Mountain View, CA, Feb. 1993.
- [7] Digital Equipment Corp., *Alpha Architecture Handbook*, 1992.
- [8] S. Eggers, D. Keppel, E. Koldinger, and H. Levy, "Techniques for efficient inline tracing on a shared-memory multiprocessor," Proceedings of 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems, (Boulder, CO), May 1990, pp. 37-47.
- [9] K. Flanagan, K. Grimsrud, J. Archibald, B. Nelson, "BACH: BYU Address Collection Hardware," Technical Report TR-A150-92.1, Department of Electrical and Computer Engineering, Brigham Young University, Provo, UT, Jan. 1992.
- [10] G. Gircys, *Understanding and Using COFF*, O'Reilly & Associates, Sebastopol, CA.
- [11] M. Golden, "Issues in Trace Collection Through Program Instrumentation," MS Thesis, Department of Electrical and Computer Engineering, The University of Illinois, Urbana-Champaign, 1991.

- [12] J. Hennessy and D. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers: San Mateo, CA, 1993.
- [13] Intel Corp., *i486 Microprocessor Programmer's Reference Manual*, 1990.
- [14] Intel Corp., *UNIX System V Rel. 4.0 Programmer's Guide*, Order #465800-001, 1990.
- [15] Kane, Gerry, *MIPS R2000 RISC Architecture*, Prentice Hall: Englewood Cliffs, NJ, 1987.
- [16] J. Larus, "Abstract execution: A technique for efficiently tracing programs," *Software Practice and Experience*, Volume 20, Number 12, Dec. 1990, pp. 1241-1258.
- [17] J. Larus, "Efficient program tracing," *IEEE Computer*, Volume 26, Number 5, May 1993, pp. 52-60.
- [18] J. Larus and T. Ball, "Rewriting executable files to measure program behavior," *Software Practice and Experience*, Volume 24, Number 2, Feb. 1994, pp. 197-218.
- [19] D. Mazieres and M. Smith, "Abstract Execution in a Multitasking Environment," Technical Report 31-94, Center for Research in Computing Technology, Harvard University, Cambridge, MA, Nov. 1994.
- [20] J. C. Mogul and A. Borg, "The effect of context switches on cache performance," Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, (Santa Clara, CA), 1991, pp. 75-84.
- [21] R. Uhlig, D. Nagle, T. Stanley, T. Mudge, S. Sechrest and R. Brown, "Design tradeoff for software-managed TLBs," *ACM Transactions on Computer Systems*, Volume 12, Number 3, Aug. 1995, pp. 206-235.
- [22] J. Pierce and T. Mudge, "IDtrace: A Tracing Tool for i486 Simulation," Technical Report CSE-TR-203-94, Dept. of Electrical Engineering and Computer Science, University of Michigan, Jan. 1994.
- [23] M. Smith, "Tracing with Pixie," Technical Report CSL-TR-91-497, Center for Integrated Systems, Stanford University, Nov. 1991.
- [24] A. Srivastava and A. Eustace. "ATOM: A system for building customized program analysis tools," Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation, (Orlando, FL), Jun. 1994, pp. 196-205.

- [25] A. Srivastava and D. Wall, "A Practical System for Intermodular Code Optimization at Link-Time," Research Report 92/6, DEC Western Research Laboratory, Palo Alto, CA, Dec. 1992.
- [26] C. Stephens, B. Cogswell, J. Heinlein, G. Palmer, and J. Shen, "Instruction level profiling and evaluation of the IBM RS/6000," Proceedings of 18th Annual International Symposium on Computer Architecture, (Toronto, Canada), May 1991, pp. 180-189.
- [27] Sun Microsystems, *The Sparc Architecture Manual*, 1989.
- [28] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest, "Trap-driven simulation with Tapeworm II," Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, (San Jose, CA), Oct. 1994.
- [29] D. Wall, "Systems for late code modification," In *Code Generation-Concepts, Tools, Techniques*, Springer-Verlag, 1992, pp. 275-293.
- [30] D. Wall, "Link-Time Code Modification," Research Report 89/17, DEC Western Research Laboratory, Palo Alto, CA, Sept. 1989.

Appendix: Instrumentation Tool Use Examples

This appendix gives two examples of how late code modification tools can be used to gather dynamic information. We assume that the user is familiar with Unix and can create a statically-linked executable on a Unix system.

Runtime statistics

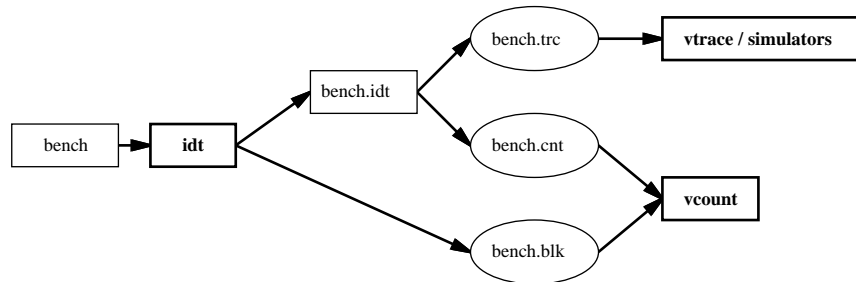


Figure 4 IDtrace programs and files – Rectangles are executables, ovals are data files produced by IDtrace, boldface names are IDtrace tools.

Suppose one wanted to compare the frequency of usage of certain instructions between several architectures. In particular, suppose one wanted to compare the most frequently used instructions in a typical RISC processor (R3000) with that of a CISC-like processor (i486). This could easily be done using two instrumentation tools: `pixie` on a MIPS R3000-based DECstation running Ultrix and IDtrace on a i486-based SysV Unix system. Suppose `cc1`, the major part of the C compiler `gcc`, is used as a benchmark program. The program `cc1` must be statically linked but neither the symbol table in the binary nor the sources are necessary. The steps required to use IDtrace are shown in Figure 4. The use of `pixie` is similar. First, we instrument the i486 version of `cc1` by typing

```
idt cc1
```

This will produce the instrumented binary `cc1.idt`, and the basic block information file `cc1.blk`. Then typing

```
cc1.idt stmt.i
```

will execute the instrumented version of `cc1` and also produce the basic block execution count file `cc1.cnt`. The post-processing tool `vcount` can then be run,

```
vcount cc1
```

to produce some basic runtime statistics. Part of the list of statistics is shown in Table 1.

Table 1 i486 profile information gathered using IDtrace and `vcount`.

Instruction Usage Percentage			Other Information	
<code>mov</code>	19,306,218	29.7%	Dynamic instruction count:	65,081,680
<code>cmp</code>	9,642,978	14.8%	Dunamic block count:	17,257,218
<code>push</code>	4,211,418	6.5%	Average inst. per block:	3.8
<code>je</code>	4,166,722	6.4%	Static block count:	41,807
<code>jne</code>	3,309,404	5.1%	Largest block (# of inst.):	95

Pixie works in a similar manner. First the executable is instrumented by typing

```
pixie cc1
```

which creates the files `cc1.pixie` and `cc1.Addrs`. Then the new program is run,

```
cc1.pixie stmt.i
```

to produce the `cc1.Counts` file. Finally, `pixstats` reads the output files to calculate an extensive list of runtime information part of which is shown in Table 2.

Memory simulation trace

Now suppose one needs memory reference traces for some type of memory system simulation. The method to generate the trace is similar to that explained above. To create a reference trace using `pixie`, type

```
pixie -idtrace cc1
```

Table 2 MIPS R3000 profile information gathered using pixie and pixstats on cc1.

Instruction Usage Percentage			Other Information
spec	27,615,307	33.19%	84,450,624 (1.015) cycles (3.38s @ 25.0MHz)
lw	13,027,613	15.66%	83,199,619 (1.000) instructions
addu	7,676,940	9.23%	17,272,839 (0.208) basic blocks
addiu	7,363,426	8.85%	13,217,812 (0.159) branches
sw	7,357,767	8.84%	4.8 instructions per basic block 6.3 instructions per branch

which modifies the binary to record both instruction and data references. Using `-itrace` or `-dtrace` will give just instructions or just data respectively. Typing

```
idt -c cc1
```

will instrument an Intel architecture binary to record a cache line trace. In this trace, all data references will be output, but only one instruction reference will be output per cache line. This reduces the number of instruction reference entries which must be recorded. The cache line size can be adjusted using the `-l` option. When `cc1.pixie` is executed, the trace is sent to a file descriptor. Using a program called `makepipe`, the trace can be piped directly to a cache simulator. IDtrace will send the output trace to a file, in this case `cc1.trc`. The trace can be send directly to a simulator by using standard csh pipe commands. Technical reports for both tools give trace format descriptions as well as complete descriptions of command-line options and trace piping methods [22][23].