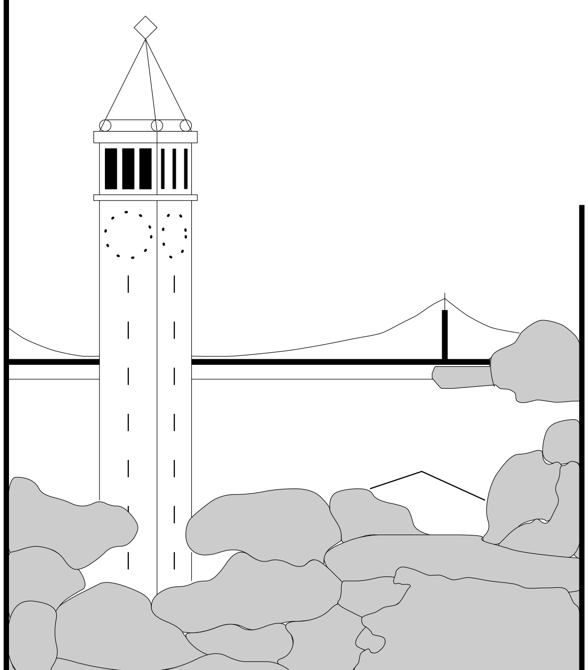
Multiprocessor Memory Reference Generation Using Cerberus

Jeffrey B. Rothman and Alan Jay Smith



Report No. UCB/CSD-99-1054

August 1999

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Multiprocessor Memory Reference Generation Using **Cerberus**†

Jeffrey B. Rothman and Alan Jay Smith
Computer Science Division
University of California
Berkeley, CA 94720

August 6, 1999

Abstract

This paper presents **Cerberus**, an efficient system for simulating the execution of shared-memory multiprocessor programs on a uniprocessor workstation. Using EDS (execution driven simulation), it generates address traces which can be used to drive cache simulations on the fly, eliminating the large disk space requirements needed by trace files. It is fast because it links the program to be traced together with the cache or statistics gathering tool into a single executable, which eliminates the context-switching needed by communicating processes. It is flexible because it has a simple interface which allows users to easily add any kind of module to use the generated trace information. It compares favorably to other existing tracers and runs on a commonly available workstation. And it is accurate, allowing cycle-by-cycle interactions between the simulated processors. The resulting slowdown from **Cerberus** is approximately 31 in uniprocessor mode and 45–50 in multiprocessor mode relative to the workloads run natively on the same machines.We demonstrate that EDS uses only 5 percent of the total execution cycles when combined with a cache simulator and show that EDS is just as efficient as using trace driven simulation.

The implementation details of **Cerberus** are provided here, along with a performance analysis of multiprocessor simulation in the **Cerberus** environment. Some of the other simulation and trace generation tools are surveyed, with the strengths and weaknesses of those tools discussed.

†The work presented here has been supported in part by the State of California under the MICRO program, Sun Microsystems, Toshiba Corporation, Fujitsu Microelectronics, Cirrus Corporation, Microsoft Corporation, Quantum Corporation, and Sony USA Research bLaboratories. Partial support was also provided by Siemens A.G., which supported Jeff Rothman during some of this work.

1 Introduction

Execution driven simulation (EDS) is a powerful method for evaluating computer architectures. EDS is particularly useful for deriving accurate results for multiprocessor systems, since the precise sequence of instruction executions on the parallel processors may vary as the simulated design varies. We have developed **Cerberus**, an EDS simulation system, to provide accurate cycle-by-cycle instruction simulation with a low degree of slowdown relative to untraced native execution. **Cerberus** is an extremely flexible system for simulating programs targeted for MIMD machines. Many other EDS tools allow a trade-off of accuracy for speed. **Cerberus** provides the highest level of simulation accuracy with speed comparable to other tools using less accurate simulation modes that also run on uniprocessor workstations. Note that this is a two-edged sword. If the behavior of the traced workload is unstable with respect to minor changes in parameters of the platform on which it is running (e.g., memory timing) or minor changes in the code (e.g., data layout), then EDS will reflect this instability. Such effects may make it hard to do studies in which other parameters are varied, since the effects of those changes may be swamped by the inherant instability of the workload.

There are a number of methods for studying multiprocessor machine behavior. Hardware can be electronically monitored and workloads run directly [TS90]. Synthetic models can be created to generate artificial reference streams [THW90]. Trace driven simulation (TDS) has been widely used, with a wide variety of methods for collecting traces [Smi94, UM97]. EDS is gaining wide usage for accurate simulation of new architectural models. EDS varies from the other methods of modeling by linking the architectural simulator directly to the trace instruction generator, without the intermediate step of storing a set of (invariant) traces. This allows interaction between the hardware model and the software. A trace generation system using EDS has the advantage of simplicity of use, low slowdown, small disk space requirements, and accuracy with easy system parameter reconfiguration. EDS does have the disadvantage that the trace consumer (e.g., a cache simulator) would normally be run on the same system as the trace generator; running the trace consumer on a separate machine would require separate processes on different machines and a pipe between them, which would be very inefficient. Thus, if the tracer runs on a slow machine, the trace consumer must do so as well.

The execution time overhead of a trace generation system is also an important factor to consider. For a system which studies multiprocessor designs by running the simulator on a uniprocessor, the most important speed optimization is incorporating the parallel trace generator, processor thread scheduler and cache simulator into a single process. By utilizating these optimizations with careful design and refinement through profiling, we have created in **Cerberus** a very efficient system for investigating the properties of multiprocessor hardware and software which does not trade accuracy for speed. It is easy to create programs for, using simple SMP primitives to specify parallelism. It uses low-level cycle-by-cycle simulation to very accurately model the interactions between simulated processors. It uses assembly language routines for commonly called simulation functions, which reduces simulation slowdown. And it has a very simple trace generation interface, allowing users to easily add cache simulators, code profilers or other statistics gathering tools.

The remainder of this paper is structured as follows: Section 2 provides background on multiprocessor tracing methodology and an overview of some of the most recent tools developed for tracing memory accesses. Sections 3 and 4 discuss the design and implementation of **Cerberus** and the method used for annotating object code to provide memory reference traces, as well as examples of the code modification process. The performance evaluation of **Cerberus** is found in Section 5. In Section 6 we present our conclusions. The Appendices provide a detailed survey of other multiprocessor tracing tools, discuss difficulties encountered while developing **Cerberus**, details about using the **Cerberus** system, and some of the low-level specifications about our tool.

2 Summary of Trace Generation Methods

A variety of methods exist for deriving trace information from workloads. We provide a brief summary of the methods here; see Appendix A and [UM97] for a more detailed description of the different approaches. Table 1 provides some details of each of the various methods, specifying whether the method could only collect traces for the target architecture (trace driven simulation or TDS) or if the trace information could be piped or used directly with a cache simulator (execution driven simulation or EDS, also referred to as program driven simulation). In addition, we specify the hardware on which it can be run and the approximate slowdown compared to running the workloads under test in native and/or uniprocessor mode.

Hardware approaches to collecting traces for multiprocessors have included adding I/O cards to capture transactions on the memory bus [Wil90, Vas93] and modifying the microcode to capture memory references [SA88]. OS system calls have also been used to capture traces, such as the UNIX debugging command *Ptrace*, to cause an interrupt to a reference collection routine for each instruction of a user-level program [Lac88]. Modification of error correcting code (ECC) bits was used to allow interrupts on just the shared memory references in [RHL$^+$93], which then invoked a cache simulator subsystem.

Software methods have been the most widespread means for collecting and using traces. These approaches can be generally broken down into interpreters (native code running on top of a simulator) and code modifiers (augmented code which allows traces to be collected). The interpretive methods have been used to simulate 68000 execution [Dah91] and the MIPS R3000 instruction sets [VF94].

Augmenting source or object code has been used to trace code running on parallel machines or to simulate parallel execution on uniprocessor workstations. The former approach has been used to add minimally intrusive instructions to the low-level code, writing records to memory buffers [EKKL90, AE90, SJF92]. Modifying code and adding routines to simulate the execution of a parallel machine on a uniprocessor machine has been the most popular method [CMJS88, DG90, BDCW91, Del91, Boo94]. Our trace simulator falls into this last category.

Name	Reference	Method	Memory Model	System	Processor	Slowdown
ATUM2	[SA88]	TDS	Bus	VAX 8350	VAX	20
Wilson	[Wil90]	TDS	Bus	1 Proc. Multimax	NS32032	1
Vashaw	[Vas93]	TDS	Bus	Multimax	MS32032	6
Lacy	[Lac88]	TDS	Bus	Sequent Balance	NS32032	100,000
Tracer	[AE90]	TDS	Any	Sun 4/260	Sparc	10
MPTrace	[EKKL90]	TDS	Bus	Sequent Symmetry	i386	3–8
Trapeds	[SJF92]	TDS	Bus	Multimax	NS32532	45–163
		EDS	Hcube	iPSC/2	i386	13–22
RPPT	[CMJS88]	EDS	Any	Any	Any	1.3–35
Proteus	[BDCW91, Del91]	EDS	Any	MIPS Workstation	MIPS R3000	35–100
Tango	[DG90]	EDS	Any	MIPS Workstation	MIPS R3000	500-18000
FAST	[Boo94]	EDS	Any	MIPS Workstation	MIPS R3000	10–110
Wisconsin Wind Tunnel	[RHL$^+$93]	EDS	Fat Tree	CM–5	Sparc	52–250 w/ Cache Sim.
DDM	[Dah91]	EDS	Bus Tree	Sun 3/80	MC68000	2000
MINT	[VF94]	EDS	Any	Any	Any	20-70
Cerberus	This Paper	EDS	Any	MIPS Workstation	MIPS R3000	19-52

Table 1: Summary of surveyed tracing tools.

3 The Cerberus Model

At the time we were developing **Cerberus**, the existing tools for generating traces were slow, inexact with respect to individual processor timings, and/or difficult to use or interface with user created cache simulators. We designed our tool to overcome these problems and to simulate multiprocessor program execution as quickly and efficiently as possible. The goals we set out to achieve in creating our tracing tool were: (1) create a system for uni- and multiprocessor simulation and instruction and address trace generation; (2) have an efficient system that would be able to simulate billions of instructions in a reasonable amount of time; (3) support more than one model for expressing shared-memory parallelism; (4) be able to attach modules easily (such as a cache simulator or code profiler) to consume traces on the fly, avoiding the use of massive amounts of disk space to store traces; (5) be able to link a simulator to the trace generator in one UNIX process to minimize operating system context switching; (6) accurately estimate execution time by simulating all user code and system libraries using instruction-level simulation (i.e., processor thread switching after each instruction).

The following subsections describe the multiprocessor shared memory model and design of the **Cerberus** simulation system.

3.1 Expressing Parallelism

Parallel programs begin with a single thread, which then creates the other threads after an initialization phase in the program. **Cerberus** supports two programming models, the Sequent model [Seq87] in which N threads are forked off at once (**m_fork**), and the **s_fork** model, which forks off one thread at a time. Much of the parallelism is created by the use of special functions in the original source code, such as special **fork** functions (**m_fork** and **s_fork**) to create multiple processor threads. These functions, as well as synchronization (locks and barriers) are handled by function calls to the thread library at runtime.

In the **m_fork** (Sequent) model, the most important function is **m_fork**. The arguments to **m_fork** are a pointer to a function (which is to be executed in parallel by all processors) and arguments to that function (same for **s_fork**). Upon a call to **m_fork**, the program switches to multiprocessor execution mode and a previously set number of processors start executing the same function. Each processor runs until it completes execution of the task it was assigned. Once all processors finish executing their functions, the program returns to uniprocessor mode. From uniprocessor mode it is possible to start other parallel executions of functions using **m_fork**. However, **m_fork** is usually called once per program.

The number of processors is set by using the **m_set_procs** command before **m_fork** is invoked. Each processor is assigned a unique identification number (which can be obtained by calls to the function **m_get_myid**) that is used to differentiate between the processors to allow them to perform separate parts of the computation. Special synchronization constructs are also provided. Locks are provided to serialize accesses to shared data structures and to implement critical sections. Barriers are provided to separate program phases and for synchronization. This can be used to make sure that all processors have completed a task before continuing to the next one. The most commonly used barrier is **m_sync**, which waits until the number of processors set by **m_set_procs** have checked-in before resuming execution of the program.

In the **s_fork** model, processor threads are created one at a time by a loop in which processor 0 calls **s_fork** (similar to the **fork** process-creating function in UNIX). Unlike the Sequent model, there is no implicit count of the number of active threads, so barrier operations must explicitly state the number of threads involved.

3.2 Memory Model

Two different memory models are supported by **Cerberus**: the Sequent model, which can called the "heavy threads" model, and the "lightweight threads" model. Most of the description here concerns the Sequent model, with differences between the models described at the end of this section.

During compilation, variables are assigned by the compiler to various locations in the memory data space, depending on the size and type of the variables (Figure 1). Due to the RISC nature of the MIPS processor, 32-bit address pointer creation requires two instructions to load the full data address. Small data (typically less than or equal to eight bytes in size) are assigned to a 64 Kbyte region of memory (the global pointer (**gp**) table) which allows access to data using a one instruction sequence. Larger data must be put into other data sections which requires two instruction sequences to access. Bulk storage space is also available for uninitialized data, which is not stored as part of the executable file but allocated at run-time. The data space above the bulk storage area is used for dynamic memory allocation.

During runtime, when a "fork" routine is called the first time, each new thread is provided with its own copy of the entire data space (except code and read-only data) when it is created. The original data space is shared by all threads, and holds the shared variables, as shown in Figure 1. **Cerberus** ensures that all references to shared variables are directed to this shared address space. In the Sequent model, all variables that are not explicitly designated to be shared with the **shared** type qualifier are private. This includes global variables as well as automatic (stack) variables. Therefore it is necessary to provide each processor with its own (private) copy of the memory space. There also needs to be a shared memory space, for which purpose we use the memory space created by the compiler. In addition, memory space is set aside to allow dynamic memory allocation, keeping the shared heap in proximity to the shared memory space. Likewise, the local heaps and stacks for each processor are contiguous with each processor's local memory space.

The "lightweight threads" model is also supported by **Cerberus**, which is used by the SPLASH-2 applications suite [WOT+95]. In this model, all global variables are implicitly shared; only the stack variables are private. Use of this model is specified by passing a command to the runtime system at start-up, which causes the threads to use the shared memory for all global variables. It is thus not necessary to specify which memory locations are shared as in the heavy threads model.

3.3 Processor Model

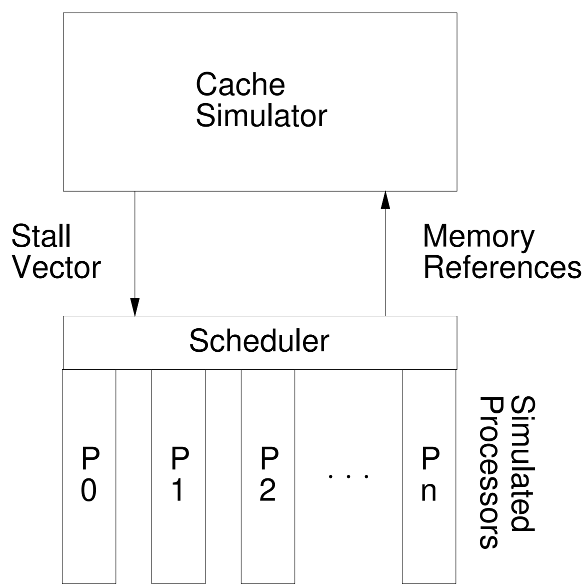


Figure 2: Model of processor-scheduler-cache simulator interaction.

Figure 2 shows a schematic of the **Cerberus** simulation model. At each time step, all available processors are scheduled to run for a single instruction and return the instruction (and data) addresses accessed during that cycle. Some threads may not be available

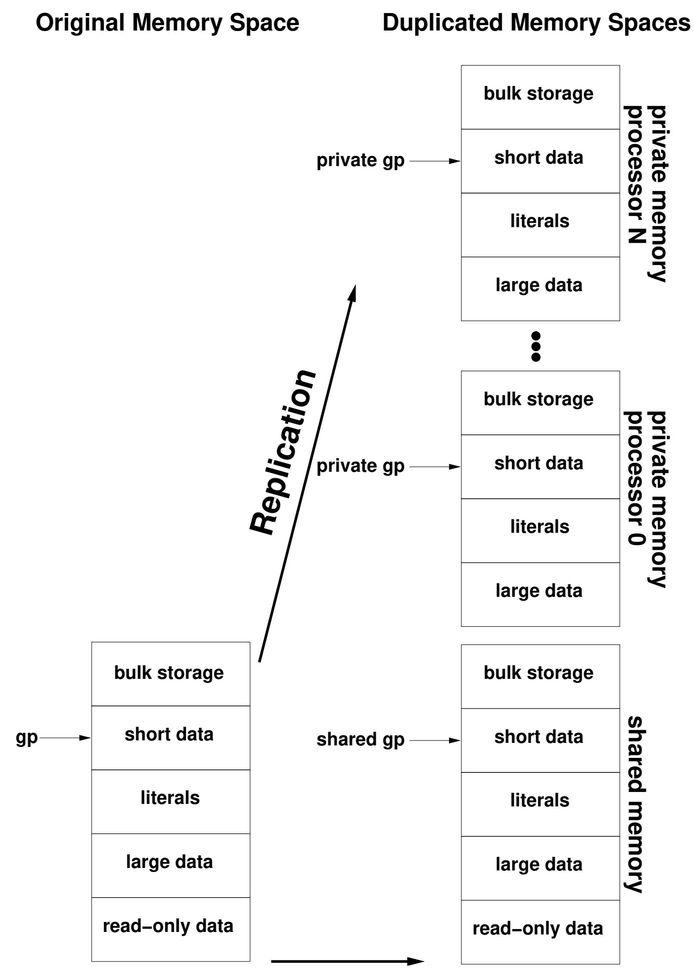


Figure 1: The memory space assigned by the MIPS C compiler is duplicated, with one copy for the shared memory space, and a copy for each simulated processor.

for scheduling, due to the ability to stall individual threads for cache misses or at barriers. Once all the memory references are collated for a time step, the information is passed on to the cache simulator (or other tool). The cache simulator is called even if all threads are stalled, to tell the cache simulator to advance its clock and perform shared bus/memory system operations, which will eventually cause the simulated processors to become available again. When each thread has completed execution of the function it was assigned, it is deactivated. When all threads have been deactivated, the system returns to single processor mode and finishes the program.

To support simulation of parallelism on a uniprocessor workstation, it is necessary to provide the appearance of multiple code streams executing simultaneously and to support the illusion of shared and private data spaces. Supporting both of these requires a 200+ byte context block per processor to keep track of the state of each simulated processor, such as integer and floating-point register values, control register values, special private and shared memory pointers, and simulation state (detailed in Appendix E). The context block is the key item through which modified code and simulated processors interact to provide the semblance of parallelism.

To be able to simulate multiple threads of execution on a uniprocessor workstation, a simulator has to provide some sort of a thread package with a simulated processor scheduler, or rely on the host machine's operating system for scheduling the simulated processors. With a user-level threads package using instruction-level task switching granularity (which **Cerberus** uses), each simulated instruction appropriately loads and stores the registers with which it interacts and returns back to the simulator. All instructions that read a register (or registers) must load that register from the context block (except for register **r0**, which is always 0). For many instructions, particularly ALU instructions, two registers are read, which requires two values to be read from the context block for those simulated instructions. Any register that is modified during the instruction (typically one register) must have its value copied to the context block at the end of that simulated instruction.

Loading and storing state each cycle provides the ability to switch processor threads after execution of a single instruction. Some simulators switch processors on a basic block granularity [CMJS88, Boo94], which involves calling the scheduler at the beginning of each basic block and requires loading and storing all the affected simulated registers at the beginning and end of each basic block.

Synchronization operations provides an opportunity for scheduling optimization. When processors reach a barrier, they are descheduled until all processors reach that barrier. This provides a means of reducing simulation overhead and eliminating redundant spin-wait traces. A further optimization (which we have not investigated) is to also descheduling processors waiting for locks, reactivating them when they acquire the lock.

Not all EDS methods use the low-level scheduling and fine granularity thread context switches that **Cerberus** uses. Some methods use a UNIX process per simulated processor, which avoids user-level scheduling and explicit saving of simulated processor status [AE90, DG90]; UNIX process switching, however, incurs a rather high overhead. Synchronization between processors is performed at varying levels of granularity, which can be controlled by the user by specifying the accuracy/slowdown trade-off they are willing to make. Another method inserts calls to the simulator into the C code before compilation, which is used to switch processor contexts when an interesting event requiring synchronization occurs [Del91]. Since the compiler takes care of saving state between function calls, little explicit effort needs to be made to keep track of each processor's state.

4 Implementation Details

In the process of designing **Cerberus**, we set about with certain goals to make the implementation as clean, simple, accurate, and efficient as possible. To make it clean and simple, it was designed to automatically convert parallel C code to allow it compile on a normal workstation with minimal work by the user. To make it accurate, each simulated processor was designed to execute a single emulated instruction and pass control back to the thread scheduler. In addition, the scheduler was designed to deschedule threads while they were stalled on various conditions, like cache misses. Note, however, that our simulator does not currently account for varying instruction times - we treat an integer add and a floating-point divide as taking the same time. Such timing could be added, if desired, but it would make the EDS system extremely platform specific.

To aid in attaining the efficiency goal, we created our own multi-threaded processor simulator that can be linked up with a cache simulator (or other tools) and run as a single executable using one UNIX process. This eliminates a significant amount of OS overhead caused by context-switching. We also made great efforts to update information in the executable file (like source code line numbers and other symbolic information) in order to be able to easily use a uniprocessor debugger such as **dbx** [Com86] with the system. Our tool has roughly only a 40 to 50 times slowdown running with a stub cache simulator in multiprocessor mode; this makes it roughly 200 times faster than Tango with the same granularity of simulation (simulating each thread in instruction sized steps). We have thus met our goals of an efficient multiprocessor simulator that allows use of execution driven simulation for cache simulators and other tools.

The following sections describe the code modification process from C code to machine language executable. Section 4.1 describes some of the salient characteristics of the MIPS R3000 processor, which is the target architecture of the modification process. Sections 4.2 through 4.8 follow the modification procedure step by step. Section 4.9 and 4.10 discuss how the threads package and scheduler interface with the instrumented code. Section 4.11 provides some of the details of the types of cache simulators used with **Cerberus** and how they interact with the trace generation package.

4.1 R3000 Architecture Characteristics

The original target of **Cerberus** was the DEC 3100/5000 series of machines, which use the MIPS R2000/3000 RISC microprocessors. The R3000 instruction set employs a fixed 32-bit format with 3 main format types. This makes it easy to disassemble and decode for modification purposes. There are some characteristics that cause difficulties (discussed in Section 4.12 and Appendix B). One of the most interesting "features" of the MIPS architecture is the partial exposure of the 5 stage instruction pipeline to the programmer. To allow exploitation of pipeline delays caused by certain operations, delay slots are associated with these instructions. Any branch or jump instruction is followed by a delay slot instruction that is executed during the bubble in the instruction pipeline caused by the change in control flow. The delay slot instruction is executed regardless of whether the branch is taken. As part of the code modification process detailed below, the instruction in the delay slot is moved to the position before the branch or jump with which it is associated, taking care to make sure that the results of the delay slot instruction have no effect on the branch. This instruction movement can cause other difficulties when the delay slot is the target of a branch (which some optimizers can introduce into the code).

Load instructions are also followed by a delay slot instruction, which cannot have a dependency on the register being loaded. This load delay causes no major difficulties, but must be observed in the modification process and in the hand-coded assembly routines.

4.2 Overview of Code Annotation and Modification

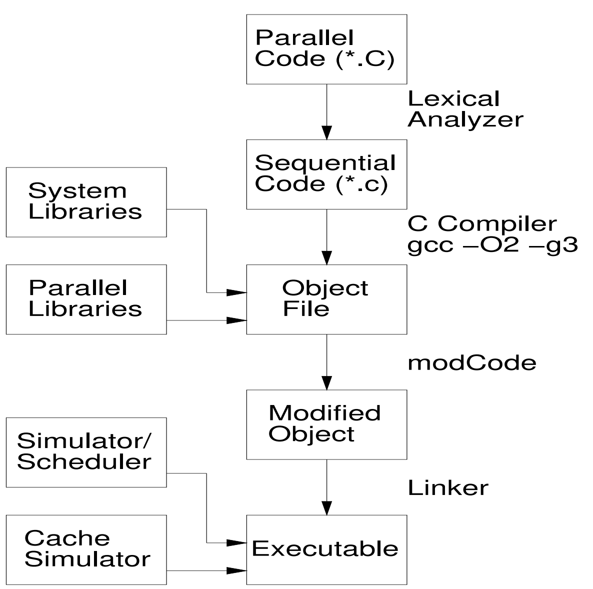


Figure 3: Flowchart of the process by which **Cerberus** takes code targeted for a parallel machine and modifies it to run on a uniprocessor workstation.

One of the chief goals for **Cerberus** was to be able to run a multiprocessor trace generator on a uniprocessor workstation using a standard C compiler. Figure 3 shows the steps necessary to accomplish this task. This process begins by running the parallel source code through a series of macro packages and lexical analyzers, which generate suitable new sequential uniprocessor source code. The new source code renames shared memory variables to allow detection of the intended parallelism in the compiled (object) code. Shared variables are detected because they are prepended with the **shared_** prefix, which can be detected in the object code's symbol table. The following sections detail each stage of the compilation process.

4.3 Parallel C Code Annotation

To generate sequential C code from a Sequent C program, the original source code is run through a lexical analyzer and some shell scripts. The resulting source code retains hints about reconstructing the original parallel shared memory organization of the code, but can be compiled with a standard C compiler. Sequent C's principal difference from normal C is the addition of the **shared** and **private** type qualifiers. The shared and private keywords determine whether a variable is visible to all the processors, or whether each processor has a private copy of the variable. Variables are private by default. These qualifiers are applied to global variables (ones which have a name scope which is visible in all functions).

The lexical analyzer examines all the files included by each .C (parallel code) file with the aid of the shell scripts to identify which variables are shared. Variables are considered private by default, so the **private** keyword is eliminated. Each variable that is shared is annotated by using the C pre-processor's #define statement. For each shared variable, the prefix **shared_** is prepended to the variable name. This prefix can easily be recognized in the object code during the modification process when the variable names are examined. To make sure the variable names exist in the compiler output, the compiler debugging option (-g or -g3) must be turned on.

4.4 Compilation

After the C code is annotated, it is compiled with standard compiler switches (plus -g3 for symbolic information). At this stage standard library functions that are to be traced, along with a pseudo-parallel functions library, are linked together. The parallel library contains additional functionality such as locks and barriers, and **printf** type functions with locks. Special versions of printf and fprintf are necessary to serialize I/O calls to prevent programs from writing their output on top of each other. In addition, a file is linked into the code which contains dummy (unused) calls to the simulator functions. This aids **modCode** (**Cerberus'** code modifier) in inserting calls to the simulator during the code modification process by defining the simulator functions in the symbol table. The file also contains a large array used to keep relocation pointers to data pointers in memory. At run-time, this array (called **SIMrelptr**) is consulted to keep pointers coherent for any new data spaces created (e.g., by **m_fork**).

4.5 Code Modification

The new object file is run through **Cerberus**'s code modifier **modCode**, which adds instructions to the original code in such a manner as to create an output file with exactly the same functionality as the original program, but instrumented so that it generates instruction and data memory addresses as the program executes as well as calls simulation routines for each original instruction.

The modification process turns each original instruction from the unmodified object file into an eight instruction block of code in the output file. Figures 4 and 5 (Table 2 explains the opcodes used) show the before and after samples from the code modification process. The one to eight instruction change causes the final executable to have a factor of eight increase in the code size in addition to the data and code space used by the scheduler and memory subsystem simulator. This expansion is done to permit quick calculations of original instruction addresses, given the modified instruction address. The address calculation consists of a subtraction of a base address, a division by eight (right shift by three), and an addition of a new base address. Expansion by a variable amount would make the calculation more complicated and would likely require table lookups. The choice of eight as the expansion factor was made because it is a power of two (allowing a shift instead of division for address calculations) and because most instructions can be emulated by and make the proper simulator calls with a sequence of eight instructions. Due to the nature of the MIPS instruction set, it is generally necessary to be able to load two values from the simulated register set, directly execute the emulated instruction, and write the result back to the simulated register set. Because of the delay slot associated with every branch on the MIPS architecture, this means the minimum expansion requirement for most instructions is six additional instructions, for a total of seven. This dictates the use of eight as the expansion factor, because 16 would be too much, and would slow the program down even more than it is, as well as multiplying the code size by another factor of two. To speed up simulation, unconditional branches are added into the code to skip over **nop**s when the actions associated with an instruction can be performed in less than 6 instructions (such as **nop**s and **move** instructions in the original code).

4.6 Accessing Data

Since the object code is analyzed instruction by instruction, it is possible to determine statically which variable is being referenced, look up the variable's name, and decide whether or not the variable is shared (by looking for the **shared_** prefix). If the variable is shared, the data space assigned by the compiler is used as the destination, because that region is the designated shared memory area. Otherwise, code is inserted into the eight-instruction block sequence to add a run-time offset (whose value is processor dependent) to the data address, causing the variable accesses to be in one of the private memory spaces associated with the processor, as displayed in Figure 6. In the case of the lightweight threads model, this run-time offset is 0, which causes all global memory to be shared memory.

As mentioned in Section 3.2, forming a 32-bit pointer to access "large" data requires two instructions. In a normal MIPS executable, a special range of memory space, called the global pointer table, is maintained to allow single instruction accesses to memory. Instructions using this table access memory by a 16 bit signed offset from the global pointer (**gp**). Only small data and small block storage are kept in the global pointer region, which can be up to 64 Kbytes in size. When a new data space is created, each processor gets its own copy of the entire data space, with a corresponding global pointer which points to its own copy of the global pointer table. This local (private, unique to each processor) pointer to the copy of the global pointer table is referred to as the **lgp** register. For accesses to the shared part of memory, there is a shared global pointer (**sgp**) which all processors have in common.

Shared variables are accessed using an offset from the **sgp** or by creating a 32-bit pointer to the shared data space. Private variables are accessed using an offset of the **lgp** or by adding a processor dependent offset to a 32-bit pointer, which then points to a private data space. Each processor has a complete duplicate of the original memory space (copied the first time **m_fork** or **s_fork** is called) as its private space. The portions of the shared memory space used by shared variables is duplicated in each processor's memory space, but is never accessed. Likewise, all private global variables have a corresponding region

Opcode	Explanation
addiu r2,r3,100	Add immediate (unsigned) r3 and 100, result goes into r2
addu r2,r3,r4	Add unsigned r3 and r4, results put in register r2
beq r2,r0,0x100001e0	branch to address 0x10001e0 if r2 equals r0 (r0 is hardwired 0)
b 0x10001000	Unconditional branch to the instruction at address 0x10001000
jal *routine*	Jump to *routine* and store return address in register r31
lui r9,0x1008	The upper 16 bits of r9 are set to 0x1008, the lower 16 bits are set to 0
lw r2,-5000(r10)	Load the word at address r10-5000 into register r2
nop	Do nothing for one cycle
slti r2,r3,100	r2 set to 1 if r3 is less than 100, r2 set to 0 otherwise
sw r10,16(r2)	Store the word in r10 into the address r2+16

Table 2: Explanation of opcodes used in Figures 4–7.

```
(mp3d.c:  181) 0x10000184:  0c001211 jal sscanf
(mp3d.c:  181) 0x10000188:  24c60008 addiu r6,r6,8
(mp3d.c:  182) 0x1000018c:  8f828330 lw r2,-31952(gp)
(mp3d.c:  182) 0x10000190:  00000000 nop
(mp3d.c:  182) 0x10000194:  8c420008 lw r2,8(r2)
(mp3d.c:  182) 0x10000198:  00000000 nop
(mp3d.c:  182) 0x1000019c:  28420064 slti r2,r2,100
(mp3d.c:  182) 0x100001a0:  1040000f beq r2,r0,0x100001e0
(mp3d.c:  182) 0x100001a4:  00000000 nop
```

Figure 4: Example of code before expansion, Figure 5 shows code after augmentation.

in the shared memory space which is never accessed. Private memory accesses using 32-bit pointers are initially calculated as pointers to the shared memory space, but are deflected to the private memory space of a processor by adding an offset. When the lightweight threads model is in use, all global memory is shared, so the **lgp** and **sgp** point to the same place in memory.

One complication in copying the memory space is finding pointers and determining how to deal with them: copy them without modification (pointer to a shared data structure) or to change them to point to a local version of the same data structure in a processor's memory space. The heavy threads paradigm employed by **Cerberus** causes the private data space for each processor to be copied from the data space created by the compiler, which can have pointers within it defined at compile-time. If the data space is initialized with pointers to data objects, it is necessary to determine whether the pointers are really to private or global memory when new data spaces are created. If a pointer is to shared memory, the address in the copy of the data space is already correct. If the address points to private data space, care must be taken to make sure the pointer points to the appropriate space in the copied (private) data space, not to the memory location the compiler established (which by default is the shared memory space).

For example, some routines have pointers to buffers that are defined at compile-time, and each processor must be sure to have its own private buffers. Because pointers to statically defined buffers have relocation entries associated with them, it is possible to create a list of relocation entries which automatically point to the variables which point to the buffers (maintained as an array of pointers **SIMrelptr**). During the process of creating the data space for simulated processors at run-time, it is necessary to update these pointers when the data space is copied by **m_fork**; otherwise these pointers would point to a spuriously shared data object (in the default memory space that is used as the shared area), instead of private per-processor copies of the objects. To keep track of these pointers, the SIMrelptr array of pointers is linked in with the code before modification. New relocation entries are added during code modification which point to entries in this array to keep track of all the pointers which need to be fixed; these relocation entries and the entries in SIMrelptr are maintained and updated by the object linker.

Each time a new data space for a processor is created, the original memory space allocated by the compiler is scanned for these pointers by consulting the SIMrelptr array, and the values copied to the new

```
# Delay slot of the call to sscanf
(mp3d.c:  181) 0x400e40:  0c112939 jal pSIMstep
(mp3d.c:  181) 0x400e44:  00000000 nop
(mp3d.c:  181) 0x400e48:  8c480018 lw r8,24(r2)
(mp3d.c:  181) 0x400e4c:  00000000 nop
(mp3d.c:  181) 0x400e50:  25090008 addiu r9,r8,8
(mp3d.c:  181) 0x400e54:  10000002 b 0x400e60
(mp3d.c:  181) 0x400e58:  ac490018 sw r9,24(r2)
(mp3d.c:  181) 0x400e5c:  00000000 nop

# Call to sscanf
(mp3d.c:  181) 0x400e60:  0c112939 jal pSIMstep
(mp3d.c:  181) 0x400e64:  00000000 nop
(mp3d.c:  181) 0x400e68:  00000000 nop
(mp3d.c:  181) 0x400e6c:  0c109110 jal sscanf
(mp3d.c:  181) 0x400e70:  ac5f007c sw r31,124(r2)
(mp3d.c:  181) 0x400e74:  10000000 b 0x400e80
(mp3d.c:  181) 0x400e78:  00000000 nop
(mp3d.c:  181) 0x400e7c:  00000000 nop

# Load a pointer from gp region
(mp3d.c:  182) 0x400e80:  8c480070 lw r8,112(r2)
(mp3d.c:  182) 0x400e84:  0c112948 jal pSIMld
(mp3d.c:  182) 0x400e88:  25058630 addiu r5,r8,-31184
(mp3d.c:  182) 0x400e8c:  8c480070 lw r8,112(r2)
(mp3d.c:  182) 0x400e90:  00000000 nop
(mp3d.c:  182) 0x400e94:  8d098630 lw r9,-31184(r8)
(mp3d.c:  182) 0x400e98:  00000000 nop
(mp3d.c:  182) 0x400e9c:  ac490008 sw r9,8(r2)

# NOP for load delay slot
(mp3d.c:  182) 0x400ea0:  0c112939 jal pSIMstep
(mp3d.c:  182) 0x400ea4:  00000000 nop
(mp3d.c:  182) 0x400ea8:  10000005 b 0x400ec0
(mp3d.c:  182) 0x400eac:  00000000 nop
(mp3d.c:  182) 0x400eb0:  00000000 nop
(mp3d.c:  182) 0x400eb4:  00000000 nop
(mp3d.c:  182) 0x400eb8:  00000000 nop
(mp3d.c:  182) 0x400ebc:  00000000 nop

# Load a value at an offset from the pointer
(mp3d.c:  182) 0x400ec0:  8c480008 lw r8,8(r2)
(mp3d.c:  182) 0x400ec4:  0c112948 jal pSIMld
(mp3d.c:  182) 0x400ec8:  25050008 addiu r5,r8,8
(mp3d.c:  182) 0x400ecc:  8c480008 lw r8,8(r2)
(mp3d.c:  182) 0x400ed0:  00000000 nop
(mp3d.c:  182) 0x400ed4:  8d090008 lw r9,8(r8)
(mp3d.c:  182) 0x400ed8:  00000000 nop
(mp3d.c:  182) 0x400edc:  ac490008 sw r9,8(r2)

# NOP for load delay slot
(mp3d.c:  182) 0x400ee0:  0c112939 jal pSIMstep
(mp3d.c:  182) 0x400ee4:  00000000 nop
(mp3d.c:  182) 0x400ee8:  10000005 b 0x400f00
(mp3d.c:  182) 0x400eec:  00000000 nop
(mp3d.c:  182) 0x400ef0:  00000000 nop
(mp3d.c:  182) 0x400ef4:  00000000 nop
(mp3d.c:  182) 0x400ef8:  00000000 nop
(mp3d.c:  182) 0x400efc:  00000000 nop

# Evaluate loaded value
(mp3d.c:  182) 0x400f00:  0c112939 jal pSIMstep
(mp3d.c:  182) 0x400f04:  00000000 nop
(mp3d.c:  182) 0x400f08:  8c480008 lw r8,8(r2)
(mp3d.c:  182) 0x400f0c:  00000000 nop
(mp3d.c:  182) 0x400f10:  29090064 slti r9,r8,100
(mp3d.c:  182) 0x400f14:  10000002 b 0x400f20
(mp3d.c:  182) 0x400f18:  ac490008 sw r9,8(r2)
(mp3d.c:  182) 0x400f1c:  00000000 nop

# NOP from delay slot of branch
(mp3d.c:  182) 0x400f20:  0c112939 jal pSIMstep
(mp3d.c:  182) 0x400f24:  00000000 nop
(mp3d.c:  182) 0x400f28:  10000005 b 0x400f40
(mp3d.c:  182) 0x400f2c:  00000000 nop
(mp3d.c:  182) 0x400f30:  00000000 nop
(mp3d.c:  182) 0x400f34:  00000000 nop
(mp3d.c:  182) 0x400f38:  00000000 nop
(mp3d.c:  182) 0x400f3c:  00000000 nop

# Branch instruction
(mp3d.c:  182) 0x400f40:  0c112939 jal pSIMstep
(mp3d.c:  182) 0x400f44:  00000000 nop
(mp3d.c:  182) 0x400f48:  8c480008 lw r8,8(r2)
(mp3d.c:  182) 0x400f4c:  00000000 nop
(mp3d.c:  182) 0x400f50:  11000073 beq r8,r0,0x401120
(mp3d.c:  182) 0x400f54:  00000000 nop
(mp3d.c:  182) 0x400f58:  00000000 nop
(mp3d.c:  182) 0x400f5c:  00000000 nop
```

Figure 5: Example of code after expansion.

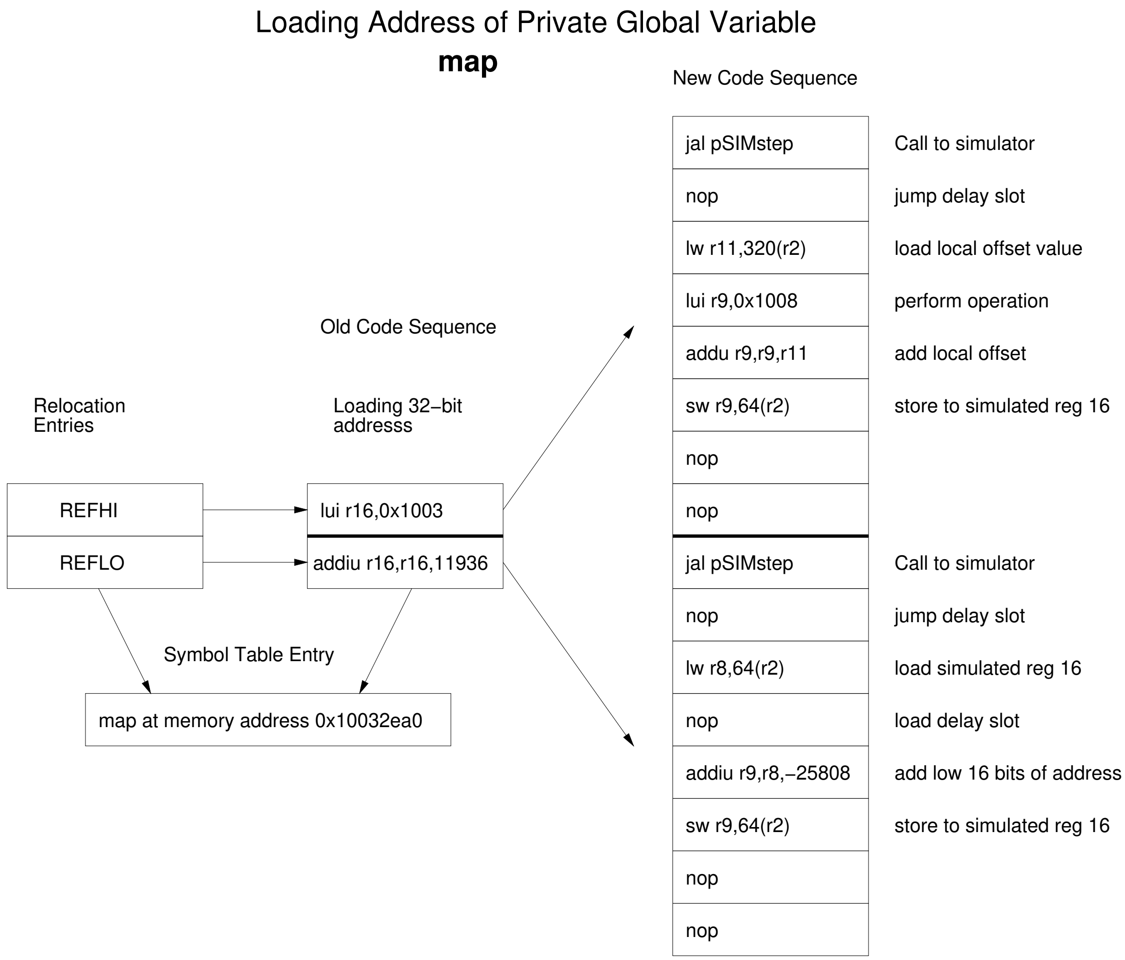


Figure 6: Expansion of a two instruction sequence to load the 32-bit address of a global variable which is private to each processor. Without intervening in the address calculating process, the address would point to the shared region of memory. For private global variables a special offset value is added to the upper 16 bits of the address, to redirect the memory reference (to variable **map**) into a processor's private memory space. The redirection register value is kept in simulated register 80 (a 320 byte offset from the simulated processor's context block pointer (register **r2**)).

data space are updated to be appropriate for the new private memory space (Figure 1).

4.7 Interfacing with the Simulator-Scheduler Package

Each instruction in the original unmodified object code is turned into a block of eight instructions in the modified object code. Not all the instructions in the block are used in all cases, but all use the first few instructions of the block of eight to call the address collator (indirectly through an assembly language "staging" function). If there are still active threads (simulated processors) to process during the current cycle, the thread scheduler is called. When all the addresses have been collected for the current time step, the address collator calls the cache simulator (or other tool module). The scheduler returns control to a particular thread only when this simulated processor is ready to proceed, after any (simulated) memory or other delay. Figure 7 shows an example of augmentation, demonstrating how an instruction from the unmodified object code gets augmented to call the simulator, load the necessary simulated register information, perform the operation, and save the resulting state. The call to **pSIMstep** is a call to an assembly language "staging" function which in turn calls the C language portions of the simulator. This staging function sets up for a call to the C-based collator/simulator routines, saving sim-

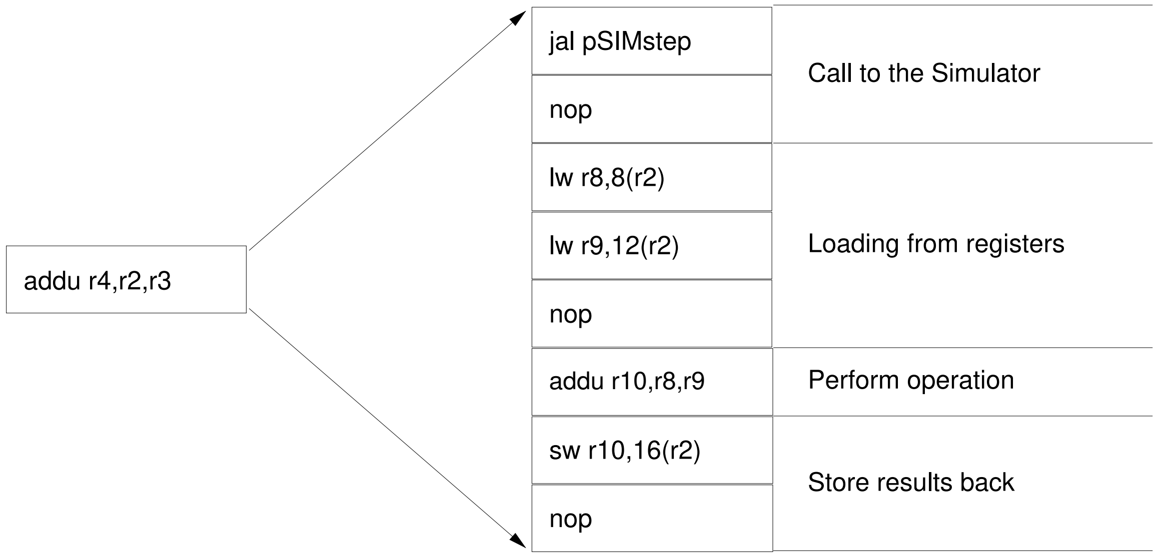


Figure 7: Expansion of one instruction in the original object code to eight instructions in the modified object file. The first **nop** is necessary due to the jump delay slots used on the MIPS architecture. The second **nop** is due to a load delay slot. Most instructions can be emulated with an eight instruction block; those requiring more instructions call out to additional assembly language routines.

ulated processor state, preparing the instruction and data (if appropriate) addresses for the simulator to process using the MIPS standard function calling conventions [KH92]. Not all augmented instructions call **pSIMstep**, but may call one of a number of similar hand-coded assembly language routines to handle special cases, such as load-store operations, locking, floating-point instructions and other more complicated operations. The assembly language routines call one of three C language routines, which handle instruction addresses (**SIMstep**), instruction and data addresses (**SIMmem**), or locking operations (**SIMlock**). These three routines collate the memory addresses, call the cache simulator or other tool once each time step, then call the scheduler to return control to one of the active threads. At some point, control is returned to each of the threads that are active, on a round-robin basis.

Once the scheduler has returned control to the thread, the rest of the eight instruction block is run, including the direct execution of the actions performed by the simulated instruction. The value returned by the scheduler to the thread in register **r2** is the base address of the context block for that particular thread. All the saved context for a thread is accessed as an offset from **r2**. For example, all simulated integer registers **rx** can be found at memory address $r2 + 4 * x$. Simulated floating-point registers **fy** are found at memory address $r2+4*y+128$. Other information, such as the program counter and status control registers are found as higher offsets from **r2**.

Register **r2** was chosen for this purpose because the MIPS calling conventions specify that **r2** contains the return value from function calls. For uniprocessor mode operation, this allows the C simulator functions to return control to a thread using a simple C **return** instruction. This has the effect of making it appear as though the augmented program makes simple calls to the simulator, which then returns as a function call should. In the case of a multiprocessor simulation, the paradigm is somewhat different than standard function calls, in that control jumps around between different parts of the augmented code and the scheduler, and some scheduler related function calls never return, or at least not as expected in a normal program. The changes of control act more as **gotos** with passed parameters than proper function calls. An assembly language routine **psched** is called by the scheduler to change the function call paradigm into the appropriate changes of control for the whole simulation.

4.8 Final Linking

The final stage in creating the simulation executable is linking the modified code to the analysis tools. Typically a cache simulator with a memory subsystem is linked in with the trace generator, but any kind of user defined tracing module can be interfaced. A simple stub that dumps the traces to disk or performs simple analysis of the generated traces can be used in place of a complex cache simulator.

Standard library files for the I/O and math functions that the analysis tools use are linked in as well. There may be two copies of some standard library functions in the executable: modified and normal. For example, the **printf** function used by the parallel code needs to be distinct from that used by the cache simulator, otherwise all printf's would be traced, not just those used by the parallel code under study. The modification process changes the names of the instrumented versions of the standard library functions to prevent the analysis tools from using the wrong libraries. Once the modified code, the cache simulator and the standard libraries at linked together, a single executable which generates and consumes traces from the code is ready to run.

4.9 Interface to the Cache Simulator

The C simulation routines collate the addresses generated by the instrumented code in a special data structure in the preparation of a call to the cache simulator (detailed in Appendix E.1).

In uniprocessor mode, the cache simulator is called each time the scheduler is called. In multiprocessor mode, addresses are accumulated for each active processor during a global cycle time (or time step). Once all the addresses have been accumulated for all active processors during a time step, the cache simulator is called. The return value from the cache simulator is a bit vector which tells the scheduler which processors are stalled due to cache misses, and allows the processors with cache hits to proceed to the next instruction. The scheduler uses the bit vector to determine which processors are capable of being scheduled during the next time step. A bit value of 1 means the processor is stalled; 0 indicates that the processor can be scheduled. Each non-blocked processor is scheduled and capable of generating an instruction address and possibly a data address for the next cycle.

4.10 Scheduler

There are two types of schedulers used in **Cerberus**. For sequential mode operation, a very simple scheduler runs the single thread if it is ready, and waits while it is stalled on a cache miss. While the processor is stalled, the simple scheduler waits in a loop, repeatedly calling the cache simulator (which runs a bus simulator) until the cache miss has been serviced. A simple C **return** instruction suffices to return control to the single thread. A more complex scheduler is used for multiprocessor operations and is described below.

The parallel scheduler consists of two loops in series. The first loop searches for a processor that is ready to run during the current time step. This loop schedules any processor that is ready and calls the low-level assembly language scheduler **psched** (which never returns). If all non-stalled processors have been processed during the current time step, the loop finishes and the second loop is run.

The second loop calls the cache simulator with all the addresses that have been accumulated during the current time step. The cache simulator generates a bit vector of active and stalled processors for the scheduler to use, based on cache misses occurring and resolved during the current time step. Internal to the second scheduling loop is a loop which searches for any processor ready to run. If successful, the simulated processor is scheduled and the low-level scheduler **psched** is called. If all processors are stalled (either due to cache misses, waiting at a barrier, or deactivated), the second loop calls the cache simulator and advances the global time by one. In this case the cache simulator is not provided with any new memory addresses, but is called to advance the bus simulation, which will eventually un-stall the caches. This process continues until either a processor is ready (cache operation finishes) or until the parallel part of the program has finished. At some point a thread will be ready to execute and will be scheduled. Eventually, all threads will die at the end of parallelism in the program and the system will resume single thread execution using the **pjoin** routine. The only time the second major loop really exits is when parallelism finishes, otherwise a non-returning call is made to a thread which is ready to be scheduled.

4.11 Cache Simulators

The most typical modules used with **Cerberus** are cache simulators, code profilers, or other statistic gathering tools. All use the same interface and are easily linked into the trace generator in the last stage of executable file creation. Most of the cache simulators we have created to use with **Cerberus** model fully-associative caches, designed to work with a fixed cache size for each simulation. To make the caches as efficient as possible, cache block (line, sector) lookups use hash tables which have the same number of entries as the number of blocks in the cache (like a direct-mapped hash table), leading to a quick determination of whether a hit or miss has occurred. The blocks are also maintained in a single large linked-list in LRU order for replacement, with a pointer to the last block for quick LRU block displacement on a cache miss when the cache is full. The LRU list and hash lists are doubly-linked for ease in updating the list when removing blocks from any location within the lists (which can occur on cache hits). No attempt is made to maintain stack distances, as was done in [TS89].

To aid in shared memory operations, a special cache-like structure with unlimited capacity is used which has an entry for every distinct data block referenced by all processors. This "supercache" is a feature in many of our cache simulators. It has no effect on the consistency protocols under test, but it keeps track of which caches contain which blocks. Without the supercache, all caches must be searched when certain operations (such as invalidations, updates, cache misses, etc.) occur to shared blocks. It is generally the case that contention for shared blocks is low [GW92], so the supercache is useful in cutting down on searches.

To reduce bus traffic caused by locks, our cache simulator uses a slightly different model of locks than do Sequent machines. This has no effect on the lock paradigm assumed by the code, but is made for simulation efficiency purposes. If a cache read miss occurs to the memory location used for a lock operation, the Sequent treats the resulting bus access as a write miss [TS90], which can cause a tremendous amount of bus traffic for high contention locks. Our cache model treats a read miss during a locking operation with a special read-invalidate operation, which reads the cache block and detects whether the word-size lock is locked or not. If not locked, a block invalidation operation is broadcast if the block is in the shared state. If the lock is locked, the bus operation acts like a block read. This allows processors to spin on locks in the cache without causing extra bus traffic. To implement this feature, it is necessary to use the lock's actual data address, which is passed by the trace generator. The cache simulator modifies (locks) the lock's memory location (using the **SIMgotlock** call), modifying the lock's value if it is unlocked and takes the appropriate coherence action.

Part of the **Cerberus** package contains locking routines to use with the cache simulator. The cache simulators (even simple stubs) are required to call the locking routines when appropriate. The locking was originally handled by special assembly language locking routines called directly from the modified code, but this was found to be incompatible with using the simulators interchangeably with TDS mode. By handling locking in the cache simulator, it is possible to make the coherence operations take place properly at the time the lock is modified, instead of trying to anticipate the lock being acquired by the modified code based on the cache block state.

4.12 Summary of Implementation Difficulties

This is a short summary of the implementation difficulties we encountered while creating our tool. A more detailed version with our solutions to the problems can be found in Appendix B. Many of the difficulties have little to do with multiprocessor simulation, but were encountered while trying to simulate the uniprocessor workloads, such as SPEC92 or the usual systems libraries.

When trying to modify FORTRAN code, it was found that the FORTRAN compiler puts read-only data into the text section, which has to be detected in order not to modify the data, assuming it was code.

A pair of functions implementing non-local gotos allowing jumps out of and into the middle of functions (**setjmp** and **longjmp**) requires that special state be saved in order to save and restore state the way those functions expect.

Low-level memory allocation functions **sbrk** and **brk** need to be intercepted, because both the modified code and the simulation package have their own versions of these functions. This can lead to inconsistent pointers to the top of memory, which can cause segmentation faults.

Branches must reach eight times as far in the modified code, since the code is expanded by a factor of eight. On rare occasions the branches cannot reach far enough, so measures had to be taken to substitute jumps for branches.

Branch delay slots in the MIPS machine language are very difficult to deal with in certain cases. Some optimized code makes the delay slot instruction the target of other branches. This situation effectively makes the delay slot instruction part of the overlap between two basic blocks. It is necessary to be able to detect during run-time whether the delay slot instruction is being executed at the end of a basic block or at the beginning of the next basic block.

It was found when porting the simulator between different machines using the same CPU that the C compilers defined program symbols in inconsistent ways, and typically at variance with the official MIPS specifications [MIP89]. C macros were analyzed to detect which compiler was being used.

Running the simulator on different machines (with identical operating systems) or under the standard debugger would often give slightly different results in terms of miss ratios. This was due to slightly different values that the stack pointer was assigned by the different machines at run-time.

Some of the C compilers ignore the **volatile** type qualifier, which is used to force the code to read values from (shared) memory instead of caching those values in registers. This requires using more recent compilers which implement **volatile**. Sometimes it is necessary to insert extra **volatile** qualifiers in front of variables which are used for spin-locks.

5 Performance

To determine the overhead of simulation, we have compared the user's CPU time, as computed by **/bin/time** of the uniprocessor version of the programs, with the time from the stub cache simulator in uniprocessor mode (Table 3) and for multiprocessor mode with a stub simulator and a full cache coherent cache simulator (Table 5). As will be shown, an average slowdown of 31 is observed for the workloads tested by the simulator in uniprocessor mode and a 40 to 50 times slowdown for multiprocessor operation, simulating just the stub (no cache simulator), but with synchronization operations supported.

5.1 Workloads

The four programs used for performance measurements come from the first SPLASH suite [SWG92], which are regularly used in our research. These workloads consist of: **MP3D**: a hypersonic rarefied fluid flow simulation, using Monte Carlo methods; **LOCUS**: a commercial quality VLSI standard cell router; **OCEAN**: a simulation of large-scale ocean movements based on eddy and boundary currents; and **WATER**: a measurement of the forces and potentials involved over time among water molecules in motion. The problem size and characteristics of the workloads we used can be found in Tables 3–5.

5.2 Measurement of Overhead

To measure the overhead due to the trace generation process, we compared the user times to run the unmodified program (with a single processor) to the modified code with a stub memory interface. The first set of measurements (Table 3) uses the **m4** NULL (uniprocessor) ANL macros [LO87] supplied by Stanford University, which eliminate all parallel constructs from programs. This removes locks, barriers, and the parallel fork mechanism. Without using parallel scheduling, **Cerberus**'s scheduler is a much simpler routine which has low overhead.

The difference in slowdowns among the various workloads (seen in Table 3 and further down in Table 5) is due to the instruction mix of each program. As will be explained in Section 5.4, floating-point instructions require about 40 instructions to emulate in the worst case, yet because many of them require multiple cycles to execute, the slowdown for a floating-point instruction can be relatively small. The LOCUS workload has no floating-point instructions, whereas MP3D, OCEAN, and WATER have 9.7, 25.6 and 17.2 percent floating-point operations of instructions executed, respectively. In addition, these programs have varying mixes of fixed point arithmetic and memory operations, which can cause their overheads to vary.

For the multiprocessor simulation evaluation, the ANL **m4** macros were modified to use the Sequent's **m_fork** command to create new processor threads and the **m_sync** command for barriers. The code in the workloads was slightly changed to work with the new Sequent primitives. The major modification was to change the loop that created the threads into an **m_fork** call to a function that differentiated between the master thread (processor 0) and the slave threads. The declarations for variables were changed to include the **shared** type qualifier when appropriate. More recently we implemented the **s_fork** single thread creation routine, allowing the workloads to be used without modifying the original source code.

5.3 Simulation Slowdown

Using a single workstation and the user CPU time from the UNIX **/bin/time** command, we measured the execution time for the sequential version of our workloads run natively on a workstation and for the simulated parallel versions. These measurements were all taken on a DEC 5000/125. Table 5 shows the ratios of the runtimes for all the simulations in comparison to the native sequential version of the workloads. The average slowdown with the stub simulator attached for small amounts of parallelism is around 45; with a complex cache simulator attached simulating a cache coherency protocol [RS99] (similar to the Illinois protocol [PP84]) with 16 byte blocks and 16 Kbyte split instruction and data caches per processor, the average slowdown ranges between 920 and 1030. Note that we are not claiming that our cache simulator is as efficent as some more tightly integrated EDS-cache simulator tools (like CacheMire [BDNS93]; rather our point is that the EDS portion of the simulation only requires about 5 percent of the execution cycles.

One reason the simulations slow down with more

Simulation Characteristics without Parallelism					
Measure	Programs				
	LOCUS	MP3D	OCEAN	WATER	Average
Simulated References (Millions)	100.9	236.7	54.5	65.2	
Instruction References (Millions)	79.8	175.6	38.5	48.4	
Data References (Millions)	21.1	61.1	16.0	16.8	
Slowdown (Simulated/Native Times)	41.2	19.0	29.1	36.4	31.4

Table 3: Comparisons of the overhead in time for several programs compiled with parallel fork and synchronization constructs removed.

Memory Refs. with Parallelism ($\times 10^6$)					
Workload	Number of Processors				
	1	2	4	8	16
LOCUS	97.8	97.8	99.4	101.0	107.8
MP3D	310.8	310.9	311.0	311.2	311.7
OCEAN	52.8	55.8	62.7	78.6	106.5
WATER	66.4	66.4	66.4	66.5	66.6

Table 4: Number of total memory references for the workloads increases as the number of processors increase.

processors is the increase in the number of references (Table 4). OCEAN in particular shows a large increase in the number of addresses generated, which naturally causes the simulation to slow down, particularly with a cache simulator in use. In addition, increasing the number of processors (threads) increases the size of the working set perceived by the host workstation, causing the system to slow down due to cache misses and page faults.

5.4 Instruction Overhead

The modification process is designed to minimize the amount of overhead for the most common instructions while fitting all the necessary state preserving operations into the 8 instruction block. However, a number of instructions cannot be handled under those restrictions and require additional routines to aid in saving and restoring simulated processor state. For example, floating-point instructions have approximately twice the overhead of integer instructions, due to the large number of instructions it takes to load all of the registers involved. In the worst case it must load four floating-point registers (two double precision registers) and the floating-point control register. To make the call to the FP loading routine fit into the eight instruction limit, a fair amount of decoding must be done to the argument passed to the special assembly language loading routine. The routine determines what precision must be handled, and which registers must be loaded. For example, an integer instruction with no special cases to handle (such as the **add** instruction in Figure 7) requires the eight instruction block in the code and 12 instructions in assembly language to handle the interfacing with the C code routines. Floating-point operations require the eight instruction block plus 32 instructions to load two floating-point registers (27 for one register). However, many floating-point instructions require multiple cycles to execute. For example, a double precision multiply takes 5 cycles to execute; a double precision divide takes 19 cycles†. Code with a high density of floating-point instructions will generally show less slowdown than pure integer code, because as noted, all instructions are treated as having the same execution time.

The simulation overhead for each workload depends upon the mix of instructions to be simulated. Some instructions have less overhead than normal integer instructions. Load-store instructions have lower overhead per memory reference because the additional overhead for capturing the data address is small. Branch and jump instructions also have slightly lower overhead than normal instructions, because the branch often takes place in the middle of the block of eight instructions, skipping several of the **nop**s used to pad-out the block to the proper length. In addition, some instructions use register **r0** (which always has value 0) or have only 0 or 1 operands. It is then possible to complete those operations in less

†The latency of these operations on the R3000 can be partially hidden until the results are required. Our simulations, however, assume that instructions are executed serially.

Program	Problem Size	Ratio of Simulation Time to Native Runtime									
		Stub Simulator (Procs)					Cache Simulator (Procs)				
		1	2	4	8	16	1	2	4	8	16
LOCUS	Primary1	44.6	44.3	44.4	45.0	52.4	842.6	792.3	797.3	881.7	1164.0
MP3D	50 steps, 20000 particles	39.5	38.3	37.8	37.5	40.3	1075.8	968.2	926.6	935.1	1030.9
OCEAN	34 x 34 grid, 1000 meter res.	43.8	44.8	50.4	64.7	86.2	1003.0	957.0	1190.3	1357.3	2389.8
WATER	4 steps, 64 molecules	52.0	49.5	48.7	49.2	50.6	1113.1	975.6	914.8	937.8	1035.5
Average		45.0	44.2	45.3	49.1	57.4	1008.6	923.3	957.2	1028.0	1404.8

Table 5: Slowdown ratios of simulated vs. native execution of workloads with parallelism support added.

than 8 instructions (such as **nop**s from the original code, which require only 4). In cases where the operation can be performed with few instructions, an optimization is made whereby a branch is inserted to skip over the **nop**s which are used to pad-out the block of 8 instructions.

5.5 Simulation Overhead

To determine and measure where the simulator spends its time, Pixie [Smi91] was used to profile the simulation executable. Among other statistics, Pixie is able to determine the number of instruction and data references for the simulation and the cycles spent in each function. Table 6 was derived by grouping related functions together and adding up the instructions executed in the functions. The largest single source of overhead (data collation is spread over several functions and also includes the single thread scheduler) is the multi-thread scheduler. During execution, it cycles through the processors, scheduling each one that is not stalled. When all the available processors have been run, the cache simulator is called with the address information generated by the active processors. The return value is a bit vector of the processors which are active and can be scheduled the next step. The process of determining which processors (threads) to schedule, calling the cache simulator and advancing the time step requires approximately 20–40 instructions per call to the scheduler. The number of instructions it takes to find the next processor to schedule falls with the number of processors simulated, so that the simulator actually executes fewer instructions to run the whole simulation with more processors (for well behaved workloads). As the number of processors increases further, the working-set size of the simulation increases sufficiently to bog down the host system due to cache misses and page faults.

Figure 8 shows the average user time for each workload spent by the simulator for each memory reference generated, with a stub simulator attached, and

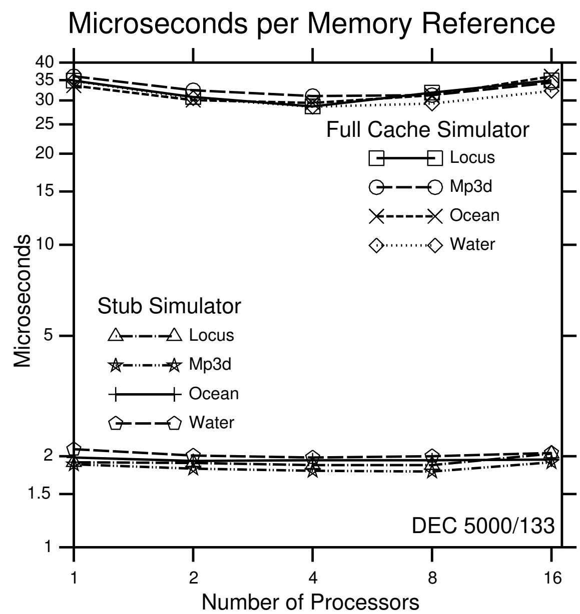


Figure 8: Microseconds per memory reference with a stub simulator and a full cache simulator.

with a shared bus coherent cache simulator. These times per reference are quite low: corresponding numbers for a uniprocessor simulator were reported in the 200–400 microsecond per second range for a DEC 5000/240 in [GHPS93]. We note that the trace generation time is insignificant compared to the trace consumption (cache simulation) time (around 5 percent).

As the number of processors increases, the execution time slightly decreases up to 4 processors and then increases with additional processors. This is particularly noticeable for the simulations with a cache attached. The decrease is due to factors such as more efficient scheduling of processors and the increase in total simulated cache space causing a reduction of the miss ratios (misses are costlier than hits to simulate). The increasing execution time with more processors (beyond 4) is due to the increasing working-

Percentage of Cycles in Simulation					
Source	MP3D	LOCUS	OCEAN	WATER	Avg.
Stub Tool	19.9	22.9	18.8	19.0	20.1
Multithread Scheduler	28.9	24.5	29.8	29.9	28.3
Data Collation (C)	31.2	33.7	29.6	29.9	31.1
Prep. Routines (Asm.)	13.1	11.5	14.6	14.2	13.4
Instruction Simulation	6.9	7.4	7.2	7.0	7.1

Table 6: Measurement of overhead in the simulator with a minimal tool stub, 4 processors.

set/memory space requirements on the host system and increasing communications and synchronization needs for the coherent cache simulation. In addition, the system time for larger simulations increases as the demand on the virtual memory system to run the simulation increases with more processors.

Table 7 shows a comparison of the processing time per reference for a few cache simulators. All of the simulations were run on a DEC 5000/125. The timings were computed using the user time from the ULTRIX utility **/bin/time**. The input to each TDS simulator is a trace file generated from MP3D using 10 steps of 1000 particles in the test geometry. Our cache simulator computes timing information and simulates cycle-by-cycle for bus operations, maintaining a write buffer and performing complex coherency operations, including caching of locks. It also interacts with the memory reference generator to stop the flow of addresses during cache stalls. Each processor maintains fully-associative data and instruction caches. The MPSIM simulator is based on the work in [Tho87], which simulates a range of cache sizes by using stacks. It does not simulate bus timing. The other cache simulator is Dinero, a public domain uniprocessor cache simulator by Mark Hill. Dinero was evaluated using both direct-mapped and fully-associative configurations.

Also included in Table 7 is the time it took to just generate the traces, as well as the trace file sizes. The trace format uses 6 bytes per entry, with 1 byte for the processor number, 1 byte for the reference type, and 4 bytes for the memory address. The traces were generated using **Cerberus** with a utility to dump the traces to disk. An interesting point to note is that it takes **Cerberus** approximately 2 microseconds for each address reference generated (Figure 8), but 13 to 15 additional microseconds to write the information to disk (Table 7), which shows that disk operations heavily dominate trace generation time. A filter program was used to turn the trace files into a format suitable for Dinero, which takes ASCII input of the form "*type address*" (processor number is not necessary). All of the simulations used similar cache configurations (as much as possible) with 16 byte blocks, with 16K byte instruction and data caches when it was possible to specify. The parallel cache simulators used the Illinois coherency protocol for MPSIM and an adaptive invalidation protocol similar to Illinois [RS99] for our cache simulator which used the **Cerberus** system. The results show that a cache simulator using **Cerberus** has speed comparable to (and often better than) TDS based simulators and scales well in multiprocessor mode.

5.6 Design and Performance Tradeoffs

In the process of designing the simulation tool, we had to make decisions about trading accuracy for speed. Our approach was to use the minimum granularity possible for switching between threads, while attempting to minimize the slowdown. Other EDS simulators have chosen a coarser grain of context switching. Other possibilities were on a basic block granularity [CMJS88, Boo94] and at user determined levels of granularity [DG90, Del91]. Proteus [Del91] in particular modifies the C language source code just for certain (shared) memory references, which can cause each processor to have a different value of the global time at any given point in the program, as the processors synchronize time at shared memory points. Our method of task switching on each instruction is the most accurate, yet is not any slower than the other, less accurate, emulators. Considering all the other management features that must be handled by our simulator, such as scheduling processor threads, our simulator does very well in comparison to other simulators that can be run on single processor workstations.

6 Conclusion

The key to understanding how multiprocessor systems work is an accurate model of the interactions between the processors. Many parallel programs use

Microseconds per Reference, MP3D trace							
Cache Simulator	Type	Reference	Simulated Processors				
			1	2	4	8	16
Cerberus with Cache Simulator	EDS	This Paper	49.0	50.9	49.7	47.6	54.9
MPSIM	TDS	[Tho87]	45.8	50.6	51.7	55.0	67.2
DineroIII with Direct Mapped Caches	TDS	[Hil]	41.2				
DineroIII with Fully Associative Caches	TDS	[Hil]	64.1				
Trace File Generation Time (μS per ref)			16.8	15.6	15.0	14.8	14.9
Trace File Size (MB)			24.9	25.0	25.0	25.1	25.3

Table 7: Comparison of **Cerberus** attached to a fully-associative cache simulator (using an adaptive coherence protocol) with various TDS cache simulators.

dynamically scheduled task distribution among the processors, with work queues for load balancing. This can cause the interleaving of memory references to be quite different depending on the target environment. Failure to accurately model processor interactions could lead to mutual exclusion and synchronization violations, as well as incorrect load balancing [Bit90]. To provide an accurate view of program execution, execution driven simulation is the best method for dynamically scheduled workloads. EDS also has the side benefit of eliminating the massive amount of disk space necessary for storing traces.

Cerberus is an EDS-based multiprocessor simulation system that allows program trace generation with a high degree of flexibility and fine grain accuracy without sacrificing performance. It is flexible in allowing the easy attachment of user created tools for code profiling, cache simulation, trace generation and other statistics gathering. The intuitive shared memory programming models used by **Cerberus** lead to easy expression of parallelism in programs.

Cerberus provides efficient simulation of multiprocessors by creating a single UNIX process with lightweight threads for each simulated processor, tightly linked with a user's measurement tool. This eliminates the extra context switches needed by other simulation systems, and allows for very low-level and accurate simulation of the interleaving of processor memory references.

Some trace generation tools have less slowdown, but with some loss of accuracy. **Cerberus** derives its accuracy by simulating instruction-by-instruction; others tools use basic block granularity or instrumented high-level code to synchronize simulated processors. **Cerberus** does not sacrifice efficiency to attain its accuracy. However, since the actual execution time of instrumented code is heavily dominated by the measurement tools (especially in the case of multiprocessor cache simulators), efficiency is generally not a major concern for the address generation subsystem. One of the results of this study shows that execution time measurements of TDS systems show little speed advantage over **Cerberus**. We believe that **Cerberus** is a very good system for accurately and flexibly studying new computer architecture designs.

7 Acknowledgements

We would like to thank Jack Veenstra for his ideas and aid in debugging **Cerberus** during its early development, Windsor Hsu and Jacob Lorch for comments in creating this document, and Siemens A.G. of Munich, Germany for hosting Jeffrey Rothman as a Guest Scientist during early development of **Cerberus**.

References

[AE90] Mani Azimi and Carl Erickson. A Software Approach to Multiprocessor Address Trace Generation. In *Proc. Fourteenth Annual International Computer Software and Applications Conference*, pages 99–105, Chicago, IL, October 31–November 2 1990.

[BDCW91] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.

[BDNS93] Mats Brorsson, Fredrik Dahlgren, Håkan Nilsson, and Per Stenström. The CacheMire Test Bench – A Flexible and Effective Approach for Simulation of Multiprocessors. In *Proc. 26th Annual Simulation Symposium*, pages 41–49, Arlington, VA, March 29–April 1 1993.

[Bit90] Philip Bitar. A Critique of Trace-Driven Simulation for Shared-Memory Multiprocessors. In Michael Dubois and Shreekant S. Thakkar, editors, *Cache and Interconnect Architecture in Multiprocessors*, pages 37–52. Kluwer Academic Publishers, May 1990.

[Boo94] Bob Boothe. Fast Accurate Simulation of Large Shared Memory Multiprocessors. In *Proc. Twenty–Seventh Annual Hawaii International Conference on System Sciences*, Wailea, HI, January 4–7 1994.

[CMJS88] R. C. Covington, S. Madala, J. R. Jump, and J. B. Sinclair. The Rice Parallel Processing Testbed. In *Proc. 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 4–11, Santa Fe, NM, May 24–27 1988.

[Com86] Computer Systems Research Group. *UNIX User's Reference Manual*, April 1986.

[Dah91] Fredrik Dahlgren. A Program-driven Simulation Model of an MIMD Multiprocessor. In *Proc. 24th Annual Simulation Symposium*, pages 40–49, New Orleans, LA, April 1–5 1991.

[Del91] Chrysanthos N. Dellarocas. A High–Performance Retargetable Simulator for Parallel Architectures. Technical Report MIT/LCS/TR-505, Massachusetts Institute of Technology, June 1991.

[DG90] Helen Davis and Stephen R. Goldschmidt. Tango: A Multiprocessor Simulation and Tracing System. Technical Report CSL-TR-90-439, Stanford, July 1990.

[EK88] Susan J. Eggers and Randy H. Katz. A Characterization of Sharing in Parallel Programs And Its Applicibility to Coherency Protocol Evaluation. In *Proc. 15th Annual International Symposium on Computer Architecture*, pages 373–382, Honolulu, HI, May 30–June 2 1988.

[EK89a] Susan J. Eggers and Randy H. Katz. Evaluating the Performance of Four Snooping Cache Coherency Protocols. In *Proc. 16th Annual International Symposium on Computer Architecture*, pages 2–15, Jerusalem, Israel, May 28–June 1 1989.

[EK89b] Susan J. Eggers and Randy H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proc. Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, Boston, MA, April 3–6 1989. ACM.

[EKKL90] S. J. Eggers, D.R. Keppel, E. J. Koldinger, and H. M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proc. 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 37–47, Boulder, CO, May 22–25 1990.

[GH93] Stephen R. Goldschmidt and John L. Hennessy. The Accuracy of Trace–Driven Simulation of Multiprocessors. In *Proc. 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 146–157, Santa Clara, CA, May 10–14 1993.

[GHPS93] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan Jay Smith. Cache Performance of the SPEC92 Benchmark Suite. *IEEE Micro*, 13(4):17–27, August 1993.

[GS94] Jeffrey D. Gee and Alan Jay Smith. Analysis of Multiprocessor Memory Reference Behavior. In *Proc. 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 53–59, Cambridge, MA, October 10–12 1994.

[GW92] Anoop Gupta and Wolf-Dietrich Weber. Cache Invalidation Patterns in Shared–Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[HE92] Mark A. Holliday and Carla S. Ellis. Accuracy of Memory Reference Traces of Parallel Computations in Trace–Driven Simulation. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):97–109, January 1992.

[Hil] Mark D. Hill. DineroIII Cache Simulation Tool. Available from http://www.cs.wisc.edu/~larus/warts.html.

[Kel90] Tom Keller. SPEC Benchmarks and Competitive Results. *Performance Evaluation Review*, 18(3):19–20, 1990.

[KEL91] Eric J. Koldinger, Susan J. Eggers, and Henry M. Levy. On the Validity of Trace–Driven Simulation for Multiprocessors. In *Proc. 18th Annual International Symposium on Computer Architecture*, pages 244–253, Toronto, Canada, May 27–30 1991.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[KYI91] M. Kainaga, K. Yamada, and H. Inayoshi. Analysis of SPEC Benchmark Programs. In *Proc. The Eighth TRON Project Symposium*, pages 208–215, Tokyo, Japan, November 21–27 1991.

[Lac88] Frank Lacy. An Address Trace Generator for Trace–Driven Simualation of Shared Memory Multiprocessors. Technical Report UCB/CSD 88/407, University of California, Berkeley, March 1988.

[LO87] E. L. Lusk and R. A. Overbeek. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, 1987.

[MIP89] MIPS Computer Systems, Inc. *MIPS Assembly Language Programmer's Guide*, May 1989.

[PP84] Mark S. Papamarcos and Janak H. Patel. A Low–Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proc. 11th Annual International Symposium on Computer Architecture*, pages 348–354, Ann Arbor, MI, June 5–7 1984.

[RHL+93] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proc. 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, Santa Clara, CA, May 17–21 1993.

[Rot] Jeffrey B. Rothman. The Cthulhu Trace Generation Package. Man page on Trace Generation.

[RS99] Jeffrey B. Rothman and Alan Jay Smith. An Adaptive Subblock Coherence Protocol for Improved SMP Performance. Technical Report UCB/CSD–99–10XX, Computer Science Division, University of California, Berkeley, Berkeley, CA 94720, September 1999. In preparation.

[SA88] Richard L. Sites and Anant Agarwal. Multiprocessor Cache Analysis Using ATUM. In *Proc. 15th Annual International Symposium on Computer Architecture*, pages 186–195, Honolulu, HI, May 30–June 2 1988.

[Sam89] A. Dain Samples. Mache: No–Loss Trace Compaction. In *Proc. International Conference on Measurement and Modeling of Computer Systems*, pages 89–97, Berkeley, CA, May 23–26 1989.

[Seq87] Sequent Computer Systems. *Sequent Guide to Parallel Programming*, 1987.

[SJF92] Craig B. Stunkel, Bob Janssens, and W. Kent Fuchs. Address tracing of parallel systems via TRAPEDS. *Microprocessors and Microsystems*, 16(5):249–261, June 1992.

[Smi91] Michael D. Smith. Tracing with Pixie. Technical report, Stanford, November 1991. Technical Report CSL-TR-91-497.

[Smi94] Alan Jay Smith. Trace Driven Simulation in Research on Computer Architecture and Operating Systems, (invited paper). In Morito, Sakasegawa, Yoneda, Fushimi, and Nakano, editors, *Proc. Conference on New Directions in Simulation for Manufacturing and Communications*, pages 43–49, Tokyo, Japan, August 1–2 1994.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical report, Stanford, June 1992. Report No. CSL-TR-92-526.

[Tho87] James G. Thompson. Efficient Analysis of Caching Systems. Technical Report UCB/CSD 87/374, U.C. Berkeley, October 1987.

[THW90] Josep Torrellas, John Hennessy, and Thierry Weil. Analysis of Critical Architectural and Program Parameters in a Hierarchical Shared–Memory Multiprocessor. In *Proc. 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 163–172, Boulder, CO, May 22–25 1990.

[TS89] James G. Thompson and Alan Jay Smith. Efficient (Stack) Algorithms for Analysis of Write–Back and Sector Memories. *ACM Transactions on Computer Systems*, 7(1):78–116, February 1989.

[TS90] Shreekant S. Thakkar and Mark Sweiger. Performace of an OLTP Application on Symmetry Multiprocessor System. In *Proc. 17th Annual International Symposium on Computer Architecture*, pages 228–238, Seattle, WA, May 28–31 1990.

[UM97] Richard A. Uhlig and Trevor N. Mudge. Trace–Driven Memory Simulation: A Survey. *ACM Computing Surveys*, 29(2):128–170, June 1997.

[Vas93] Bart Vashaw. Address Trace Collection and Trace Driven Simulation of Bus Based, Shared Memory Multiprocessors. Technical Report CMUCDS-93-4, Carnegie Mellon University, March 1993.

[VF94] Jack E. Veenstra and Robert J. Fowler. MINT: A Front End for Efficient Simulation of Shared–Memory Multiprocessors. In *Proc. Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–207, Durham, NC, January 31–February 2 1994.

[Wil90] Andrew W. Wilson Jr. Multiprocessor Cache Simulation Using Hardware Collected Address Traces. In *Proc. Twenty–Third Annual Hawaii International Conference on System Sciences*, pages 252–260, Kailua–Kona, HI, January 2–5 1990.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH–2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 22–24 1995.

A Survey of Trace Generation Methods

There are a number of other packages for examining and generating processor traces in the public domain. Both hardware and software approaches have been taken to collect traces of multiprocessor behavior. Many of the papers on these methods stress the efficiency of how the traces were collected, but the major issues are the amount of disk space necessary to store the traces, the time to simulate the multiprocessor machine, and the accuracy of the traces. As was demonstrated in the main section of this report, cache simulator time significantly outweighs the overhead due to trace generation in EDS systems.

Two major methods of collection and using multiprocessor traces exist: trace driven simulation (TDS) and execution driven simulation (EDS). TDS implies two steps: trace collection and trace consumption. Trace collection methods span a wide range from software modification of code to hardware probes on the system bus to collect addresses. The traces collected reflect the architecture of the system under test as well as any distortions induced by the collection method. Appendix A.1 describes the different methods of TDS.

EDS provides the ability to combine generating and consuming traces into a single package. Traces can be created by a number of different simulation environments ranging from simulating a multiprocessor system on a uniprocessor machine to using slightly modified multiprocessor operating system software for trace generation. Instead of dumping traces to disk, they are fed into a cache and memory simulator, which allows interaction between the program and the simulated hardware. The benefits of this method are discussed in Appendix A.2. A summary of our survey of the different trace generation tools for multiprocessors can be found in Table 1. A general survey of uni- and multiprocessor trace generation methods can be found in [UM97].

A.1 Trace Driven Simulation

Numerous papers have been written using from the results of Trace Driven Simulation (TDS) for multiprocessors (such as [EK88, EK89b, GS94]). A number of tools have been created to generate traces from multiprocessor programs. Besides potential inaccuracy from timing distortions in predicting performance (discussed in [Bit90, HE92, KEL91, GH93]), one critical problem with TDS is the large trace files required to store long traces. There are several methods which can be used to compact uniprocessor traces. By taking advantage of the spatial locality naturally present in instruction streams, it is possible to use difference encoding to reduce the trace size by up to 100 [Sam89]. In addition, general compression methods such as the Lempel-Ziv algorithm in tools such as **gzip** or **compress** can be applied to the traces after more intelligent difference encoding has been used.

A.1.1 Hardware/Firmware Trace Collection

There have been several attempts to collect memory reference traces by directly monitoring the system hardware, or by modifying the microcode firmware to collect traces. The main advantage of this method is the inclusion of operating system (OS) references in the traces. Among the disadvantages are the non-availability of internal processor state, such as memory references handled by on-chip caches, which does not reach the memory bus where it can be detected. Also, current processors lack microcode which can be modified by users.

Address Tracing Using Microcode (ATUM) was used to modify the microcode of a multiprocessor VAX 8350 to extract user and OS multiprocessor traces [SA88]. Because the traces had to be captured to buffers in physical memory, the length of the traces was limited to under two million instructions. By modifying the OS to be able to collect traces of system behavior, distortions in operational behavior were introduced into the traces. This is a trade-off which must often be made when OS and user code interactions need to be studied.

Wilson [Wil90] collected 14 million reference samples from an Encore Multimax using a hardware interface. The traces from a single processor executing multiprocessor code was captured by sending memory management unit (MMU) generated physical addresses to a special input/output (I/O) card. In an improvement to that method, Vashaw [Vas93] added special hardware to a Multimax to sample up to eight processors, to collect address and data references from the system. Vashaw collected approximately 32 million references from eight programs, including OS references. The OS typically had a small impact on the number of references (the median number of OS references was a 4 percent share), but showed worse behavior than the user code. As was noted, the traces reflected the interference of the measurement technique, as well as the influence of the underlying machine architecture.

A.1.2 Software Generated Traces

Traces were generated from a Sequent Balance 12000 using the UNIX *Ptrace* system call [Lac88]. By single-stepping the program to generate memory references, a slowdown of over 100,000 resulted; thus requiring about two weeks to collect the target of 42 million references. The traces generated in this manner were used by Eggers and Katz in [EK88, EK89b, EK89a].

Tracer [AE90] augments the assembly language of a parallel application to generate data references; however, it was unable to augment system library routines. The UNIX *fork* and *wait* calls were used to run the parallel simulation on a sequential machine. This package requires tasks to be statically assigned to particular processors, and memory accesses were reconstructed from estimating memory access delays from hints in the generated traces.

MPTrace [EKKL90] augments Sequent code to generate traces by directly executing on the machine. Data is put into buffers; once the buffer from a processor fills up, all processors are halted to allow dumping to disk. Trace generation slowdown is quite small, on the order of three to eight times. However, as noted in [UM97], reconstructing the trace from the information saved in this procedure causes a slowdown of an estimated 1000 cycles per reference generated.

A.2 Execution Driven Simulation

Execution driven simulation (EDS), also referred to as program driven simulation or direct execution, offers an approach that instruments code or emulates it so that it calls a simulation package when an interesting event occurs during execution. Code can be directly executed, which can lead to faster run time than can be obtained by interpreting each instruction. A thread scheduler can be used to interleave several processor threads to give the appearance of multiprocessor execution. The level at which the code is modified varies between these approaches; some modify each instruction in the object code, some at the level of basic blocks; some add code to a high-level language to denote shared memory accesses. The method of scheduling the simulated processors varies between the different approaches: implementing a user-level threads package or creating a process for each simulated processor to use the underlying system OS scheduler. Some simulators run on top of multiprocessor machines, allowing each simulated processor to be hosted by a real processor.

There are a number of benefits to using EDS for uni- and multiprocessor tracing. The primary advantages of EDS are accuracy and the ability to produce and consume large traces without reading large trace files from disk. Traces can occupy hundreds of megabytes for any reasonably long sample. Uniprocessor traces can be significantly compressed without losing information [Sam89], but multiprocessor trace generation/compaction is somewhat more complex. Acquisition and release of locks and barrier synchronization can cause timing-dependencies in the generated traces [GH93], and should be abstracted to allow processors in a hypothetical architecture to change to a allow variation in orderings. These synchronization operations must be handled in some manner to provide a reasonably accurate execution order, which is generally not a concern for uniprocessors, as timing dependencies rarely exist. Using a Pixie-like encoding scheme [Smi91] in conjunction with the UNIX utility **gzip**, our tool **Cthulhu** [Rot], which interfaces with **Cerberus** to generate multiprocessor address traces to disk, we achieved a trace compression ratio ranging from 12 to 71, with an average of 31. Even with this compression, trace files containing less than a billion memory references can require more than 100 MBytes of storage. [UM97] provides more details about the compression of multiprocessor traces.

EDS also allows flexibility in changing the hypothetical system under simulation without requiring a new dump of traces to disk. For example, the number of simulated processors can be changed in many EDS systems without much effort, whereas TDS requires a separate trace generation run (with the corresponding large output file(s)) for major configuration adjustments. EDS also allows feedback between the trace generation portion and the system architecture simulator, which is important in determining the correct ordering of events. In addition, EDS allows access to the simulated processor state and the entire contents of memory. It is possible to imagine developing a cache coherence algorithm that avoids write-invalidations when the value to be overwritten is the same as the new value. Such optimizations are not possible to evaluate with TDS. TDS has certain architectural assumptions built into the traces in the form of event timings, which as described in Appendix A.1, may produce inaccurate results when a different environment is simulated.

A.2.1 Interpreters

The DDM simulator [Dah91] consists of a number of simulated MC68000 modules which are connected with memory system modules to simulate a hierarchical bus architecture. The system was written in ADA for expressing concurrent constructs, and suffered a slowdown of approximately 2000 [†]. One of the important points this paper makes concerns the suitability of TDS for multiprocessor simulations. The author argues that TDS for multiprocessors fails to take the dependence of the trace on the architectural assumptions of generation system, which can cause timing distortions when evaluating a different system architecture.

MINT [VF94] is a MIPS R3000 interpreter on which binaries for a parallel machine can be efficiently simulated. By using arrays of function pointers instead of C switch-case statements, it is faster than most systems, whether interpreted or directly simulated. It has a slowdown of between 20 and 70. MINT can be run on systems other than the MIPS, but it requires that the system calls for the MIPS executable be supported on any platform on which it is used. It does have the rather humorous property of being able to interpret the Tango system (described below) faster than Tango can be run directly on a UNIX system.

A.2.2 Modified Code Execution

The Rice Parallel Processing Testbed [CMJS88] offers fast simulation of many memory architectures by inserting code at the beginning of basic blocks and for global communication events. Code written in Concurrent C is instrumented and linked to modules defining the memory system to form an executable file. This method can suffer from a lack of accuracy because the timing ignores local processor events such as cache misses.

The EDS method in Trapeds [SJF92] modifies the assembly language of parallel programs as well as the operating systems on a multiprocessor iPSC/2 (composed of a hypercube interconnect of i386 processors).

[†]The slowdowns reported in this section are for simulators without other subsystems (such as cache simulators) compiled in, with exceptions noted.

The ordering of messages is collected from unmodified code, which is used to enforce the ordering of messages in the instrumented code. This ordering may cause distortions as the simulated hardware parameters vary from those of the underlying machine. The OS kernel can also be modified on the iPSC/2, but also can suffer distortion from extra page faults, etc. The OS was found to consist of a significant portion of the execution time, caused by the kernel calls necessary for message passing.

Due to the poor I/O organization and performance on the iPSC/2, the system is best utilized consuming the traces on the fly with a cache simulator rather than writing the traces to disk. While allowing quick simulation of parallel programs, it is restricted to the memory architectures of the iPSC/2, on which it runs directly. Trapeds was also applied to an Encore Multimax, a shared memory bus-based system using National Semiconductor's NS32532 processors. The traces from this system were dumped to disk with time and synchronization based tags to allow re-creation of the interleaving of the transactions of the processors. There were found to be three types of distortions that disturbed the timing validity of the traces: (1) reordering of original pattern of accesses at synchronization points; (2) modification of time spent waiting at synchronization points; and (3) modification of the ordering of memory accesses between synchronization points.

FAST [Boo94], created at U.C. Berkeley, is a relatively low-overhead simulator with a light-weight threads package, which accurately models low-level thread interleavings, with a slowdown between 10 and 110. The processor threads are interleaved using basic block granularity. It very accurately models the execution time of basic blocks, taking into account interlocks for the floating-point pipeline for operations started in other basic blocks. To avoid difficulties with simulating system libraries, it makes approximations about the duration of the library calls it cannot directly trace.

Tango [DG90] is a multiprocessor simulator created at Stanford University. It uses UNIX processes for each processor in the simulation, which can cause a great deal of context switching overhead depending on the granularity of the simulation. Each process is scheduled by the normal UNIX scheduler, and coordinates with other processes when a shared memory or synchronization event occurs. The disadvantage of this approach is the heavy weight context switches used by the operation system to switch between simulated processors. Tango allows the user to specify the granularity of synchronization between processes (simulated processors) for shared memory event communication. At the least accurate and fastest level, processors synchronize only at *synchronization events*, which are presumably locks or barriers between tasks. The next level of accuracy requires accessing the *memory process* for shared data references as well. The most accurate, but slowest level of granularity requires synchronization for all global and local events. Using this model at the most accurate granularity can cause the simulation to run at up to 20,000 times slower than an unmodified version of the program. A newer version called Tango Lite is available that uses a light weight threads instead of processes. This version is reported to have a slowdown of 45 [GH93].

Proteus [BDCW91, Del91] is an EDS simulator from the Massachusetts Institute of Technology. It is one of the fastest simulators, because only the source code concerning shared memory events is augmented. Calls to the simulator and processor scheduler are added for each shared memory reference. This method of code modification is particularly clever because it uses the C compiler to manage saving and restoring registers when each processor context is changed, which is necessary only for calls to the simulator. The result of this high-level management is a typical slowdown of 35–100, but at the cost in the accuracy of instruction timing. Processors are allowed to drift from the correct global time for efficiency purposes. It uses a light weight thread scheduler to avoid the context switch overhead in Tango.

The Wisconsin Wind Tunnel [RHL+93] directly executes parallel binaries on a CM-5 machine. Their method allows most code to run directly without modification on the system, using the error correcting code (ECC) bits to trap accesses to shared memory. Only in the case of shared memory misses does extra simulation code run, caused by an ECC trap. A software layer is added to provide the abstraction of shared memory on top of a message passing machine. Since each simulated processor actually runs on a real processor, the simulator is extremely fast. The slowdown reported for this system ranges from 52–250 *with* the memory simulator connected to the trace generation subsystem. One of the side effects of this simulation is possible inaccuracies of the shared memory latency due to the coarseness of timing with message passing. In addition, use of small caches (less than 32 Kbytes) can cause significant slowdowns (30–40 without the cache simulator) because the OS trap handler is invoked frequently [UM97].

B Implementation Difficulties

A number of issues arose during the creation of the simulator which caused programming difficulties. A summary some of the chief difficulties that had to be overcome and our solutions for the implementation of the project follows:

1. Although object code produced by FORTRAN is very similar to C object code, there are a few important differences. The chief issue is the method of passing parameters. FORTRAN passes by reference, i.e., uses pointers to all parameters, including constants. The compiler puts the constants in the text section to make the values read-only. Although the **rdata** (read-only data) section is available in the MIPS **a.out** executable format for that purpose, the compiler sticks the constants at the beginning of the procedures that make function calls using the constants. It was necessary to determine which items in the text section were instructions and which were constants. It was also necessary to make sure that the constants were not modified as the instructions are and that the address passed to the cache simulator reflected the same address as in the original code.

2. Non-local gotos (**setjmp** and **longjmp** pair) were found in the SPEC92 benchmark Xlisp [Kel90, KYI91]. This pair allows jumps out of an arbitrary function call depth by **longjmp** to the point in any function set by **setjmp**. A fair amount of information must be saved to allow jumping into the middle of a function. This environment information consists of the "saved" integer and floating-point registers (those preserved across function calls), PC, stack, and floating-point control register values. When longjump is called, these values must be restored to the appropriate registers to restore the environment to allow the function jumped into to continue operating properly. A special routine is used to detect when longjump has been called and update the simulated registers.

3. Most system calls can be directly made from the modified code (typically from library routines), but the low-level memory allocation system calls **sbrk** and **brk** must be intercepted. Both modified and unmodified versions of the standard library exist in the same executable file, and both sets of the libraries are actively used. Most of the time this does not cause any problems and is necessary, but it does not work with **sbrk**. Both the modified and unmodified versions of the standard library function **sbrk** maintain a pointer to the highest location of the data segment that has been allocated (the **break point**). This value becomes inconsistent when there are two versions of **sbrk** running. The break point can occasionally be decreased by the inconsistency, causing segmentation faults. More importantly, it is useful for correctness and debugging reasons for each processor to have its own contiguous range of private memory. A special local memory allocator is used instead of **sbrk** in the modified code to implement this feature and to avoid the multiple **sbrk** problem.

4. Branches are limited to distances of ± 131072 ($\pm 2^{17}$), because of the 16-bits allowed for destination specification (since the address has to be word aligned, it provides for an 18-bit signed integer displacement range). Any branch with a displacement exceeding 16 Kbytes in distance in the original code needs to be restructured in the modified code to work correctly, because when the code is augmented, the distance is multiplied by 8, which exceeds the range of a branch instruction. This was solved by replacing affected branches with an unconditional jump, that jumps to a routine which tests the branch condition, and resumes execution at the correct place both for taken and not-taken branches. Code space is reserved at the end of the modified code to put the branch and jump instructions for each long branch encountered. It is rare that branches need to be re-vectored in this way.

5. One issue that only occurs in multiprocessor mode is the reaction of the C compiler to the keyword **volatile**. This type qualifier is used for shared variables to force reads from shared memory to always fetch values from memory (as opposed to caching it in a register) before the value is tested. The optimizer tends to cache the value in a register, and this precludes coherent alteration of the value by another simulated processor. The **volatile** qualifier is not acted upon by some C compilers (such as the standard **cc** compiler supplied with Ultrix), and the result can be an infinite loop or other problem for some of the simulated processors. This is due to the fact that the cached value cannot be updated by other processors. Using ANSI compilers (such as **gcc**) takes care of this is-

sue. It is still necessary to watch for spins on shared-memory variables, to make sure that optimization does not interfere with proper operation. This problem has been observed only in **LOCUS**.

6. The existence of branch delay slots causes an occasional problem. Because the delay slot instruction must be expanded into eight instructions and executed before the branch takes place, the delay slot instruction is moved before the branch instruction in the modified code. But the branch condition may be affected by the values computed in the delay slot, so it is necessary to use temporary registers to avoid conflicts.

   However, some optimizers target branches to the instruction in the delay slot, which saves one instruction in the executable, but has no effect on the dynamic number of instructions executed. Figure 9 shows an example of code with such a branch. In the case of a branch into the delay slot, the delay slot instruction is executed independently of the branch instruction whose delay slot it occupies; the delay slot instruction acts like the beginning of a basic block. For correct execution in this odd case, the delay slot instruction, which has been moved before the branch by the code modification process, must skip the branch instruction and continue execution of its basic block. It is necessary to determine whether the delay slot instruction should allow execution to continue to the next eight instruction block containing the branch instruction (normal case) or to jump over the block of code containing the branch.

   This problem was solved by using the delay slot instruction of the branch that targets the delay slot to load special registers which dynamically determine whether to skip the branch. But the targeted delay slot can also be needed for this special role, causing more complexity. In Figure 9, the instruction at address 0x10007b7c is the target of a branch into the delay slot. In the modified code, when this instruction is branched to, for correct execution it must be ensured that the branch at 0x10007b78 is skipped. To determine when to do this, the delay slot of the targeting branch (at 0x10007b64) loads a special value into a location in the processor's context block, indicating that this special branch might occur. This value is detected at 0x10007b7c if the branch occurs; it is not detected if the delay slot instruction is executed as the delay slot of the branch at 0x10007b78. If this problem sounds confusing, it was certainly no fun to resolve, but it does work correctly.

7. Program symbols are inconsistently defined by the C compiler. This project was developed on several MIPS based platforms, and depending on the operating system and the compiler, the **a.out** format symbol definitions vary. According to the MIPS manual [MIP89], symbols that are undefined are placed in the external symbol table, while defined symbols reside in the local symbol table. On one system, variables stayed in the external table even after they were defined, which does not follow the correct MIPS rules. Another related issue is that some of the compilers have different types of symbols, which are incompatible with other compilers, specifically those related to **struct** definitions. This caused problems as the project was ported between different MIPS platforms. The solution to this problem was to use C macros for conditional compilation, which can be used to include the appropriate C routines for each type of OS and compiler.

8. Running the trace generator under the debugger, on different machines, or with different path names to input files can generate slightly different results. It was disconcerting to find the miss ratios were different for runs of the same workload and cache simulator on different machine, but only on the order of two or three extra misses out of millions of references. This problem was tracked down to the initial value of the stack pointer. This can differ between machines of the same type, as well as if the program is run using a debugger. The value typically differed by a small amount (e.g., the initial value of the stack pointer **sp** varied by 16 for two machines in our research group: 0x7fffbac8 vs. 0x7fffbab8 running MP3D under the debugger on two workstations under otherwise identical circumstances). This is enough to affect the mapping of some stack variables to cache blocks. Fortunately this difference is so small that it has very little impact on the results, changing the number of cache hits or misses by around 3 out of more than 100,000,000 references.

```
(../doopen.c:  110) 0x10007b5c:  li r4,43
(../doopen.c:  110) 0x10007b60:  beq r4,r2,0x10007b7c % branch to delay slot
(../doopen.c:  113) 0x10007b64:  r1,98
(../doopen.c:  110) 0x10007b68:  r2,r1,0x10007ba4
(../doopen.c:  113) 0x10007b6c:  nop
(../doopen.c:  113) 0x10007b70:  lb r11,2(r7)
(../doopen.c:  113) 0x10007b74:  nop
(../doopen.c:  113) 0x10007b78:  bne r4,r11,0x10007ba4
(../doopen.c:  113) 0x10007b7c:  li r1,-2 % delay slot and target of branch
(../doopen.c:  113) 0x10007b80:  lh r12,16(r3)
```

Figure 9: Code snippet which exhibits a branch into a delay slot.

C Installing Cerberus

The **Cerberus** tar file contains four main subdirectories: **modCode**, **appl**, **cache**, and **docs**. **modCode** contains source code for the code modifier **modCode**, the simulator libraries and routines in C and MIPS assembly language, and a subdirectory called **parallel**, which contains some include files. Simply typing **make** in the **modCode** directory will compile all the tools and libraries necessary for parallelization. Certain variables, such as the **DIR** variable, should be adjusted to point to the **modCode** directory.

The **appl** directory contains the global Makefile included by the Makefiles of the applications to be parallelized. The applications can be placed in subdirectories of **appl**, but that isn't required. In all the Makefiles, it is possible to adjust the Makefile variables to point to the appropriate directory locations, such as the **include** statement in Appendix F.1.

When a program to be simulated is created or un-tarred into a directory, it is necessary to make a symbolic link from that directory to the subdirectory **modCode/parallel** to allow the Makefile to find the proper include files. In addition, it is necessary to have a file called **paramFile** in the workload's directory, which is used for passing command line switches to the cache simulator (specified in Appendix D). These can all be seen in the sample **mp3d** subdirectory of **appl**.

The **cache** directory holds a simple multiprocessor cache simulator stub which keeps track of some simple statistics (instruction, load, store, lock, and unlock operation counts). It provides the minimum code necessary for unpacking the data structure containing all the address references, and handling the lock management package. We found it necessary to have the cache simulator perform the actual locking operations, since there was no way for the cache to be sure a lock was going to take place unless it modified the lock directly. It also makes it possible to cleanly attach a trace reading (TDS) program as the front end to the cache simulator interchangeably with a **Cerberus** generated front end (such as **Cthulhu** [Rot]).

In the **docs** directory will be found a copy of this Technical Report as well as any other notes that aid in running the **Cerberus** system. In addition, there is a **Cerberus** website at http://www.cs.berkeley.edu/~rothman/cerberus, where contact information, updates and other information can be found.

D Using Cerberus

Common Commands in *paramFile*	
#	rest of the line is a comment
-c *switches*	commands passed to the cache simulator
-n *n*	maximum number of simulated processors
-mem *n*	memory allocated (shared and local)
-mems *n*	shared memory allocated
-meml *n*	local memory allocated per processor
-meml0 *n*	local memory allocated for processor 0
-sharedglobals	use lightweight threads (shared globals)

Table 8: Some of the commands passed to the thread package and the cache simulator.

The first step in compiling Sequent code for **Cerberus** requires using a lexical analyzer (generated by **new_get_shared.l**) to convert shared memory variables with the **shared** type qualifier to variables with the word **shared_** prepended. This makes it possible to determine in the modification stage which variables are shared in order to target the correct part of the memory space. This also allows use of a regular C or FORTRAN compiler to generate object code. A lexical analyzer is necessary because of the necessity of processing C grammar for all the possible ways the **shared** type qualifier can be incorporated into variable declarations.

In the process of compiling the source code, it is necessary to create an object file with full debugging information in it (using the -g or -g3 options). The only restriction on the contents of the object file is that the function **main** must be renamed to **app_main** and must be the first function of the object file that serves as input to the code modifier. The combined object file must be linked together using **ld** with the -r and -d options, to force definitions of all symbols, and to keep relocation information in the file. There are also three library files that have to be linked-in to help support some of the tracing functions and to provide new multiprocessor versions of such routines as **printf**. These files are **pps.o**, the pseudo parallel library; **newprint.o**, the locked print routines; and **headers.o**, a special files that defines symbols for the code modifier as well as the data space for the special array (SIMrelptr) of relocation entries.

Once the object files are linked together, they are passed through **Cerberus**'s code modifier (**modCode**), which among other tasks augments the code by inserting calls to the appropriate simulator entry points for each instruction of the original code. To aid in debugging, the symbol table information and the C line number data are updated to accurately keep the correspondence between the original C or FORTRAN source code with the modified assembly language instructions.

The final stage in the process links the scheduler, helper functions and other tools (such as a cache and network simulator) to the augmented code to create the final executable. This program can be executed directly on the MIPS platform, and will generate trace information as the program executes.

To avoid the necessity of recompiling and relinking the executable each time the cache simulator configuration changes, we have a special method of passing arguments to it. A special file, called **paramFile** must exist in the directory from which the simulation is being run. Commands can be passed to the cache simulator or the thread scheduler package to override certain defaults. Table 8 shows the commands that can be placed in **paramFile**.

There are a handful of functions that **Cerberus** expects the cache simulator to provide. Some of these may not be used in a particular simulation, but they must all be present:

- **void user_init(int* pargc, char** argv)**: used for passing switches to the cache simulator, similar to the **main** function, except the pointer to the number of arguments (pargc) is passed instead of the number of arguments.
- **unsigned user_sim(CYCLE_INFO* pci)**: called every time step with a pointer to the data structure that contains the memory addresses (specified in Appendix E). It returns a bit vector, with a bit set for each processor that is stalled (processor 0 is bit 0, etc.).
- **void user_multi()**: called right before the simulator enters multiprocessor mode for the first time.
- **void user_done()**: called at the end of the simulation, to print statistics to a file or clean up.
- **void user_set_procs(int num)**: called with the new number of active processors any time the number of processors change.

E Low-Level Details

E.1 Trace Format

At each time step, a pointer to a data structure of type **CYCLE_INFO** with trace information in arrays of type **TRACE_INFO** is passed to a routine called **user_sim**, which is the front end for a cache simulator or other tool. The data structures are defined as:

```
typedef struct t_info
{
  int proc,type,addr;
  unsigned shared:1;
} TRACE_INFO;

typedef struct c_info
{
  int num;
  TRACE_INFO data[MAXNUMPROCS*2];
} CYCLE_INFO;
```

The **num** field specifies the number of entries to be processed during the current cycle. There are one or two entries per active processor; the second entry contains the data memory address for load-store instructions.

The **trace_info** data structure contains information about each memory reference that is passed to the cache simulator. There are four fields: one that identifies the processor, one containing memory address of the reference, a field specifying the type of operation to be performed, and a bit indicating whether the reference is to the shared region of memory. The operations consist of read data, write data, instruction read, and (un)locking operations that have the ability to coordinate with the cache simulator to determine if the lock is already locked. All operations are assumed to be operating on word size (4-byte) data; this simplifies the design of the system. The **shared** bit is used for debugging purposes and is set if the memory access is in the region of memory determined to be the shared section.

The return value from the cache simulator is a bit vector which specifies which processors hit in the cache or are stalled. Processors are stalled (inactive) while they are waiting for the cache (bus operation), waiting for a synchronization event (such as a synchronization barrier, **m_sync**), or while the program is in a special single processor mode (between **m_single** and **m_multi**). Each processor can be stalled independently of other processors.

The trace generator is directly linked with the cache/memory simulator or other tool, putting all the pieces into one executable file. Only one UNIX process for the simulation is required. There are no large trace files necessary for **Cerberus**, as the reference stream generated by the augmented code is immediately consumed by the attached cache simulator. Another advantage of **Cerberus** is that full data address values are generated and sent to the cache simulator, resulting in more accurate cache simulations. Some TDS tracing tools (e.g., Pixie) pack trace information into 32-bit chunks, which consist of 24 bits of address, 4 bits of instruction count and 4 bits of reference type information. The most significant upper 8 bits of address are eliminated, which may cause aliasing problems between instruction and data addresses.

For uniprocessor operations, one of the more efficient tools for MIPS workstations is Pixie [Smi91]. The advantage Pixie has over our simulation method (besides speed) is the type of programs on which it can be used. Pixie can be used on any executable file. Our method requires that the source code be available (or at least in the form of an object file that still has all the symbolic and relocation information in it). We used Pixie for some of the sequential simulations in our research, but only those for which we did not have the source code. We have found that **Cerberus** is generally more robust than Pixie in tracing code. Pixie occasionally breaks on some programs, **Cerberus** works on all workloads for which we have the source code. However, **Cerberus** is significantly slower than Pixie at generating uniprocessor traces.

E.2 Simulated Context State

The context for each simulated processor contains a copy of that processors' version of the register file as well as other processor state and information to assist the thread scheduler. This context, which we interchangeably refer to as the *register file*, contains 128 entries kept in a 2 dimensional array called **context_block**. Size 128 was chosen for two reasons: because nearly 100 registers are used to maintain context information for each processor, and using a power of 2 number of registers saves instructions in calculating the absolute memory address of array locations (one shift instruction can replace an integer multiply or multiple additions). When control is passed back from the scheduler to the augmented code, a slice of **context_block** corresponding to the particular simulated processor is passed in register **r2**.

Here are the uses of the various array locations in **context_block**:

- 0–31: simulated integer register file
- 32–63: simulated floating-point register file
- 64–65: the LO and HI registers used for fixed point math
- 66: floating-point unit control register
- 67: program counter
- 68: return point in assembly language interface routine
- 70–71: temp space
- 72: pointer to the bottom of the processor's stack
- 73: stack size
- 74: processor ID
- 78: used for detecting jumps into branch delay slots
- 80: local data offset (redirection register)
- 81: shared global pointer (sgp)
- 82: pointer to the base of local data
- 84: debug register

When these locations are referenced in the assembly language file, they are multiplied by 4 to account for the 4-byte size of integers. For example, Figure 6 mentions the word at 320 offset from the context block, which when divided by 4, is simulated register 80, the local data offset.

E.3 Interfacing Augmented Code with the Scheduler

The block of eight instructions generated for each original instruction must call the simulator, load simulated registers from the context block, perform the original instruction's actions, and save the results. The call to the simulator entry point needs to pass the address of the instruction and the address of the data memory accessed (for load-store instructions) and requires two or three instructions. The remaining five or six instructions are usually sufficient to perform the remaining tasks. In the cases where eight instructions are not enough to load and save state (such as floating-point operations), special tightly coded assembly language routines are utilized to perform the tasks. The routines will be described below and some are discussed in detail in Appendix B.

The entry points to the simulator are assembly language routines which gather the address information into the proper form for calling the C routines according to the MIPS register usage conventions. The stack pointer (sp) and global pointer (gp) are loaded with appropriate values to allow proper execution of the C routine. The context block for the simulated processor is also updated with information to aid in resuming execution when the processor thread is rescheduled. Some of the information saved to the context block is necessary due to the complexity of simulating multiple processor threads. The multiprocessor scheduling mode is unlike the uniprocessor mode of simple call and return, in that it requires the scheduler to actively pick the next thread to run. This in turn requires the return location within the assembly language routine to be saved. The return address back to the modified eight instruction block of code also needs to be saved.

Some assembly language routines, besides providing the above mentioned functionality, must also handle cases where eight instructions are not enough to both perform the actions of the original instruction and provide information to the simulator. These cases are floating-point operations, branch delay slots which are targets of branches (discussed in Appendix B), calls to functions which are not traced (requiring setup of the real CPU's registers **r4**–**r7** with values from the simulated registers), system calls, and calls to **setjmp-longjmp** (a non-local goto).

The most commonly used simulator entry calls, which interface between the augmented object code and the C language simulator routines, are **pSIMstep**, **pSIMmem** and **loadfsft**, which are 11, 12, and 32 instructions long, respectively (with some small variance). **loadfsft** is used by all floating-point operations, **pSIMmem** is used by all memory operations (except for locking operations) and **pSIMstep** is used by most other instructions. All the assembly language entry points to the simulator can be found in **pSIM.s**. The file **pFIG.s** contains assembly language routines that aid the simulator in performing low-level operations, such as preparing the threads during a call to **m_fork** or **s_fork**, killing a thread that is finished, lock operations, etc.

F Makefile

Here are some sample makefiles we use in compiling for **Cerberus**:

F.1 Example Makefile for MP3D

This file is a modification of the file provided from the SPLASH collection:

```
# Makefile for mp3d
MODTARG = mp3d
GFLAG = -G 16
CCFLAGS = -g3 -c -O2 ${GFLAG} -DLOCKING

SHDIR = ${CACHE}

OBJS = mp3d.o setup.o adv.o
include ${HOME}/appl/Makefile.common

sh${MODTARG}: ${OJBS}

adv.c: adv.C parallel/parallel.h common.h
mp3d.c: mp3d.C parallel/parallel.h common.h
setup.c: setup.C parallel/parallel.h common.h

adv.o: adv.c parallel/parallel.h common.h
mp3d.o: mp3d.c parallel/parallel.h common.h
setup.o: setup.c parallel/parallel.h common.h
```

F.2 Global Makefile

This file is common among all the workloads simulated:

```
# Makefile.common
CC = gcc
SHELL = /bin/sh
LDFLAGS = -r -d

# locations of modCode files
MACROS =  ${HOME}/appl/monmacs/lib/c.m4.seq
MODDIR = ${HOME}/modCode
MODLIBS = ${MODDIR}/pps.o ${MODDIR}/newprint.o
MODHEAD = ${MODDIR}/headers.o
MOD = ${MODDIR}/modCode
SIM = ${MODDIR}/sim.o

# cache simulators we might want to use
CACHE = ${HOME}/appl/cache

SHFILES = ${SIM} ${SHDIR}/cache.o
to_do: sh${MODTARG}.o mod_sh${MODTARG}.o \
    mod_sh${MODTARG}

# how to turn .U and .H files into object code
.SUFFIXES:
.SUFFIXES: .o .c .h .H .C .U
.H.h: ; m4 ${MACROS} $*.H >$*.h
.U.C: ; m4 $(MACROS) $*.U >$*.C
.C.c: ; ${MODDIR}/make_seq $*.C
.c.o: ; ${CC} -c $(CCFLAGS) $*.c

# dependencies
mod_sh${MODTARG}: mod_sh${MODTARG}.o ${SHFILES} ${OBJS} ${EXTRATOOLS} dummy
  ${CC} -o mod_sh${MODTARG} mod_sh${MODTARG}.o ${SHFILES} ${EXTRATOOLS} -lm

mod_sh${MODTARG}.o: sh${MODTARG}.o ${MOD} ${OBJS}
  ${MOD} sh${MODTARG}.o

sh${MODTARG}.o: $(OBJS) ${MODLIBS} ${MODHEAD}
  ld ${LDFLAGS} -o $@ ${OBJS} ${MODLIBS} ${LIBS} -lc -lm ${MODHEAD}

dummy:
  gmake -C ${SHDIR} cache.o


clean:
  rm -f $(OBJS) sh$(MODTARG}.o mod_sh$(MODTARG}.o mod_sh$(MODTARG}
```