Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor

Susan J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

While much current research concerns multiprocessor design, few traces of parallel programs are available for analyzing the effect of design tradeoffs. Existing trace collection methods have serious drawbacks: trap-driven methods often slow down program execution by more than 1000 times, significantly perturbing program behavior; microcode modification is faster, but the technique is neither general nor portable.

This paper describes a new tool, called MPTRACE, for collecting traces of multithreaded parallel programs executing on shared-memory multiprocessors. MPTRACE requires no hardware or microcode modification; it collects complete program traces; it is portable; and it reduces execution-time dilation to less than a factor 3. MPTRACE is based on inline tracing, in which a program is automatically modified to produce trace information as it executes. We show how the use of compiler flow analysis techniques can reduce the amount of data collected and therefore the runtime dilation of the traced program. We also discuss problematic issues concerning buffering and writing of trace data on a multiprocessor.

This work was supported in part by the National Science Foundation under Grants No. CCR-8619663, CCR-8904190 and CCR-8907666, and by The Boeing Corporation.

1 Introduction

Trace-driven simulation is one of the most widely used techniques for architectural evaluation; in particular, it has been invaluable to analyses of cache memories [Smith 82, Hill 87, Przybylski et al. 88, Eggers & Katz 89]. The data with which a cache simulation is driven consists of a trace of all instruction and data references gathered from the execution of one or more application programs. Typically, this trace is gathered using an interpretive technique; a trap is taken preceding the execution of each instruction, the instruction is analyzed by software loaded with the program, and trace information about the instruction is written to a buffer.

While trap-oriented trace generation is sufficient for use in uniprocessors, its application for multiprocessor tracing is somewhat questionable. The problem is that traps and subsequent instruction analysis drastically slow the execution of a program, often by two or three orders of magnitude [Shustek 78, Wiecek 82]. This dilation is not important in the uniprocessor case, because the trace content is independent of the dilation factor; in other words, data gathered by the trap analysis is identical to data that would be gathered in real-time by a hardware monitor.

However, for multiprocessors, a drastic reduction in the execution speed of cooperating threads may completely change the behavior of the program. This change occurs because the additional time per instruction used in trace generation overwhelms the normal timing differences between the execution of different instructions (e.g., that caused by cache hits versus misses). Consequently, all instructions appear to take the same amount of time. The original, global (cross-thread) order of instruction execution is perturbed, altering the order in which threads gain access to shared data and process messages to each other.

A more efficient alternative to trap-oriented trace

generation is microcode modification on a microprogrammed computer. One such tool, called ATUM, has been used on both uniprocessors and multiprocessors [Agarwal et al. 86, Sites & Agarwal 88]. An advantage of microcode alteration is that all executing code, including the operating system, can be traced. An obvious disadvantage of this approach, however, is that many current RISC processors are not microprogrammed, and even on CISCs, access to microcode is typically restricted to the vendor. Even with the microcode approach, the dilation may be as high as 20, because a special microroutine is invoked on every reference [Agarwal et al. 86].

This paper describes a software tool, called MPTRACE, that provides another alternative for tracing the execution of multi-threaded applications on a shared-memory multiprocessor. Three goals of MPTRACE were (1) to produce a portable software trace tool for multiprocessors that requires no hardware monitoring or modification; (2) to reduce runtime dilation well below that of other software techniques, and even below that of the microcode schemes; and (3) to produce *accurate* traces. Accuracy means both that the trace should be a complete list of addresses referenced by the program and that the address trace should be as close as possible to the trace that would be obtained with a nonintrusive tracing technique. MPTRACE is based on inline tracing, a software approach that entails no hardware modification. The technique automatically modifies the assembly language version of the compiled target application, inserting code to collect traces. Thus, at execution time, the program traces itself.

The tracing code is highly portable; the machine-dependent portions of the system are limited to 25 percent of the code, over two thirds of which is a description of the instruction set. (While the current implementation of MPTRACE executes on the Sequent Symmetry [Lovett & Thakkar 88], which uses Intel 80386 processors, RISC machines will have much smaller descriptions, and therefore proportionately less machine-dependent code.) If the OS scheduler allows migration, then it must be modified so that when a process (thread) is rescheduled, the scheduler will drop a timestamp and processor number into the trace buffer. Such a change is typically only a few machine instructions. Execution dilation for trace generation with MPTRACE, excluding trace storage, is only 2 to 3 times normal execution time.

MPTRACE's low dilation is achieved through three means. First, as previously stated, inline tracing eliminates the costly breakpoint and instruction decoding used by trap-based trace generators. Second, MPTRACE uses compiler flow analysis techniques to reduce the number of program points that need to be instrumented and the amount of data that is collected. Third, MPTRACE incorporates a trace postprocessor that reconstructs a full trace from the original source and the dynamic trace information.

The basic approach, inline tracing, has been used in other tracing and performance tools. Our technique differs from these systems in both the application targeted and the amount of dilation achieved. Pixie [MIPS 86] is a profiling tool developed to analyze basic block usage and execution time on the MIPS R2000. TRAPEDS [Stunkel & Fuchs 89] is a facility for performing trace-driven simulations on non-shared memory multicomputers (e.g., the Intel Hypercube). It modifies an assembly program, but inserts both tracing instructions *and* calls to a runtime simulator; the simulator processes and then disposes of the trace data. Dilation for TRAPEDS is 10 to 30 depending on the number of processors; the time does not include instruction simulation but does include a null procedure call where the simulation would occur. A similar system, based on link-time code modification, has been constructed for uniprocessor tracing and analysis on the TITAN [Borg et al. 89]. Its dilation, again excluding simulation, ranges from 8 to 12 times normal program execution. Finally, a similar technique has been used to implement a software instruction counter for debugging [Mellor-Crummey & LeBlanc 89].

The objective of this paper is both to describe MPTRACE and to evaluate the techniques it uses to reduce dilation. The rest of the paper is organized as follows. The next section presents an overview of MPTRACE, describing its operation and its processing phases. Section 3 describes the optimizations that are made to reduce tracing overhead. Section 4 discusses the buffering of trace data for I/O. Section 5 evaluates the performance of MPTRACE and shows the effect of several optimizations. Section 6 summarizes our results and discusses future directions. An example of MPTRACE-modified code is shown in Appendix A.

2 Overview of MPTRACE

MPTRACE consists of three phases:

- The *preprocessor* reads the assembly version of a source program and produces a modified source to be executed. The modifications include the insertion of tracing code and the cre-

ation of a data structure for identifying types of memory references. The latter is referred to as the *roadmap*.

- The modified program is then linked with MPTRACE run-time routines and is executed. The execution produces an encoded trace file that includes only that information that cannot be reconstructed at post-generation time, such as data references or control transfers based on runtime data.

- The *postprocessor* examines the encoded trace file, along with the roadmap provided by the preprocessor, to produce the final trace data.

We provide an overview of these three phases below.

2.1 Preprocessing

The principal objective of the preprocessing analysis is to minimize both the amount of trace code inserted and the amount of data produced by the modified program at run time. Both are needed to ensure a low dilation factor. Because of the detail involved in its source program analysis, the preprocessor is the most complex part of MPTRACE. Preprocessing a file is relatively quick. Compiling a C program and preprocessing the resulting assembly take about the same time.

The preprocessor has four main tasks. First, it saves the addresses of data and instructions in the unmodified source program. Although the source will be expanded by the addition of tracing instructions, addresses in the final trace file should reflect those in the original code.

Second, the preprocessor analyzes basic blocks to determine which instruction addresses and registers must be saved and then inserts tracing code to store them. (The locations of the inserted code in the original program are known as *save points*.) It attempts to minimize save point overhead by consolidating the inserted tracing instructions. For example, code for saving multiple registers appears contiguously, before any of the registers are used, rather than just prior to each separate use. All optimizations are described in section 3.

Third, the preprocessor generates information in the roadmap that will be used by the postprocessor to reconstruct the full trace from the generated, encoded trace. The roadmap contains the directions for generating memory references as they would have occurred during execution of the unmodified program. It aids the postprocessor in examining the encoded trace to follow control flow, determine how data is referenced, and compute its address. For example, given a particular instruction address, the roadmap specifies any static addresses referenced by that instruction, and defines the calculations needed to compute dynamic addresses, i.e., those based on run-time register values. The roadmap also provides size and access type information for data references.

Fourth, the preprocessor inserts code for managing buffers of trace output.

2.2 Trace Generation

The preprocessor-modified program is assembled and linked with run-time routines that handle trace output, and is then executed. The run-time routines preallocate a pool of shared buffers before calling the application's main program. When started, each executing thread of the parallel application program dequeues a buffer from the pool, fills the buffer with trace data, and places the full buffer on a queue to be written to disk. Separate writer threads empty the buffers and return them to the free pool for reuse. (The preprocessor inserts instructions in the program to check for buffer overflow, to dequeue empty buffers when needed, and to enqueue full buffers.)

2.3 Postprocessing

The postprocessor reconstructs all memory references generated by the original program. It steps through the roadmap code, interpreting directions for one instruction at a time, and retrieving values from the encoded trace file. The postprocessor adjusts all of the addresses placed in the final trace to remove the bias caused by inserting the tracing code. Postprocessing is the slowest of the three phases, generating about 3,000 addresses per second.

3 Optimizations for trace reduction

As previously stated, the primary objective of MPTRACE was to reduce the time and space requirements of the run-time tracing code. In this section we describe the analysis performed by our current implementation. In section 5 we discuss the space and time impact of some of these optimizations.

3.1 Basic and superblock analysis

Fundamental to reducing the amount of inserted trace code and saved trace data is the size of the program segment that the preprocessor analyzes at once. Since many of the particular optimizations described below can be performed only once per unit of analysis, it is important to choose as large a unit as possible. Our minimal unit of analysis is the basic block, a sequence of instructions with one entry point and one exit point. MPTRACE also attempts to identify sequences of contiguous basic blocks, which we call *superblocks*. A superblock has a single entry point but multiple exit points.

During trace generation MPTRACE saves only the first address in a superblock. All other instruction addresses can be inferred from information provided by the roadmap during postprocessing. Once inside a superblock, it is not necessary to mark the entrance into an internal basic block. Instead, after a branch instruction the postprocessor compares the branch target address to the next data element in the trace file. That element is either the target address, in which case the branch succeeded, or trace information from the next internal block, indicating that the branch failed.

The preprocessor also inserts code that checks for trace buffer overflow only once per superblock. The check takes into account the maximum size of all data that must be saved in the superblock.

3.2 Minimizing save points and save point code

Each save point has a prologue whose execution incurs overhead, e.g., for trace buffer pointer manipulation and the saving of processor state (described in section 3.4). This overhead can be minimized by reducing the number of save points, which MPTRACE does in several ways. First, all registers whose value remains unchanged are saved only once, rather than each time they are used. Second, multiple registers are saved at a single save point; the save point occurs before any of them are used. Finally, trace buffer overflow checks are folded into the first save point in a superblock, rather than appearing separately.

3.3 Minimizing saved operands

Because of basic and superblock optimizations, most of the trace data saved during program execution is for the computation of operand references. An important optimization is to remove tracing of operand references whose addresses can be determined at preprocessing time. For example, the locations of all global variables are fixed at compile time; therefore references to them can simply be inserted in the final trace by the postprocessor from information provided by the roadmap.

Many dynamically computed operand addresses can also be optimized. For example, on some machines, local variables are referenced as an offset to a fixed register (the frame pointer). Because this register does not change within a procedure, it needs to be saved only at procedure entry and exit.

Similarly, push and pop instructions adjust the stack by a known amount, typically indicated by the instruction opcode. Often these operations occur in series, particularly at procedure calls. Rather than saving the stack pointer before each separate operation, MPTRACE saves it for the first operation only and simulates each subsequent push or pop in the postprocessor.

While the stack pointer and frame pointer are special registers on some architectures, their optimizations are general in nature. The optimizations can be applied to any register that is both used for addressing and whose value is altered by a constant that is determined at compile time.

3.4 Register and condition code utilization

One problem with inline tracing is that executing the inserted trace code can perturb the processor state. For example, trace code often requires the use of registers, and tracing instructions can change condition codes that had been set by instructions in the original program. Therefore, the preprocessor must issue code to save processor state that is modified by tracing and required by a subsequent (original) instruction.

In the simplest and worst case solution, each save point within the modified program saves all registers and condition codes that it modifies. After the trace data has been written to the trace buffer, those registers and condition codes are restored. MPTRACE avoids much of this overhead by recognizing which registers and condition codes are live at save points and which are not. If a register is dead, it can be used by the tracing code. Similarly, if condition codes are not needed by the following instructions, they will not be saved.

4 Buffer Management

The basic function of the MPTRACE runtime system, which is linked with the modified application

program, is to provide empty buffers to the trace-generating threads and to write filled buffers to secondary storage. Buffer management is a significant problem on a high-speed multiprocessor; each traced thread on our Sequent Symmetry generates about 0.8 megabytes of data per second, while the I/O subsystem has a capacity of about 2 megabytes per second. When more than two threads are traced, the I/O subsystem becomes a bottleneck. Therefore, the two principal buffer management issues are (1) what to do when all buffers become full, and (2) how to reduce the rate at which this occurs.

4.1 Basic Strategy

The runtime system is started before the traced program so that it can preallocate the trace buffers. The buffers are placed in a shared pool, and the traced program is started. As the program executes, each thread pulls a buffer from the pool, fills it with trace data, enqueues it to a filled buffer queue, and repeats the process.

One or more writer threads (started by the runtime system) remove buffers from the filled queue, write them to disk, and return the buffers to the free list. The writer threads execute on separate processors from the trace-generating threads, so that their execution does not preempt tracing.

While the runtime system must have at least one writer thread, more writers permit more concurrency. For example, one writer can be blocked on I/O, while another is removing a buffer from the queue. If there are several physical disks and channels, multiple writer threads can be used to overlap I/O operations. With our current configuration, only two writers are required to saturate a disk. We allocate two writers to each of two channels, and run the I/O subsystem at its capacity.

Even with huge buffer allocations and multiple writers, it is inevitable that for some programs the tracing threads will eventually run out of buffers. Our approach is to stop all tracing as soon as one thread finds the buffer queue empty, even though other threads may still have partially empty buffers. We chose this approach, because it is consistent with our primary goal of reducing the dilation as seen by the tracing threads. Since all threads block almost simultaneously, the additional wait does not affect interthread timing, although it does increase the overall execution time of the experiment.

Inter-thread synchronization is currently done using a global flag. When a generating thread fails to find a free buffer, it sets the flag. Other generating threads poll the flag and spin while it is set. When all buffers are written out, the flag is cleared and the generating threads continue processing. Clearing the flag only after *all* buffers have been freed maximizes the time until the system must block again.

The polling code is inserted by the preprocessor at the head of every superblock; therefore the generating threads block only on superblock boundaries. On the average, each superblock contains either one or two basic blocks, and each basic block is approximately 4 instructions long. Thus, once the condition is detected, all threads will halt on average within 6 (original program) instructions. Polling at superblock granularity increases the execution time of a traced program by about 10 percent. A more efficient alternative would be to check the flag only at the end of each buffer. However, this option is too coarse for our dilation requirements, particularly for large buffer sizes.

4.2 Data Compression and Reduction

Since the I/O system is a substantial bottleneck, we have investigated ways to decrease the rate at which data is written. There are two principal alternatives for reducing the data rate. The first is to decrease the amount of trace data produced. As previously discussed, we limit the trace size by generating only those addresses that cannot be determined statically or through previously-saved values in the trace.

The second method is to apply a compression algorithm to the trace buffers before they are written to disk. If this process consumed no CPU time (i.e., if the compression were free), then an x-fold compression of the data would have the same effect as an x-fold increase in the I/O bandwidth or the amount of buffer space. Unfortunately, compression is not free; typically it involves additional copying, and consumes CPU, memory, and bus cycles. While this overhead can be overcome somewhat by using more processors, our experiments with the Unix compress utility lead us to believe that the CPU cost for algorithms providing adequate compression is an order of magnitude too high to significantly reduce buffer overflow.

4.3 Shared–Memory Multiprocessor Buffering Issues

The target application of MPTRACE was parallel programs, particularly those using lightweight threads. In such systems, buffering problems can be caused by the migration of threads between processors. The principal issue is determining how to

Program Name	Description	Processors	Language	Instructions (Static)
Pverify	logic verification	12	C	7395
Topopt	topological optimization	9	C	5916
Psim	butterfly simulator	12	PCP	7114
Psplice	circuit simulator	12	C	62838

Table 1: Programs used in MPTRACE evaluation

assign buffers. For example, buffers could be maintained on a per-processor basis. For cache simulations, this is a reasonable choice, since we generally wish to examine the reference stream generated by each processor. However this organization fails when threads are allowed to migrate. The problem originates because each executing thread will, during a save point, hold the address of its buffer in a general-purpose register. If the thread migrates in the middle of a save point, it could write trace data into the wrong buffer, possibly corrupting another thread's data. Synchronizing access to the buffer pointer before each save point would greatly increase the cost of tracing. Furthermore, if threads can switch buffers at any time, decoding the trace buffers at postprocessing time becomes more difficult.

To eliminate this problem we have chosen to manage buffers on a per-thread basis. However, we still need to know when threads migrate in order to reconstruct the per-processor reference stream. The solution requires some modification to the thread scheduler. When a migrated thread is scheduled for execution, the thread scheduler deposits a special token in the buffer along with the processor number, indicating that the thread has moved to a new processor. In this way a per-processor instruction stream can be pieced together, if desired. Many lightweight thread systems have user-level scheduling of threads (e.g.,[Bershad et al. 88]), which simplifies this modification.

This scheduler modification has not yet been implemented. For the parallel programs measured in the following section, buffers were allocated on a per thread basis; however threads in these applications do not migrate.

5 Performance of MPTRACE

This section presents measurements of the static code expansion and dynamic execution dilation caused by MPTRACE modifications of four parallel CAD applications. Code expansion indicates the growth in code size as measured by the number of additional instructions. Dynamic execution dilation indicates the change in execution speed when the modified program executes. In the dynamic measurements presented, writing of the trace buffers to disk was disabled. As previously discussed, all threads are stopped when buffers are full; our objective here is to measure the dilation only while the threads are executing. We purposely separate measurements of tracing from those including storage of trace buffers: the former reflects the capability of our techniques; the latter reflects the capacity of the underlying hardware configuration.

The four traced programs, which are listed in Table 1, were originally written for the Sequent Balance. Topopt [Devadas & Newton 87] does topological compaction of MOS circuits, using dynamic windowing and partitioning techniques; Topopt's algorithm is based on simulated annealing. Psplice is a circuit simulator combining the original direct method solution with waveform relaxation. Verify [Ma et al. 87] is a combinational logic verification program, that compares two different circuit implementations to determine whether they are functionally (Boolean) equivalent. The final program, Psim, simulates switch nodes for a non-shared bus interconnect. All programs were compiled, modified by MPTRACE, assembled by the Intel 80386 assemblers, and traced on a 20-processor Sequent Symmetry. These measurements only account for code in the application programs, and not in the standard Unix libraries, which were not traced. (MPTRACE can instrument libraries; however in these runs the libraries were left untraced for our convenience.) Of the four programs, only Psplice makes significant use of the libraries.

5.1 Effect of the optimizations

To allow the separate evaluation of individual optimizations, MPTRACE can selectively enable and disable some of them. Table 2 displays the result of disabling all switchable MPTRACE optimiza-

tions, thereby providing a "worst case" baseline for our implementation. With optimizations disabled, MPTRACE inserts an overflow check before each instruction, saves the address of each instruction, and saves the values of all registers used in addressing. (In fact, this is somewhat better than a worst case, because the preprocessor still recognizes and does not record references whose addresses are known at preprocessing time.) As the table shows, with optimizations removed, the instruction count grows by a factor of 20, while execution time grows by a factor of 8. Execution time dilation is much smaller than code size dilation, because many of the generated instructions are conditionally executed only when buffer overflow occurs.

Table 3 shows the effect of several optimizations on code growth. The three optimizations are added one at a time, i.e., each optimization includes the previous one. The first column of the table indicates the number of instructions in the original program. The next section, marked Basic Block, shows the code expansion when basic block recognition is enabled. In this case, the modified program saves only the starting address of each basic block, performs the buffer overflow check at the head of the block, saves registers only once where possible, and attempts to save as many registers in a save point as possible. The middle table section, marked Superblock, shows the effect of recognizing multiple basic blocks that can be grouped together. Superblock recognition performs the basic block optimizations, but over a larger section of code. Finally, the section marked Procedural saves the frame pointer on procedure entry and exit, rather than in each block in which it is used. In fact, the code grows slightly here, because the frame pointer is also saved at the head of each new buffer. The extra code to save the frame pointer is generated once per superblock but is executed only when a new buffer is required.

As we can see from the table, the effect of these optimizations when compared to the worst case of Table 2 is dramatic. Overall, code expansion is reduced by a factor of 3 to 5, depending on the optimizations applied and the application program. In general, programs with larger basic blocks and better register usage will see a lower code expansion from MPTRACE preprocessing.

Much of the code removed by the basic block optimizations is for buffer overflow checking and handling; this code is now applied only once per block. Because the overflow checks normally fail, the overflow handling code is executed infrequently. Table 4 reflects the expansion of code only along this "mainline" of execution, with the infrequently executed instructions removed. It provides a better indication of the typical code expansion that the program will see, rather than the total expansion of code for both the normal and exceptional cases. The savings is greatest for the Procedural optimizations, because the special code added for frame pointer processing is executed only when acquiring or releasing a buffer.

Tables 5 and 6 show the *runtime* dilation of the programs for each of the three sets of optimizations. In both cases, writing of buffers to disk was disabled. The execution times reported are the mean of ten executions of each of the programs. Table 5 shows the execution overhead of the traced programs without the overhead of trace buffer management. In this case, the buffer checking code was modified so that buffer overflow checks would always fail; that is, the exceptional code to enqueue a full buffer and dequeue a new empty buffer is never executed. In effect, this is the dilation that would occur with an infinite (or large enough) buffer. Table 6 shows the tracing dilation when buffers are managed at runtime, i.e., they are enqueued when full, although disk writes are still disabled. This is the tracing dilation that programs will more likely see when running. In this case the programs used 4K-byte buffers. In both cases dilation ranged between 1.6 and 3, with those including buffer management 10 to 15 percent higher than for the infinite buffer scenario.

Note that the programs run faster than the static dilation (code expansion) would indicate. There are several possible explanations. First, conditional branches that are added by preprocessing are seldom taken. Second, the tracing code uses a subset of instructions that have short execution times. Third, adding trace code increases the average basic block size. These factors may improve pipeline and cache performance.

5.2 Buffered I/O Performance

We have established that the tracing dilation of programs modified by MPTRACE is approximately 2 to 3, excluding trace storage. Furthermore, blocking the generating threads does not significantly affect dilation, because all threads stop within 5 or 6 instructions. However, when writing is enabled, the wall-clock time of a traced program averages about 10 times the execution time of the original, untraced program – or 4 times worse than the traced program without I/O.

There are at least three sources of the slowdown. First is the cost of copying data from user-space trace buffers to system I/O buffers before writing.

Program	Instruction Count			Execution Time		
	Original	Modified	Expansion	Original	Modified	Dilation
Pverify	7395	148328	20.1	13.6	119.1	8.8
Topopt	5916	118446	20.0	5.8	47.1	8.1
Psim	7636	152689	20.0	5.6	32.3	5.8
Psplice	62883	1265247	20.1	6.4	35.1	5.5

Table 2: "Worst case" performance (optimizations disabled)

Program	Instr	Basic Block		Super Block		Procedural	
		Instr.	Expansion	Instr.	Expansion	Instr.	Expansion
Pverify	7395	43631	5.9	38656	5.2	42281	5.7
Topopt	5916	37241	6.3	32359	5.5	34785	5.9
Psim	7636	42699	5.6	35327	4.6	39142	5.1
Psplice	62883	309723	4.9	273280	4.4	289330	4.6

Table 3: Evaluating optimizations: code expansion

Program	Instr.	Basic Block		Super Block		Procedural	
		Instr.	Expansion	Instr.	Expansion	Instr.	Expansion
Pverify	7395	33461	4.5	30274	4.1	29708	4.0
Topopt	5916	28247	4.8	25147	4.3	23967	4.1
Psim	7636	28866	3.8	26969	3.5	26605	3.5
Psplice	62883	245415	3.9	222454	3.5	213091	3.4

Table 4: Evaluating optimizations: "mainline" codew expansion

Program	Seconds	Basic Block		Super Block		Procedural	
		Seconds	Dilation	Seconds	Dilation	Seconds	Dilation
Pverify	13.6	34.3	2.5	32.2	2.4	31.4	2.3
Topopt	5.8	14.8	2.6	13.3	2.3	12.3	2.2
Psim	5.6	10.0	1.8	9.8	1.8	9.4	1.7
Psplice	6.4	11.2	1.8	10.7	1.7	10.5	1.6

Table 5: Execution dilation: no buffer management, no disk

Program	Seconds	Basic Block		Super Block		Procedural	
		Seconds	Dilation	Seconds	Dilation	Seconds	Dilation
Pverify	13.6	38.8	2.9	35.8	2.6	35.0	2.6
Topopt	5.8	15.9	2.7	15.0	2.6	13.9	2.4
Psim	5.6	11.7	2.1	11.1	2.0	10.9	1.9
Psplice	6.4	12.3	1.9	11.9	1.9	11.9	1.9

Table 6: Execution dilation: with buffer management, no writing

Second is the overhead of disk management and virtual memory management by the operating system, particularly as buffer space gets large. Third is the cost of the actual disk accesses. All of these can be reduced, but not eliminated, through additional hardware (memory or I/O) or more intelligent management by the application or operating system.

One optimization is to increase the size or number of trace buffers. We experimented with buffer allocation, varying buffer size from 256 to 64K bytes and numbers of buffers from 25 to 400. Our results indicate several trends:

- For small buffers (e.g., 256 bytes), buffer management overhead is high, because the cost of the system calls and run-time processing to manage the buffers dominates the time to fill them.
- For larger buffers (e.g., over 1K bytes), run time is dependent on the *total* buffer space allocation, but is independent of the actual number of buffers or their size.
- As total buffer allocation increases, so does memory management overhead. An increasing number of Dynix minor faults (those that cause a hardware fault but can be handled without disk I/O) are incurred by both the tracing threads and the writers, causing an increase in their execution times.

In our experiments, the wall-clock time for a traced program was consistently 10 times the execution time of the original program, generally independent of the buffer size or number of buffers, as stated above. However, for buffers as small as 256 bytes in size, or for total buffer allocation over 3 megabytes, the wall-clock time grew to nearly 15 times original program time.

6 Conclusion

We believe that the results shown in Section 5 demonstrate that the optimizations used by MPTRACE are useful in reducing dilation. The results also demonstrate the potential of inline tracing for trace generation on shared-memory multiprocessors. The execution-time dilations, which are only two to three times the execution time of the original program, are three orders of magnitude faster than trap-driven techniques, nearly one order of magnitude faster than microcode tracing, and 3 to 10 times faster than other in-line tracing facilities. Even including disk I/O, our system is comparable to or better than other schemes that only trace. Furthermore, MPTRACE produces a continuous trace by stopping all threads when buffers are full.

There are several reasons why MPTRACE is able to achieve its performance. First and foremost is the preprocessing of the assembly program to reduce both the amount of data that must be saved and the number of save points that must be inserted. Third, tracing utilized a subset of instructions that had shorter execution times than those used in the applications on average. This factor also contributes to the discrepancy between code expansion and runtime dilation.

We plan to take trace work in two different directions. First, we will produce multiprocessor traces for programs with fine-grained parallelism and fine-grained scheduling. Second, the effect of dilation on accuracy is still an open issue. We would like to test whether dilation affects tracing results. MPTRACE can introduce various amounts of dilation in a controlled fashion. We will thus be able to trace individual programs with various dilations but under otherwise identical circumstances.

7 Acknowledgements

We would like to thank Robert Henry for helping us to better understand code generation techniques. Jean-Loup Baer provided much useful feedback both during the project and on the paper. Tom Anderson and Brian Bershad helped with details of the Sequent and Presto.

References

[Agarwal et al. 86] A. Agarwal, R. L. Sites, and M. Horowitz. ATUM: A new technique for capturing address traces using microcode. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 119–127, June 1986.

[Bershad et al. 88] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A system for object-oriented parallel programming. *Software – Practice and Experience*, 18(8), August 1988.

[Borg et al. 89] A. Borg, R. Kessler, G. Lazana, and D. W. Wall. Long address traces from RISC machines: Generation and analysis. Technical Report 89/14, Digtal Equipment Corporation Western Research Laboratory, Palo Alto, CA, September 1989.

[Devadas & Newton 87] S. Devadas and A. Newton. Topological optimization of multiple level array logic. *IEEE Transactions on Computer-Aided Design*, November 1987.

[Eggers & Katz 89] S. J. Eggers and R. H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[Hill 87] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* PhD dissertation, University of California, Berkeley, November 1987.

[Lovett & Thakkar 88] R. Lovett and S. Thakkar. The Symmetry multiprocessory system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[Ma et al. 87] H.-K. T. Ma, S. Devadas, R. Wei, and A. Sangiovanni-Vincentelli. Logic verification algorithms and their parallel implementation. In *Proceedings of the 24th Design Automation Conference*, pages 283–290, November 1987.

[Mellor-Crummey & LeBlanc 89] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In *Proceedings of the 3rd Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78–86, April 1989.

[MIPS 86] MIPS. *Languages and Programmer's Manual.* MIPS Computer Systems, Inc., 1986.

[Przybylski et al. 88] S. Przybylski, M. Horowitz, and J. Hennessy. Performance tradeoffs in cache design. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 290–298, May 1988.

[Shustek 78] L. J. Shustek. *Analysis and Performance of Computer Instruction Sets.* PhD dissertation, Stanford University, January 1978.

[Sites & Agarwal 88] R. L. Sites and A. Agarwal. Multiprocessor cache analysis using ATUM. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 186–195, May 1988.

[Smith 82] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[Stunkel & Fuchs 89] C. B. Stunkel and W. K. Fuchs. TRAPEDS: Producing traces for multicomputers via execution driven simulation. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 70–78, May 1989.

[Wiecek 82] C. A. Wiecek. A case study of VAX-11 instruction set usage for compiler execution. In *Proceedings, Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 177–184, March 1982.

Appendix A: Example of MPTRACE-modified code

```
L0:
        cmpl    12(%ebp),%eax
        movl    %eax, (%ecx,%eax,4)
        movl    %eax, _b(,%eax,4)
        jg      L5
L1:
```

An assembly fragment to trace.

```
L0:
        cmpl    12(%ebp),%eax       # %ebp is a well-known register
                                    # .. thus need not be saved
# Added trace code
        pushfl                      # Flags need to be saved
        pushl   %ebx                # .. and need a register
.L0SPN: cmpl    $1,_BUF_writers_behind
        je      .L0SPN              # Check for writers behind
        movl    _BUF_ppbd+8,%ebx    # Check for buffer overflow
        subl    _BUF_ppbd+4,%ebx    # See if there is space for ALL
        cmpl    $12,%ebx            # saves in the superblock.
        jle     .L0JMP              # Jump to special code if overflow.
.L0RET: movl    _BUF_ppbd+4,%ebx
        movl    $0x10fc,(%ebx)      # Save addr of first instruction.
        movl    %eax,4(%ebx)        # And registers used in
        movl    %ecx,8(%ebx)        # .. forming addresses.
        addl    $12,_BUF_ppbd+4     # Adjust buffer pointer.
        popl    %ebx                # Restore saved registers
        popfl                       # .. and condition codes.
# End of added code

        movl    %eax,(%ecx,%eax,4)
        movl    %eax,_b(,%eax,4)
        jg      L5
L1:
```

An example of the least optimized save point.
Buffer overflow code (at .L0JMP) is not shown.

```
L0:
        ## iref        0x10fc          # Instruction reference at 0x10fc
        cmpl    12(%ebp),%eax
        ## drrefl      12(%ebp)        # Read longword at 12(%ebp)
        ## iref        0x10ff          # Instruction reference at 0x10ff
        ## savei       0x10fc          # Save basic block in trace buffer
        ## saver       %eax            # Save register %eax in buffer
        ## saver       %ecx            # Save register %ecx in buffer
        movl    %eax,(%ecx,%eax,4)
        ## dwrefl      (%ecx,%eax,4)   # Write longword at (%ecx,%eax,4)
        ## iref        0x1102          # Instruction reference at 0x1102
        movl    %eax,_b(,%eax,4)
        ## dwrefl      _b(,%eax,4)     # Write longword at _b(,%eax,4)
        ## iref        0x1109          # Instruction reference at 0x1109
        jg      L5
        ## condflow    L5              # Condition control flow to L5
L1:
```

An example of the roadmap for the same basic block.
Lines beginning with "##" are roadmap instructions.