MINT Tutorial and User Manual

Jack E. Veenstra Robert J. Fowler

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 452

June 1993
(Revised August 1994)

Abstract

This document describes MINT, a software package designed to ease the process of constructing event-driven memory hierarchy simulators for multiprocessors. It provides a set of simulated processors that run standard Unix executable files compiled for a MIPS R3000 based multiprocessor. These generate multiple streams of memory reference events that drive a user-provided memory system simulator. MINT uses a novel hybrid technique that exploits the best aspects of native execution and software interpretation to minimize the overhead of processor simulation. Combined with related techniques to improve performance, this approach makes simulation on uniprocessor hosts extremely efficient.

This material is based upon work supported by the National Science Foundation under Grant number CDA-8822724. The U. S. Government has certain rights in this material.

Contents

1 Introduction

The simulation of high-performance computer systems is computationally expensive. Each unit of simulated processor execution time requires many units of simulator time. If the target is a parallel computer, simulator time necessarily grows in proportion to the total amount of work done in the simulated parallel machine. On top of this, simulators can incur substantial overhead in scheduling and dispatching the large number of concurrent activities within a parallel machine. The overhead problem is especially bad when the purpose of simulation is the evaluation of memory hierarchy designs for multiprocessors; system components potentially interact on every memory reference operation. Given the high cost of simulation, it is vital that the multiprocessor aspects be efficient.

MINT is a software package on top of which multiprocessor memory hierarchy simulators can be constructed. The principal motivation for MINT is to reduce the total cost of simulation over many runs, rather than trying to improve the turnaround time of a single simulation run by introducing internal parallelism and running on a parallel machine. Internal parallelism introduces overhead not present in sequential programs; thus, MINT is designed to be a highly efficient sequential simulator suitable for use on low-cost workstations. MINT provides direct support for spinlocks, semaphores, barriers, shared memory, and most Unix system calls. Extending the instruction set and adding new system calls can be done easily.

MINT runs on MIPS-based machines including those from Silicon Graphics and Digital Equipment Corporation. The source code for MINT is available by anonymous ftp from cs.rochester.edu and includes a tutorial and user manual.

1.1 Trace-driven simulation

Memory reference traces of parallel programs have been used to drive a wide variety of multiprocessor memory simulators. Each trace contains a set of memory references for a fixed number of processors, and the references are presented to the simulator in the order in which they were recorded. (See Figure 1.) While trace-driven simulation is reproducible and efficient, it has several disadvantages. There is usually no feedback from the simulator to the trace file during simulation, so the behavior of the simulated system cannot change the timing or interleaving of memory references in the trace, nor is it possible to model situations in which timing differences can cause programs to take different branches than they did in the original execution. This limits the range of validity to systems very much like the one on which the trace was collected. The trace may not even be accurate for the system on which it was collected, since the very action of trace collection slows down some parts of the program and may perturb the interleaving of references [5]. A practical disadvantage is that each trace file can require hundreds of megabytes of disk space, though there are clever techniques for reducing the storage requirements [7].

1.2 Program-driven simulation

Program-driven simulation [3] overcomes these problems by performing the simulation as the traced program executes. The simulator controls the scheduling of processes so that

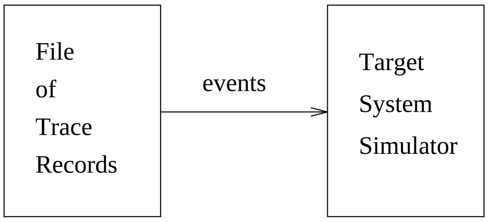

Figure 1: A trace-driven simulator reads a fixed sequence of trace records from a file.

the interleaving of memory references is the same as it would be on the simulated machine. Furthermore, there is no need to store a trace file on disk.

A program-driven simulator can be partitioned into two main parts: a memory reference generator (the "front end"), and a target system simulator (the "back end"). (See Figure 2.) The reference generator models the execution of an application program on some number of processors. When the program performs an interesting operation, typically the generation of a memory reference, the front end sends an event to the back end. The back end models the system interconnect and the memory hierarchy. When the operations for an event complete, the back end signals the front end that some process can continue. A third component, though less visible, is an underlying simulation library that manages and schedules events and processes. The target system simulator is the part that is specified by and is of the most interest to a designer or researcher. From the perspective of a memory system researcher, the reference generator and simulation library components are a source of overhead, both in terms of execution time and in terms of the time to write and maintain them. Therefore, it is desirable to have an efficient simulator package that allows a researcher to focus on the memory system and interconnect.

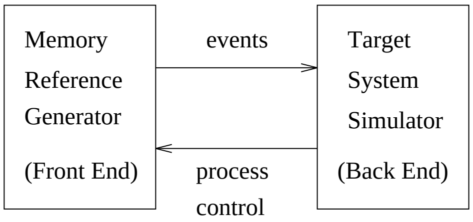

Figure 2: A program-driven simulator contains two major components: a memory reference generator and a target system simulator.

MINT ("MIPS interpreter") encompasses the memory reference generator and simulation library components of a program-driven simulator in an integrated package. MINT has been carefully engineered for performance by using a novel simulation technique, tuning the speed of each component, and by choosing mechanisms that interact efficiently. MINT is designed specifically for multiprocessor simulation, so the internal mechanisms are tuned for that task.

1.3 Execution-driven simulation

Execution-driven simulation [3] is a special case of program-driven simulation. Execution-driven simulators modify the application to be simulated by inserting calls to the back end functions directly into the application source or object code so that the execution of the program generates the memory trace. This technique usually requires preprocessing and recompiling the application.

In contrast, the input to MINT is a statically-linked Irix executable file compiled for the MIPS R3000 processor [6]. It is not necessary to instrument application code for simulation or to recompile. This allows the widest possible range of programs to execute on top of MINT and ensures that all user-level code is simulated, including libraries for which source is not available or for which recompilation is not practical.

2 Related work

Many researchers have built simulators for message-passing multicomputers. Their simulators take advantage of the properties of message-passing machines to achieve good performance and are not designed to simulate shared-memory machines. Trapeds [12] collects multicomputer traces for the Intel iPSC/2. It modifies the assembly language to insert calls to a function that analyzes the previously executed basic block. The Rice Parallel Processing Testbed (RPPT) [3] uses execution-driven simulation to model message-passing architectures. The input is a Concurrent C program, an architecture specification, and a mapping of processes and data to processors. The Concurrent C program is preprocessed to insert calls to a user-defined function whenever processes interact, such as when sending or receiving messages or accessing semaphores.

Shared-memory multiprocessors are harder to simulate than message-passing multicomputers because every reference to a shared location by some process can affect the state of other processes and potentially requires context-switching to the process that is the furthest behind in simulated time. In addition, every memory reference typically requires a check at run-time to determine whether the memory location is shared. In contrast, the interaction between processes on a message-passing machine is limited to sending and receiving messages.

PROTEUS[2] is an execution-driven simulator for shared-memory machines that simulates multiple processes in a single address space. The input to PROTEUS is an annotated C program that is preprocessed to produce an instrumented C source program. The annotations are language extensions that identify accesses to shared memory. For example, the unary "@" operator must be used instead of the unary "*" operator when accessing shared memory through a pointer. The preprocessor inserts a call to a simulator function at each shared memory reference, so it is important that all such references be annotated. In the current version, annotation is done manually. A performance comparison of PROTEUS with MINT is given in Section 9.

The Wisconsin Wind Tunnel [10] is a fast execution-driven simulator for distributed shared-memory machines. It runs in parallel on a CM-5 host (a SPARC-based multicomputer) and uses the ECC (error-correcting code) bits as "valid" bits for cache lines within a page to trap accesses to shared memory. References that miss in the cache cause a trap to the simulator. Since a distributed shared-memory machine uses network messages to keep memory coherent, the minimum delay between sending and receiving a message can be used to define a time quantum. If, at the beginning of a time quantum, each process waits for all outstanding messages to be received, then during the time quantum no interaction between processors is possible. This allows each process to execute natively for a whole time quantum or until it generates an event. This works well when the time quantum is large, but is less efficient for simulating multiprocessors in which the latency of interprocessor communication is small. An important limitation of this approach is that contention cannot be modelled accurately since each processor must wait for all messages to be delivered before simulating the next time quantum. Although the WWT is fast, the hardware platform required to run it is expensive.

Tango [4] is another execution-driven simulator that uses a separate Unix process for each simulated process. Although multiple processes are used, no real parallelism is exploited. Only one process executes at a time; the others remain blocked. Using separate processes in this way avoids problems with remapping memory addresses at run-time, but severely increases the cost of synchronization. A performance comparison of Tango with MINT is given in Section 9.

3 Design choices

We describe some of the performance-related design choices that must be made by writers of reference generators for program-driven simulators and we present the solutions that are used in MINT.

3.1 The representation of processes

When the program being simulated creates a new process, for instance by calling the Unix `fork()` operation, the writer of the reference generator must choose between (1) creating a new operating system process, and (2) creating a representation (such as a lightweight thread) of the simulated process within the same address space. Tango uses the first approach, while MINT and PROTEUS use the second approach.

Since each Unix process has its own address space for private data, creating a new process is attractive because it eliminates the need to manage multiple copies of private data and stack space. The cost of coordinating and scheduling multiple Unix processes, however, is much higher than it is for multiple threads within a single process. Multiprocessor simulations typically require synchronization at least at every shared memory reference and also at private memory references if cache interference is modelled. The cost advantage of handling these interactions within the simulator rather than in the operating system on which it runs is well worth the added complexity of using threads in a single address space.

3.2 Simulated program address space

The decision to simulate all processes running on the target machine within a single address space in the simulator leads to several other problems. For example, when a parent process forks a child process under Unix, the virtual address of a global variable in the child process is the same as the virtual address in its parent, but the parent and child each have separate physical copies of the global variable. If all of the simulated Unix processes are within a single simulator address space, then when the simulated parent process forks a child, the parent's data must be copied to a new location so that memory writes by the child do not change the parent's data.

Since copying the parent's data changes the virtual addresses that the child must use to access its copy, addresses used by the child must be changed. This leads to another design decision: (1) all of the child's addresses can be relocated at fork time, or (2) the addresses used in loads and stores can be dynamically mapped as they are used. The first option requires that pointers to private data be relocated so that the child does not write to the parent's private data through a copied pointer. This requires distinguishing between private and shared data. Not all pointers, however, can be relocated. For example, pointers stored in dynamically allocated memory cannot be relocated because there is nothing to distinguish pointers from other data types.

MINT does not attempt to relocate data, but dynamically translates addresses during simulation. This avoids the complexity required by relocation and allows MINT to simulate any program correctly. Another benefit of dynamic translation is that both the original and the translated addresses are available to the simulator. With relocation, the original address is not available during simulation (unless run-time translation is used, in which case there is no advantage to doing the relocation).

PROTEUS avoids this problem by using a different programming model. In PROTEUS, all global variables are assumed to be shared. Private global data must be allocated dynamically by each thread, and no data segments are copied at thread creation time. This eliminates the data relocation problem, but makes it more difficult to simulate programs (such as the SPLASH suite) that assume the semantics of Unix `fork()`.

3.3 Native execution vs. software interpretation

Two approaches to simulating the execution of an instruction are to use software interpretation, or, if the host and target instruction sets are the same, to use native execution on the host machine. Interpreters represent the state of the simulated processor as variables in memory. In contrast, under native execution the host machine state is used to hold the state of the simulated processor. Sometimes the host's registers hold the processor state of a process of the simulated program and sometimes they are used for the internal functions of the simulator. Each time a simulator function is called it is necessary to save and restore register sets. Since each call to the simulator might entail a switch to another simulated context, all of the active registers must be saved. Native execution thus trades a very low cost of actually executing each simulated instruction for a much higher cost of switching among simulated contexts and simulator code. If a large number of instructions is executed between context switches, then this is a good tradeoff. If the simulator is called frequently, however, the total cost of native execution can be higher than interpretation.

To take advantage of the best features of software interpretation and of native execution, MINT uses a hybrid approach. The target processor state is kept in memory as much as possible. To reduce the cost of loading registers from memory on every instruction, MINT uses a form of *code synthesis* [9] to encapsulate each straight-line block of code that generates no events internally.[1] For each such block, MINT creates a new function when the object code of the program is loaded. The synthesized function loads the necessary operand registers, natively executes the target instructions, and stores any modified registers back to memory. Using this approach allows instructions to be interpreted at native speed, while eliminating unnecessary context switching overhead.

3.4 The form of program input

Another advantage of interpretation over native execution is that programs to be run on the simulator need not be modified. The program is compiled separately from the simulator and the simulator treats the resulting object code as input data.

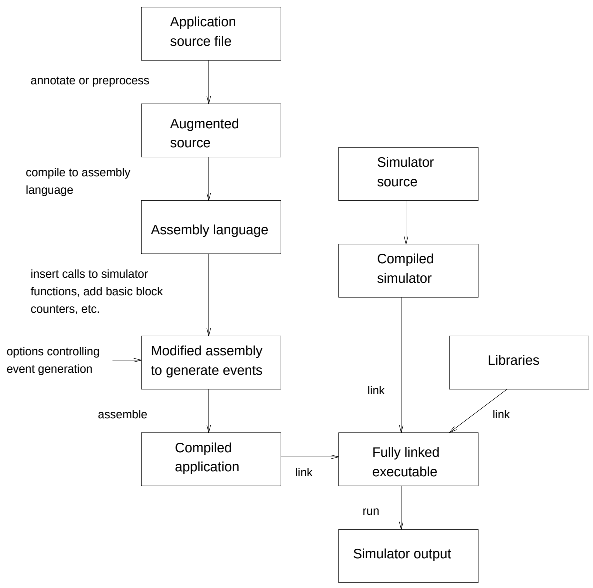

Figure 3: The procedure required to simulate an application for a typical execution-driven simulator. Each application must be compiled specially using this procedure.

In contrast, execution-driven simulators that use native execution, such as Tango [4], PROTEUS[2], and RPPT [3], work by inserting calls to the simulator directly into the program to be simulated so that the execution of the modified program explicitly generates the

[1] In the current implementation, a code block is a sequence of instructions containing no branches or memory references. Better performance could be obtained by also allowing branches and memory references that do not generate events, but this has not been implemented yet.

events that are sent to the simulator. This is usually done by compiling modified assembler source code, resulting in a new executable file with the simulator linked in. (See Figure 3.) This has the disadvantage that only programs for which the source is available can be simulated. Memory reference events for library code cannot be generated unless the library source is also available. Furthermore, the application program usually must be in a specific source language. For example, RPPT requires the input source to be Concurrent C, Tango requires the input source to use the Argonne macro package [1], and PROTEUS requires the input source to be annotated with a set of language extensions specific to PROTEUS.

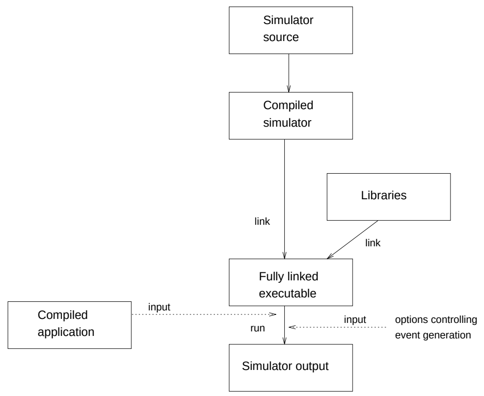

Figure 4: The procedure for simulating an application using MINT. The application need not be recompiled, so no source is needed.

The interpretation of standard object code under MINT is simpler. (See Figure 4.) It requires no preprocessing or recompilation. Almost anything that runs natively can be interpreted,[2] and a complete memory reference trace of the program execution, including library references, is easily generated. Interpretation is also language-independent, since only the executable object code is used.

In contrast to interpretation, native execution requires recompiling each application program every time the simulator changes. Furthermore, if several separately-compiled versions of the simulator are needed, then native execution requires multiple copies of each application to be maintained as well.

3.5 Dispatching

To dispatch the simulation of an instruction (or in most cases, a block of instructions), MINT calls a synthesized function through a function pointer. Even in the case of single instructions, this approach is used instead of a big switch statement because it is more flexible and, with the compiler we use, it is faster. The overhead of a 2-argument function call and return through a function pointer is 6 instructions, compared to 14 instructions for a switch statement. The extra instructions in the switch statement come from run-time checks to

[2]Some Unix system calls are not supported, but all the calls that are used in the parallel programs we have seen are supported.

ensure that the switch value is legal and converting it to a jump location. Function pointers are also more flexible since they can point to arbitrary locations in memory (allowing function creation at run-time). Unlike the cases of a switch statement, functions can be separately compiled. This allows a clean separation between the functions for instruction simulation and the main control loop.

3.6 Execution model

MINT is intended primarily for research into memory system designs. For the sake of efficiency it uses a simplified processor model. Each instruction of the target machine can be executed in a single simulated cycle. When a processor generates a memory reference the processor stalls until it is released by the target system simulator. Synchronization operations such as acquiring and releasing a lock cause special routines in the target system simulator to be called. This allows the target system to simulate fence operations needed by weak consistency models to force the flushing of write buffers.

4 Interface overview

This following two subsections give an overview of the interface that MINT provides for writing program-driven simulators and defines the terms used to describe this interface. The term "back end" is used throughout this document to refer to code called by MINT that performs some action associated with the memory trace. The back-end code is linked with MINT to produce a single executable file, but for the purposes of this manual it is useful to distinguish between the functions provided by MINT and the functions provided by the back end. For a more complete and detailed description of the interface, see section 6.

4.1 Events

The interface between MINT and the back end is modelled after Figure 2. MINT sends events of interest to the back end by calling a user-defined back-end function. For example, when a read event is generated, MINT calls the function `sim_read()` and when a write event is generated, MINT calls `sim_write()`. All back-end functions called by MINT have the prefix "`sim_`".

The return value from the call to the back-end function controls process execution. A return value of zero (`T_ADVANCE`) allows the process that generated the event to continue execution. A return value of one (`T_FREE`) simply frees the task associated with the event and does not cause the process to continue executing instructions. See section 6 for the complete list of possible return values.

Each event is represented by a structure that contains the process id of the process that generated the event, the time at which the event occurred, and the type of event. The event structure also includes some fields that are used only by certain events. For a complete description of the event structure, see section 6.4.

For read and write events, the `vaddr` field of the event structure contains the address being read or written. This is the address that the object program would have used if it were executing natively instead of being interpreted by MINT. Since MINT simulates multiple processes in a single address space, the real memory address that MINT uses is different than the address that the object program uses. The real memory address used by MINT is stored in the `paddr` field of the event structure.

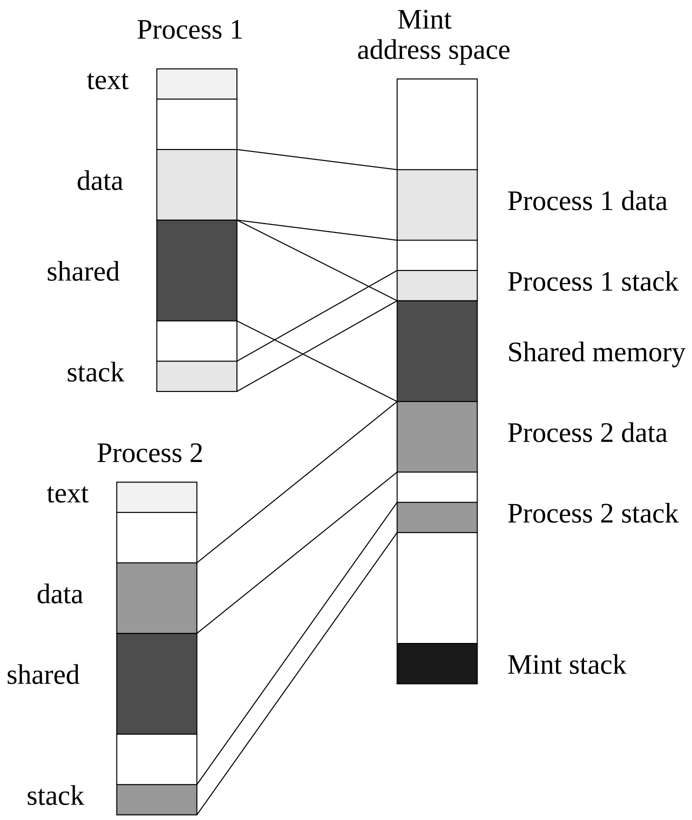

Figure 5: MINT runs in a single address space and simulates multiple processes by allocating data and stack space for each process within its own address space.

The `vaddr` and `paddr` fields are so-named because they are similar to the "virtual" and "physical" addresses used by programs in a multiprogramming operating system. Each simulated process may have its own notion of what is stored at (virtual) address 1000, for example, but they each use a different (physical) address in memory. MINT dynamically translates the "virtual" address used by a simulated process to a "physical" address in memory. (See Figure 5.) Both addresses are passed in the event structure. The `paddr` field is often more useful than the `vaddr` field since the `paddr` field provides a unique key that the back end can use to look up information about an address. A private address in processor 1 needs to be distinguished from a private address in processor 2, and the `paddr` field provides an easy way to distinguish private addresses.

4.2 Tasks

A realistic simulation of a parallel processor must allow multiple concurrent activities. A memory read by processor 1, for example, may execute simultaneously with a memory write by processor 2. Each of these operations may take a variable amount of time, and may require other operations to be performed, such as reading a cache block over the bus, writing a dirty block back to memory, or invalidating other processors' caches. (See Figure 6.)

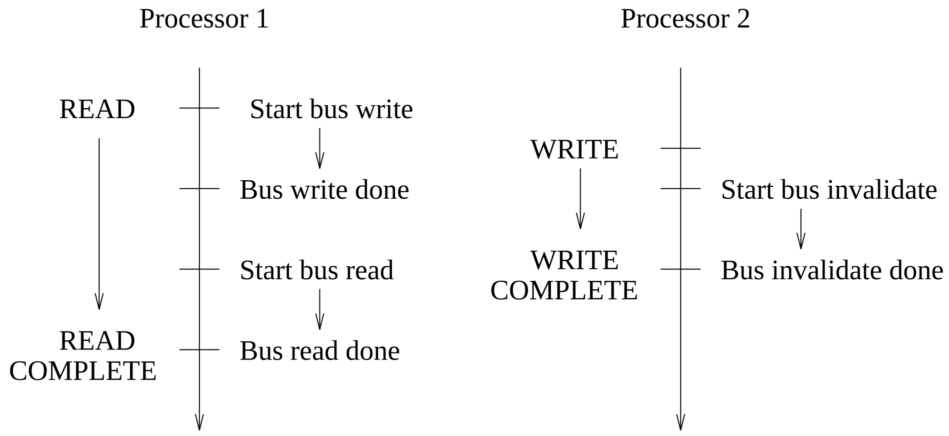

Figure 6: The read event by processor 1 may require several operations to complete before the process can continue. In this example, it must first write a dirty cache block and then read in another cache block. Meanwhile, processor 2 needs the bus to send an invalidation before the write event can complete. This affects the timing of the operations performed by processor 1.

To ensure that all operations are simulated in chronological order, MINT provides support for creating and scheduling arbitrary operations. Each independently schedulable operation is called a "task". Each task has associated with it a time value, a priority, and a function pointer. MINT "executes" each task at its specified time in priority order. Tasks are "executed" by calling the function through the function pointer. The return value from this function controls process execution as mentioned above. (Returning `T_ADVANCE` unblocks the associated process and allows it to continue executing instructions; returning `T_FREE` simply frees the task without affecting the state of the process.)

Every event generated by a process blocks that process and creates a new task. The function pointer for the task points to the back-end function corresponding to the type of event. The task also contains a pointer to the event structure and the process id. This association between a task and a process is used by MINT to determine which simulated process to continue when a task returns `T_ADVANCE`.

The simulator writer may schedule a new task by calling `sched_task()`. This function takes 3 arguments: a pointer to the current task, a function pointer, and a time value. `sched_task()` creates a new task with a default priority of zero and enqueues it on the task queue. `sched_task()` returns a pointer to the new task. The task pointer is used to copy the event pointer and process id into the new task. This maintains the association between a task and a process. If this information is not needed (because the new task will not return `T_ADVANCE` and will not schedule a task that will need this information) then the back end can call `newtask()` to create an uninitialized task, and schedule it by calling `schedule_task()`. (See section 6.2 for more information.)

The task structure contains several fields that are not used by MINT and may be used by the back end. This is useful for remembering "state" between task executions. For a complete description of the fields in a task, see section 6.6.

5 Examples

This section contains simple program examples that demonstrate how to use MINT. For an example of a back end that simulates a bus-based multiprocessor with snooping caches, see section 10.

5.1 A simple example

This example program simply counts the number of reads and writes for private and shared memory references. The code for this program is in the file "example1.c" in the examples directory.

The program needs to include the header file "`mint.h`" in order to get structure definitions and function prototypes.

```
#include "mint.h"
```

Since we want to count both private and shared memory references for each read and write, we define a structure so that we do not have to keep inventing different variable names for the same concept.

```
/* define a type for memory reference stats: shared and private */
typedef struct memref_t {
    int prv;
    int shr;
} memref_t;
```

We also define a structure to store the global counts. This groups all the counters in one place so it is easy to see what we are counting and makes it easy to add new fields later.

```
struct global_stats {
    memref_t nreads;
    memref_t nwrites;
} Data;
```

Next, we need to define the function `sim_read()` which is called on every read, and `sim_write()` which is called on every write. The type field is extracted from the event structure and the `E_SHARED` bit is checked to determine if this is a shared reference.

```
sim_read(task_ptr ptask)
{
    int type;

    type = ptask->pevent->type;

    /* count shared and private reads */
    if (type & E_SHARED)
        Data.nreads.shr++;
    else
        Data.nreads.prv++;
    return T_ADVANCE;
}
```

The `sim_write()` function is similar to `sim_read()`:

```
sim_write(task_ptr ptask)
{
    int type;

    type = ptask->pevent->type;

    /* count shared and private writes */
    if (type & E_SHARED)
        Data.nwrites.shr++;
    else
        Data.nwrites.prv++;
    return T_ADVANCE;
}
```

To print the results at the end of the simulation, we need to define the function `sim_done()`. The arguments passed in are the simulated elapsed time and the total simulated cpu time used by all processes.

```
void
sim_done(double elapsed, double cpu_time)
{
    printf("Private: %d reads, %d writes\n",
           Data.nreads.prv, Data.nwrites.prv);
    printf("Shared:  %d reads, %d writes\n",
           Data.nreads.shr, Data.nwrites.shr);
}
```

That's the entire program. The makefile to compile and link this program is shown below.

```
MINTDIR = ..
INC     = -I$(MINTDIR)/include
MINT_O  = $(MINTDIR)/mint.o

CFLAGS  = -O2 $(INC)
LIBS    = -L$(MINTDIR) -lsim

.c.o: ; cc -c $(CFLAGS) $*.c

example1: example1.o $(MINT_O)
        cc -o example1 example1.o $(MINT_O) $(LIBS)
```

This will create an executable called "example1". To run this on an application program called "foo" simply type:

```
example1 foo
```

This will count all the shared and private reads and writes made by "foo".

5.2 Another simple example

This example demonstrates the use of `sched_task()`. On every memory read or write, it schedules a task to complete 10 cycles in the future. The entire program is given below. The source for this program is in the file "example2.c" in the "examples" directory.

```
#include "mint.h"

int ref_done(task_ptr ptask);

sim_read(task_ptr ptask)
{
    /* schedule the ref_done() function 10 cycles from now */
    sched_task(ptask, ref_done, ptask->time + 10);
    return T_FREE;
}

sim_write(task_ptr ptask)
{
    /* schedule the ref_done() function 10 cycles from now */
    sched_task(ptask, ref_done, ptask->time + 10);
    return T_FREE;
}

/* This function is called when a read or write completes. */
ref_done(task_ptr ptask)
{
    return T_ADVANCE;
}
```

6 MINT interface

6.1 Functions called by MINT

These functions are called by MINT and may be defined by the back end. All of these functions are optional. If one is not supplied, an empty stub function will be linked in from the MINT library.

`int sim_init(int argc, char **argv)`

MINT calls `sim_init()` once at the beginning of execution, before starting to interpret the object program. This function is typically used to initialize data used by the back end. MINT passes its own `argc` and `argv` to `sim_init()` so that the entire command line is available. This includes the arguments to MINT, which appear first, followed by the double hyphen separator ("`--`"), followed by the arguments to the back end.

```
mint [mint options] [-- back-end options] objfile [objfile options]
```

MINT reads its command line options using the getopt() library function. (See section 7 for a description of the command line options for MINT.) Since getopt() skips over the double hyphen separator ("`--`") the next call to getopt() will return the first back-end option, if any. This makes it convenient for the back end, since the back-end options can be parsed using getopt() as if they were the only options on the command line. If the back end is composed of separate independent modules each of which has its own set of options, the convention of using (“`--`”) to separate one module's options from the next makes it easy to parse each module's options independently.

Since MINT has no way of knowing which of the command line arguments is the name of the object file to execute, the return value from from `sim_init()` must be the index in the `argv` array of the name of the object file. This is simply the value of the external variable `optind` defined by getopt() after it finishes parsing the back-end options.

So a typical `sim_init()` looks like:

```
sim_init(int argc, char **argv)
{
    extern int optind;
    int c;

    while ((c = getopt(argc, argv, "a:b:o:")) != -1)
        switch (c) {
        /*
         *  ... do something with the options ...
         *
         */
        }
    return optind;
}
```

`void sim_done(double elapsed, double cpu_time)`

MINT calls `sim_done()` once at the end of execution, after the object program terminates. This function is typically used to print statistics collected during the run. The first argument is the simulated execution time of the object file in processor cycles, and the second argument is the total accumulated cpu time of all the simulated processes.

`void sim_usage(int argc, char **argv)`

MINT calls `sim_usage()` if there was an error in parsing the command line arguments. MINT first prints its own usage message and then calls `sim_usage()`.

`int sim_read(task_ptr ptask)`

`int sim_write(task_ptr ptask)`

`int sim_user(task_ptr ptask)`

`int sim_lock_attempt(task_ptr ptask)`

```
int sim_lock_acquire(task_ptr ptask)

int sim_lock_release(task_ptr ptask)

int sim_barrier_attempt(task_ptr ptask)

int sim_barrier_acquire(task_ptr ptask)

int sim_psema_attempt(task_ptr ptask)

int sim_psema_acquire(task_ptr ptask)

int sim_vsema(task_ptr ptask)

int sim_sync(task_ptr ptask)

int sim_instr(task_ptr ptask)

int sim_exit(task_ptr ptask)

int sim_function(task_ptr ptask)

int sim_terminate(task_ptr ptask)

int sim_kill(task_ptr ptask)

int sim_alloc(task_ptr ptask)

int sim_shalloc(task_ptr ptask)

int sim_lockalloc(task_ptr ptask)
```

MINT calls `sim_read()` on every read event, passing it a pointer to a task. The task structure is described in section 6.6. Similarly, `sim_write()` is called on every write event, and `sim_user()` is called on every user-defined event. User-defined events are caused by calls to `generate_event()` from within the object program. (See section 6.2.)

MINT calls `sim_lock_attempt()` when a process attempts to acquire a spinlock. This occurs when the object program calls the library routine `ussetlock()`. MINT calls `sim_lock_acquire()` when a process acquires a spinlock, and `sim_lock_release()` when a process releases a spinlock. Locks are released when the object program calls the library routine `usunsetlock()`. Spinlocks are costly to simulate because they can cause an event every cycle. To reduce the overhead of simulating spinlocks, MINT blocks a process if it fails to acquire a spinlock on the first attempt and unblocks it only when the lock becomes free. This results in only one call to `sim_lock_attempt()` instead of one every cycle, and coincides with the behavior that would occur if queue-based locks were used. This optimization can be turned off with the `[-S]` command line option. (See section 7.)

MINT calls `sim_barrier_attempt()` when a process reaches a barrier. MINT calls `sim_barrier_acquire()` when a process continues past the barrier. Barrier events are caused by the object program calling the library routine `barrier()`.

MINT calls `sim_psema_attempt()` when a process attempts to perform a P() operation on a semaphore, and `sim_psema_acquire()` when the P() operation succeeds. MINT calls `sim_vsema()` when a process performs a V() operation on a semaphore. If several processes are waiting on a semaphore when a V() operation is performed, then exactly one waiting process is unblocked. Semaphore events are caused by the object program calling `uspsema()` or `usvsema()`. Calls to `uspsema()` or `barrier()` from within a process may cause that process to block.

To force synchronization variables to pay for the cost of reading and writing memory, the library versions of the synchronization routines may call either `sim_read()` or `sim_write()`. For example, the code for the library function `sim_lock_acquire()` is:

```
sim_lock_acquire(task_ptr ptask)
{
    return sim_write(ptask);
}
```

`sim_lock_attempt()` is the only library function that calls `sim_read()`. The library functions that call `sim_write()` are the following: `sim_barrier_attempt()`, `sim_lock_acquire()`, `sim_lock_release()`, `sim_psema_attempt()`, and `sim_vsema()`. If a behavior different from the default is desired, the user should define these functions in the back end.

With the exception of three of the above functions, MINT calls these functions *before* the operation is performed. This allows the back end to block the process that generated the event and exercise control over access to memory and synchronization variables. The exceptions are: `sim_lock_acquire()`, `sim_barrier_acquire()`, and `sim_psema_acquire()`. The three exceptions are called *after* the process has successfully acquired the lock.

MINT calls `sim_sync()` if the R4000 "sync" instruction is encountered.

MINT calls `sim_instr()` if instruction tracing is requested (using the command line option "`-t i`") and the instruction does not cause some other type of event.

MINT calls `sim_exit()` when a process exits (more precisely, when the process calls `_exit()`). This function is called before `sim_terminate()` and before waiting for all unexecuted tasks associated with this process (if any) to execute. For example, the back end may have pending writes in progress if write buffers are simulated. The `sim_exit()` function allows the back end to collect statistics and cleanup before `sim_terminate()` is called.

MINT calls `sim_terminate()` as the last back-end function call before a process is terminated. If there are still tasks scheduled for this process, a warning message is printed.

MINT calls `sim_function()` when a process makes a function call that MINT intercepts, such as `malloc()`, or `close()`. This hook exists so that programs can get an exact accounting of time spent in a simulation. `sim_function()` is not called for functions such as `fork()` that have their own dedicated event functions.

MINT calls `sim_kill()` when a process receives a signal that kills it. This allows the back end to clean up the state of the process before the process is killed. The back end should ensure that no other tasks associated with this process will return `T_ADVANCE`. This can be accomplished by calling `task_destroy()`. The default library routine for `sim_kill()` is:

```
sim_kill(task_ptr ptask)
{
    task_destroy(ptask->pid);
    return T_ADVANCE;
}
```

MINT calls `sim_alloc()` when a process allocates private memory (through malloc(), calloc(), or realloc()). MINT calls `sim_shalloc()` when a process allocates shared memory (through shmalloc(), usmalloc(), uscalloc(), or share_malloc()). The event structure includes fields that are useful for constructing a memory allocation map:

arg1 This is the size of the memory requested by the application.

rval This is the base of the memory region returned by the call to malloc(), calloc(), etc.

raddr This is the "return address", that is, the instruction address in the application where malloc() was called.

The return value from each of the above routines should be one of the following:

`T_ADVANCE` The processor associated with this task may continue executing instructions. A common mistake is to return `T_ADVANCE` from a task when the processor is not blocked. The debugging version of MINT detects this type of error. (Use "make debug" or "make versions" when building MINT to get the debugging version.)

`T_FREE` This task is put on the free list and the next task with the minimum time is removed from the task queue and executed. If the processor associated with this task is blocked, then it remains blocked.[3]

`T_YIELD` This is the same as `T_FREE` except that the task is not put on the free list. Only a reschedule is performed. This is normally used only by routines internal to MINT.

`T_CONTINUE` This return value specifies that the next task on the task stack should be scheduled now. If the task stack is empty, then this is equivalent to (but slightly less efficient than) using `T_ADVANCE`. Tasks are put on the task stack by calling `stack_task()` or `pri_stack_task()`. A task stack is associated with a task, not with a process, so multiple independent stacks can be active for the same process simultaneously. The intended use of this is to support modularization.

[3] `T_FREE` used to be called `T_NO_ADVANCE` but the name was changed because it seemed to imply that the process was not allowed to continue.

For example, a bus or network module can be written with a single entry point and such that it always returns `T_CONTINUE`. The module can then be called from several different points in the code and after the module completes the simulation will continue at the function pushed on the stack before the call to that module.

`void sim_fork(task_ptr ptask)`

`void sim_block(task_ptr ptask)`

`void sim_unblock(task_ptr ptask)`

`void sim_checkpoint(task_ptr ptask)`

`void sim_message(mint_time_t curtime, char *arg)`

MINT calls `sim_fork()` when a parent process forks a child. The `pevent` pointer in the task structure points to the event for the parent process. The pointer to the child's event is stored in the `pevent` field of the parent's event structure.

MINT calls `sim_block()` when a process blocks, and `sim_unblock()` when a process becomes unblocked. A process may block for any of the following reasons:

1. the process exits before all of its children have finished,
2. the process calls `wait()` and there are child processes but none of those that have not already been waited on have finished,
3. the process performs a P() operation on a semaphore and the semaphore is not available,
4. the process reaches a barrier and it is not the last process to reach the barrier,
5. the process attempts to acquire a lock that is held by some other process (unless spinning is enabled with the `[-S]` option).

The task pointer passed to `sim_unblock()` is for the *unblocked* process. The event pointer for the *unblocking* process can be retrieved through `ptask->pevent->pevent`. The number of cycles that the unblocked process remained blocked is stored in the `duration` field of the event structure for the unblocked process (`ptask->pevent->duration`). A separate call to `sim_unblock()` is made for each unblocked process.

MINT schedules a "checkpoint" task to execute at fixed intervals. This task does not really do any "checkpointing" in the sense of saving the simulation state to be restarted in case of a system crash, but it does provide a convenient hook for the back end to perform some operation at regular intervals. The default checkpoint interval is ten million cycles. This can be changed to any value by using the command line option "`-C interval`", where `interval` is the number of cycles between checkpoints. Setting the interval to zero turns off checkpointing. Each time the checkpoint task runs, it calls the function `sim_checkpoint()`. If the back end does not define this function, the library version is linked in which simply prints a message containing the time. If the MINT source was compiled without `NO_SOCKETS` defined, then at every checkpoint the socket connection is polled for pending connection requests and commands. The socket commands allow a client program to obtain information about a running simulation. Extensions can be added in the future to allow remote debugging as well. When using sockets a small checkpoint interval should be used to improve the response time for client commands. A value of 20000 is reasonable. The checkpoint interval can be changed dynamically during the simulation by assigning to the global variable `Ckpoint_freq`. (See section 6.3.) There is also a socket command, "check", for reading and setting the checkpoint interval.

MINT calls `sim_message()` when it receives a socket command that begins with "user". The current time is passed as the first argument to `sim_message()`, and the second argument is a character string consisting of the arguments to the "user" command. The routine `menu_parse()` is defined by MINT to make it easy for the back-end to define and parse a hierarchical menu of commands. (See section 6.2.)

6.2 Functions defined by MINT

These functions are supplied by MINT and callable from the back end.

`task_ptr sched_task(task_ptr ptask, PFTASK ufunc, mint_time_t time)`

This function creates a new task and schedules it at the given `time` with a priority of zero. A pointer to the current task is passed in as the first argument. This is required because every task is associated with some process and event. This association is passed on to the new task. When the new task is executed, the user-defined function `ufunc` is called with a pointer to the new task structure. `sched_task()` returns a pointer to the new task. This pointer may be used to fill in the user-defined fields.

`task_ptr pri_sched_task(task_ptr ptask, PFTASK ufunc, mint_time_t time, int priority)`

This function is similar to `sched_task()` with the addition of a priority level argument. The priority is used to order tasks that are scheduled to run at the same time. Higher numbered priorities execute first. Negative values may also be given. Calls to `sched_task()` may be mixed with calls to `pri_sched_task()`. Tasks scheduled with `sched_task()` have a priority of zero. A pointer to the new task is returned.

`task_ptr stack_task(task_ptr ptask, PFTASK ufunc)`

In complex simulations it is sometimes convenient to be able to specify that a task should execute when a sequence of activities completes. The completion time and the number of intermediate tasks that need to be scheduled may not be known. The `stack_task()` function provides a way to specify that a task should be scheduled whenever another task completes. This function creates a new task to execute the function pointed to by `ufunc`. The new task is not scheduled but is pushed on the task stack for `ptask`. Whenever a task returns `T_CONTINUE`, the stack for that task is "popped" and the task that was at the top of the stack is scheduled. The stack for a task is passed along to newly created tasks by calls to `sched_task()` or `pri_sched_task()`.

`task_ptr pri_stack_task(task_ptr ptask, PFTASK ufunc, int priority)`

This function is similar to `stack_task()` with the addition of the priority argument. A new task is created and pushed onto the task stack of `ptask`. When the new task is popped from the stack (because some task returned `T_CONTINUE`), the new task is scheduled at that time with the specified priority.

`task_ptr newtask()`

This creates a new task without initializing any fields. In particular, the `pevent`, `pid`, and `priority` fields may have garbage values and should be initialized by the back end, if needed. The new task can be scheduled with a call to `schedule_task`. Before scheduling this task, the `priority` field should be initialized. This task is not associated with any process id so it should not return `T_ADVANCE` without first setting the `pid` field. A pointer to the new task is returned.

`void free_task(task_ptr ptask)`

This frees the task pointed to by `ptask`. This function does not remove the task from the queue of scheduled tasks. To do that, use `cancel_task()`.

`task_ptr schedule_task(task_ptr pnew)`

This schedules the task pointed to by `pnew`.

`void cancel_task(task_ptr ptask)`

This removes the task pointed to by `ptask` from the task queue and frees it. If this routine is called with a pointer to a task that has not been scheduled, bad things will happen.

`task_ptr unblock_proc(int pid, mint_time_t time)`

A new task is created and scheduled at the given time. This task is associated with processor `pid` and, when executed, will unblock that processor. That processor will then execute instructions until it generates an event.

`void generate_event(long vaddr, long utype, void *sptr)`

This function is callable from the object program (not the back end). Calls to `generate_event()` within the object program cause an event to be generated for the currently running process. The back-end function `sim_user()` is called, and the arguments to `generate_event()` can be retrieved from the event structure.

The `vaddr` argument is expected to be the address of a variable in the object program and is passed in the `vaddr` field of the event structure. The `paddr` field of the event structure is computed from `vaddr`, and is the address in memory corresponding to `vaddr`. This makes it possible for the back end to read and write the object program's data. If not needed by the back end, the `vaddr` argument need not be a legitimate address, so a zero value may be passed in.

The `utype` argument is copied untranslated to the `utype` field of the event structure. The `sptr` argument is translated (just like `paddr`) and stored in the `sptr` field.

`generate_event()` can be used for annotating lock accesses, for example, or for simulating prefetching. The back-end writer should be aware, however, that inserting calls to `generate_event()` in the object program may increase the number of memory reference instructions.

The object program does not call the MINT-supplied `generate_event()` function directly since the object program is compiled separately from MINT. Instead, MINT replaces the function call at interpretation time with a call to its own internal function. The compilation and linking of the object program, however, requires a function definition for `generate_event()`. This can simply be an empty function by that name. An empty stub function for `generate_event()` is included in the MINT library for this purpose.

`void mem_ref(long *value, long paddr, int type)`

This function handles all the possible read and write operations. `value` is a pointer to the value to read or write, `paddr` is the memory address to read or write, and `type` is the type of event.

`int addr2src(long iaddr, char **fname, int *line, char *buf, int buflen)`

This function maps an instruction address in the application program to the filename and line number in the application source. The arguments are:

`iaddr` The instruction address.

`fname` If not NULL, the address of a character pointer that will be set to point to the file name containing the instruction address.

`line` If not NULL, the address of an integer variable that will be set to the source line number for the given instruction address.

`buf` If not NULL, a character array that will contain the source line from the application file.

`buflen` The maximum length of `buf`.

If successful at finding a mapping, `addr2src()` returns 1 and sets the non-null parameters `fname`, `linenum`, and `buf`. If not successful `addr2src()` returns 0 and does not modify any of the parameters. If a mapping (file:line) is found, but the source file does not exist then 1 is returned but "buf" is not modified.

`int readsrc(char *fname, int line, int numlines, char *buf, int buflen)`

This function reads up to `numlines` lines starting at line `line` from the source file `fname` into the character array `buf`. If the file is less than `line` lines long, then `buf` is unmodified. If modified, `buf` is terminated with a NULL (zero) byte. The number of bytes read into `buf` is returned.

`void transmit(char *buf)`

This routine sends a message through a socket to a client program. The string of characters up to (but not including) the first NULL byte in the array pointed to by `buf` are sent in the message.

`void transmit_src(long addr, int numlines)`

This routine maps the instruction address given by `addr` to the filename and line number in the application source and reads up to `numlines` lines from the file and transmits the lines through the socket connection to a client program. The first line transmitted precedes the source line containing `addr` by `numlines / 2` lines so that the context of the source line is visible.

`void menu_parse(command_ptr cmdptr, mint_time_t curtime, char *arg)`

This routine is provided to make it convenient to parse a hierarchical menu of user-defined commands. A *command* is simply a string of non-whitespace characters. Arguments to a command are separated by whitespace. The command and arguments are sent as a single string of characters by a client program and passed to the back-end through the `sim_message()` interface. The back-end defines a set of *menu commands* that it understands and calls `menu_parse()` to parse the character string given to `sim_message()`. A *menu command* is defined by the following C structure:

```
typedef struct command {
    char *name;
    int siglen;
    struct command *subcomm;
    void (*func)(mint_time_t, char *);
} command_t, *command_ptr;
```

The `name` is the name of the command; `siglen` is the number of significant characters in the command to use for string comparison; `subcomm` is a pointer to a submenu of commands, or NULL is there is no submenu; `func` is a pointer to the function to execute when a match is found. The function pointer is not used if the `subcomm` pointer is not NULL. The first argument to `menu_parse()` is a pointer to the top-level command menu. A *menu* is simply an array of commands, terminated by a command with NULL entries.

For example, suppose the back-end defines commands to return the location in the application program where the most read and write latency occur, and also a "help" command to return a list of supported commands. The menus could be defined as follows:

```
void com_whread(mint_time_t, char *);
void com_whwrite(mint_time_t, char *);
void com_help(mint_time_t, char *);

static command_t sub_wh[] = {
    "read",  2, NULL, com_whread,
    "write", 2, NULL, com_whwrite,
    NULL, 0, NULL, NULL
};
```

```
static command_t Commands[] = {
    "wh",   2, sub_wh, NULL,
    "help", 2, NULL, com_help,
    NULL, 0, NULL, NULL
};
```

The functions `com_help()`, `com_whread()`, and `com_whwrite()` also need to be defined. The top-level menu is the array `Commands`, and is passed to `menu_parse()`. A client program can then send the string "user help" through the socket interface to MINT and get back a list of the commands that the back-end supports. The command "user wh read" would invoke the `com_whread()` function, and the command "user wh write" would invoke the `com_whwrite()` function.

The `sim_message()` routine can be defined simply as:

```
void
sim_message(mint_time_t curtime, char *arg)
{
    menu_parse(Commands, curtime, arg);
}
```

`void menu_help(command_ptr cmdptr)`

This routine traverses the menu hierarchy and sends the list of commands to the client program through the socket interface. The `com_help()` function above can be defined simply as:

```
void
com_help(mint_time_t curtime, char *arg)
{
    menu_help(Commands);
}
```

6.3 Global variables

MINT provides the following global variables:

`int Nprocs`	The number of active processes.
`int Maxpid`	The maximum process id ever used. The first process has a process id of zero. The process id of a terminated process may be re-used.
`FILE *Mint_output`	MINT prints some statistics at the end of the simulation such as the total elapsed simulated time (in processor cycles) and the

total accumulated cpu time. This variable controls where the output is sent to. By default, this output is sent to `stderr`, but the back end may change this to any file descriptor. If set to NULL, then no output is generated.

`int Verify_protocol` This variable is set to true (equal to 1) if protocol verification is selected with the command line option "`-V`". If set, then the back end must supply a value on a read by using the "`value`" pointer in the event structure to write the value that should be used. On a write, the "`value`" pointer points to the value being written by the object program.

`unsigned int Ckpoint_freq` The interval between calls to `sim_checkpoint()`. `Ckpoint_freq` is used to schedule a "checkpoint" task. (This task does not really do any "checkpointing" – it simply calls `sim_checkpoint()`.) If set to zero, then no checkpoint tasks will be scheduled. See also the command line option "`-C interval`" in section 7.

`mint_time First_fork_elapsed` The time at which the first fork occurred, or zero if no forks have occurred.

`mint_time First_fork_cpu` The accumulated cpu time of processor 0 at the time of the first fork.

`int Release_consist` Set to one if the `-r` option was specified, otherwise zero. (See section 7.)

6.4 Events

Events are generated by MINT as a result of simulating instructions in the object program. Each load instruction, for example, may generate a read event, and each store instruction may generate a write event. An event is usually generated *before* the corresponding operation is initiated. A read event, for example, is generated before the load instruction is performed. This is to ensure that events are executed in chronological order and to allow the back end to return values for read events.

Some operations generate an event *after* performing the operation. These operations include forking a new process, acquiring a lock, acquiring a barrier, acquiring a semaphore, unblocking a process, and terminating. These operations require that the operation be performed before the event is generated. The `E_FORK` event, for example, passes in a pointer to the child event structure, so the fork() must occur before the `E_FORK` event. For reasons of atomicity and correct ordering of events, the `E_LOCK_ACQUIRE` event (and the corresponding events for acquiring barriers and semaphores) must not be sent before the lock is acquired, since otherwise some other process may acquire the lock instead.

The event structure is defined in `"event.h"`:

```
typedef struct event {
    mint_time_t time;
    mint_time_ptr cpu_time;
    mint_time_t duration;
    struct event *next;
    int pid;
    long type;
    long utype;
    long iaddr;
    long vaddr;
    long paddr;
    long *value;
    int nspins;
    struct event *pevent;
    void *sptr;
} event_t, *event_ptr;
```

The meaning of each field is given below.

`time`
: The time (in processor cycles) at which this event was issued. To obtain the current time, the back end should use the time field in the task structure.

`cpu_time`
: A pointer to the accumulated cpu time (in processor cycles) for this process. The cpu time for this process does not include the cpu time of its children. By default, every instruction and every operation takes 1 cycle. The defaults can by changed by creating a file of cycle times (see the "`-c`" option in section 7). The back end may use this pointer to change the cpu time of a process, but in most cases this is not necessary.

`duration`
: If this is an unblock event, then `duration` contains the time that this process spent waiting.

`next`
: A pointer for linking together a list of events.

`pid`
: The process id of the process causing the event. The first process has a pid of zero.

`type`
: The type of event. (See `"event.h"` for macro definitions.)

`utype`
: A user-defined type. If this is an `E_USER` event, then this event was caused by a call to `generate_event()` from within the object program and this field is filled in with the "value" argument passed in the `generate_event()` function. If this is a block event, then `utype` contains the type of event that caused this process to block. This can be `E_WAIT`, `E_LOCK_ATTEMPT`, `E_PSEMA_ATTEMPT`, or `E_BARRIER_ATTEMPT`.

iaddr The address of the instruction causing the event.

vaddr The effective virtual address. This is the virtual address the process would access if it were running on the native machine instead of being interpreted. For instruction events that do not reference memory, this field contains the instruction word.

paddr The effective physical address. This is the real address in memory that the interpreted program accesses. This field and the `value` field are needed to implement protocol verification. This field also provides a unique address for references to private memory, whereas the `vaddr` does not.

value A pointer to the value to read or write at paddr. If protocol verification is being used, this value is filled in by the back end on a read event, and this value is used by the back end for writing on a write event.

nspins If this is an `E_LOCK_ACQUIRE` event, then this field contains the number of spin attempts before acquiring the lock. (This is not implemented yet.)

pevent A pointer to the event structure for another process. On a fork, this field is set to point to the event structure for the child. On an `E_UNBLOCK` event, this points to the event structure for the process that did the unblocking. A process may unblock another for the following reasons: (1) when a process terminates it will unblock its parent, if necessary (so that it can wait() on its child), (2) when a process executes a V() operation on a semaphore, it will unblock another process that previously executed a P() operation for that semaphore, (3) when a process releases a lock, it will unblock a process if there is one waiting for the lock, and (4) when a process reaches a barrier and it is the last process to arrive, it unblocks all the other processes.

sptr A pointer field that is used by the task scheduler. This field must not be modified.

fname If the event type is `E_FUNCTION` then this field contains a pointer to the function name that was intercepted by MINT.

6.5 Types of events

The types of events supported by MINT are listed in `"event.h"`. This header file includes macro definitions for each of the possible types. The bit fields are shown in Figure 7.

In order to support detailed simulations, many different types are defined. The macro definitions defining the bit fields are:

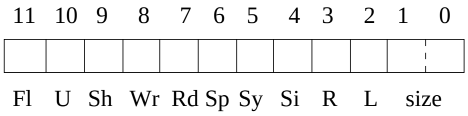

Figure 7: The bit fields that encode the type of event. (See the text for an explanation of the bit fields.)

`E_SIZE` If `E_READ` or `E_WRITE` is set, then this field indicates the size of the load or store. 0 = byte, 1 = halfword, 2 = word, 3 = double word.

`E_LEFT` If `E_READ` is set, then this is a "lwl" (load word left) instruction. If `E_WRITE` is set, then this is a "swl" (store word left) instruction.

`E_RIGHT` If `E_READ` is set, then this is a "lwr" (load word right) instruction. If `E_WRITE` is set, then this is a "swr" (store word right) instruction.

`E_SIGNED` If `E_READ` is set, then this is a signed read (either byte or halfword, depending on the value of the `E_SIZE` field). There are no signed write instructions.

`E_SYNC` If set, then this is a "ll" (load linked), "sc" (store conditional), or "sync" instruction. These are R4000 instructions. If `E_READ` is set, then this is a "ll" instruction. If `E_WRITE` is set, then this is a "sc" instruction.

`E_SPECIAL` If set, then the lower order bits are used to encode special events. These events are listed below in the description of special events.

`E_READ` This is a read operation.

`E_WRITE` This is a write operation.

`E_SHARED` This is a shared read or shared write operation.

`E_USER` This is a user-defined event. This bit is set for events that are caused by calls to `generate_event()` from within the object program.

`E_FLOAT` If set, then this is a floating point load or store.

Each event type is the bitwise OR of some of the above bit fields. Each of the possible event types has a corresponding macro definition given below. For most purposes, the back end need not examine the event type since each different event causes a separate function to be called. For example, all the load instructions (lw, lh, lb, etc.) generate a read event which causes the `sim_read()` function to be called.

The first group of event types includes all reads or writes to a private memory address. (Every memory location is either shared among all processes or private to one process.)

E_READB	Read byte unsigned.
E_READH	Read halfword unsigned.
E_READW	Read word.
E_READBS	Read byte signed.
E_READHS	Read halfword signed.
E_WRITEB	Write byte.
E_WRITEH	Write halfword.
E_WRITEW	Write word.
E_FREADW	Read single-precision float.
E_FREADD	Read double-precision float.
E_FWRITEW	Write single-precision float.
E_FWRITED	Write double-precision float.
E_LWL	Load word left.
E_LWR	Load word right.
E_SWL	Store word left.
E_SWR	Store word right.

The second group of event types is the same as the first except that all operations occur on shared memory addresses.

E_SHR_READB	Shared read byte unsigned.
E_SHR_READH	Shared read halfword unsigned.
E_SHR_READW	Shared read word.
E_SHR_READBS	Shared read byte signed.
E_SHR_READHS	Shared read halfword signed.
E_SHR_WRITEB	Shared write byte.
E_SHR_WRITEH	Shared write halfword.
E_SHR_WRITEW	Shared write word.
E_SHR_FREADW	Shared read single-precision float.

E_SHR_FREADD	Shared read double-precision float.
E_SHR_FWRITEW	Shared write single-precision float.
E_SHR_FWRITED	Shared write double-precision float.
E_SHR_LWL	Shared load word left.
E_SHR_LWR	Shared load word right.
E_SHR_SWL	Shared store word left.
E_SHR_SWR	Shared store word right.

Instructions that do not reference memory have the following type:

E_INSTR	Instruction type.

Special events indicate a change in process status or number of processes:

E_FORK	This process forked. The `pevent` field contains a pointer to the child process.
E_MFORK	This process performed an m_fork(). (This is not supported yet.)
E_DONE	This process has terminated. If the parent process is waiting, it will first generate an `E_UNBLOCK` event.
E_BLOCK	This process blocked. The `utype` field in the event structure contains the reason for blocking and has one of the following values: `E_WAIT`, `E_LOCK_ATTEMPT`, `E_BARRIER_ATTEMPT`, or `E_PSEMA_ATTEMPT`.
E_UNBLOCK	This process is about to unblock another process. The `pevent` field contains a pointer to the process that will become unblocked. The `duration` field of the event structure contains the length of time the process was blocked. The `utype` field contains the type of event that caused the process to block.
E_LOCK_ATTEMPT	This process is about to attempt a lock operation. The `paddr` field contains the address of the lock.
E_LOCK_ACQUIRE	This process has successfully acquired a lock. The `paddr` field contains the address of the lock.
E_LOCK_RELEASE	This process is about to release a lock. The `paddr` field contains the address of the lock.
E_BARRIER_ATTEMPT	This process is about to attempt a barrier operation. The `paddr` field contains the address of the barrier.

`E_BARRIER_ACQUIRE`	This process has successfully acquired a barrier, which means that it was the last process to reach the barrier. The `paddr` field contains the address of the barrier.
`E_PSEMA_ATTEMPT`	This process is about to perform a P() operation on a semaphore. This may cause the process to block if the semaphore is not available. If the process blocks, it will generate an `E_BLOCK` event. The `paddr` field contains the address of the semaphore.
`E_PSEMA_ACQUIRE`	This process has successfully acquired a semaphore without blocking.
`E_VSEMA`	This process is about to perform a V() operation on a semaphore. This may unblock another process if one is waiting. If this happens, an `E_UNBLOCK` event will be generated. The `paddr` field contains the address of the semaphore.
`E_WAIT`	This process called `wait()`.
`E_EXIT`	This process called `_exit()`.
`E_FUNCTION`	This process called a function that MINT intercepts, such as `malloc()` or `close()`. The function name is passed in the `fname` field of the event structure.

6.6 Tasks

Tasks are independently schedulable operations. Tasks are used by the back-end writer to schedule operations to be executed at some time in the future. Every task is associated with some event. If a task function returns `T_ADVANCE`, it causes the process blocked on the associated event to continue.

The back-end writer may schedule a new task by calling `sched_task()`. This creates a new task and enqueues it on the task queue. The `time` field is used to schedule tasks in chronological order. If two tasks, A and B, have identical `time` fields, then the one with higher priority is scheduled first. If two tasks are scheduled for the same time and have the same priority then the task that was scheduled first is executed first.

The task structure is defined in "`task.h`":

```
typedef struct task {
    struct task *next;
    struct task *prev;
    mint_time_t time;
    int priority;
    int pid;
    int (*ufunc)(struct task *);
    struct task *tstack;
```

```
    struct event *pevent;
    int ival1;
    int ival2;
    int ival3;
    int ival4;
    int ival5;
    double dval1;
    double dval2;
    mint_time_t tval1;
    mint_time_t tval2;
    void *uptr;
    void *uptr2;
    int (*ufunc2)(struct task *);
    struct task *unext;
    struct task *uprev;
} task_t, *task_ptr;
```

`next` A forward link for task queues.

`prev` A backward link for task queues.

`time` The time at which this task is scheduled.

`priority` The priority level. Among tasks scheduled at the same time, higher numbered priorities are executed first. The default priority is zero. Negative numbers may be used to schedule low priority tasks.

`pid` The process id.

`ufunc` The function to call when this task is scheduled.

`tstack` The task stack. This is initially empty. Tasks may be pushed onto the stack by calling `stack_task()` or `pri_stack_task()`. The task at the top of the stack is scheduled when the current task returns `T_CONTINUE`.

`pevent` A pointer to the event associated with this task.

`ival1` An arbitrary integer value. Not used by MINT.

`ival2` An arbitrary integer value. Not used by MINT.

`ival3` An arbitrary integer value. Not used by MINT.

`ival4` An arbitrary integer value. Not used by MINT.

`ival5` An arbitrary integer value. Not used by MINT.

`dval1` An arbitrary double value. Not used by MINT.

`dval2` An arbitrary double value. Not used by MINT.

tval1 An arbitrary time value. Not used by MINT.

tval2 An arbitrary time value. Not used by MINT.

uptr An arbitrary pointer. Not used by MINT.

uptr2 An arbitrary pointer. Not used by MINT.

ufunc2 A function pointer. Not used by MINT.

unext A forward link for queues. Not used by MINT.

uprev A backward link for queues. Not used by MINT.

The fields that are not used by MINT may be used by the back-end writer to hold state between calls to back-end functions. Since these fields might not be needed by the back end, they are not initialized to zero and may contain garbage values. The back end is responsible for initializing the fields that it uses.

7 MINT command line options

The command line options supported by MINT are given below. These options (if any) are given on the command line before the object file name. The object file's arguments are given after the object file name. Command line options for the back end can be included after the MINT options following a double hyphen ("`--`"). If an error occurs while parsing the arguments, MINT prints a usage message and then calls the user-defined function `sim_usage()`.

```
mint [mint options] [-- back-end options] objfile [objfile options]
```

The options for MINT are:

`-C` *interval* The number of cycles between "checkpoint" intervals. This number is used to schedule a task at regular intervals. (This task does not really do any "checkpointing" – it simply calls `sim_checkpoint()`.) If set to zero, then no checkpoint tasks will be scheduled. See also the global variable "`Ckpoint_freq`" in section 6.3.

`-c` *file* The *file* specifies how many processor cycles certain operations should take. Each line contains the name of the operation followed by the number of cycles for the operation (separated by white space). The operations that may be specified include any instruction mnemonic, plus any of the functions callable from the object file that are replaced by MINT. These functions include fork(), malloc(), and so on. For a complete list of these functions, see section 8. The default cycle time for all operations is 1.

`-e` *file* The *file* specifies a list of events to generate. This is useful for debugging a back end. If this option is specified, then any object file argument is ignored. Each line in the file is of the form:

```
pid time addr type utype
```

Each line generates an event. The meanings of each field are:

`pid` The processor id.

`time` The event will not be generated until at least this time.

`addr` The address for read or writes. This value will be used to fill in the `paddr` and `vaddr` fields in the event structure.

`type` A string representing the type of event. MINT currently supports the following types:

read
write
user
lock_attempt
lock_acquire
lock_release
barrier_attempt
barrier_acquire
psema_attempt
psema_acquire
vsema

Only enough characters to uniquely identify the type of event are required. All reads and writes are assumed to be to shared memory.

`utype` An arbitrary numeric value used only with "user" type events. This field may be omitted for read and write events.

The fields are separated by white space. Blank lines and lines that begin with "`#`" are ignored. Decimal, hexadecimal, and octal number formats are supported in all numeric fields. Hexadecimal numbers begin with "`0x`", and octal numbers begin with a leading zero. Case is not significant.

Example:

```
# This is an example file for generating events.
# pid  time    addr   type  utype
  1       0  0x1000   read
  2      10  0x1000  write
  0      10  0x2000   user  0177
```

`-h` *heap_size* The size of the per-process area used for dynamically allocated memory (through calls to malloc(), for example). The default size is 64K. As with other numeric arguments, if the size argument begins with "`0x`", the size is interpreted as a hexadecimal number.

`-K` *keyfile* The "keyfile" argument specifies a file that contains a "key" used in socket connections. The "key" can be any string of characters, and starts with the first character in the file and ends with the first newline. A client program can connect to a running simulation through a socket and must send the key in order for the connection to be accepted by MINT. After the connection is established, the client program can send commands to the simulator. This can be used for displaying performance data or for debugging. If no keyfile is given, then MINT will accept a connection with any client. The socket connection is checked for pending connections only at checkpoint intervals. If the checkpoint interval is set to zero, then no socket connections are possible.

`-k` *stack_size* The size of the stack allocated for each process. The default size is 32K. As with other numeric arguments, if the size argument begins with "`0x`", the size is interpreted as a hexadecimal number.

`-l page=`*size*`,blk=`*size* Locks, semaphores, and barriers are allocated in shared memory. The default allocation policy is to allocate these synchronization variables on separate pages from other shared memory regions that the application program allocates. The default page size is 4096 bytes. If this variable is set to zero, then locks, semaphores, and barriers are not allocated in separate pages but are allocated in the next available shared memory space. The synchronization variables can also be allocated to separate cache blocks by specifying a non-zero block size. The default block size is zero – that is, a synchronization variable is not given its own cache block but is allocated starting at the next available memory location (within the synchronization page if the page size is non-zero). The values given for the page and block sizes are used only for controlling the allocation of locks, semaphores, and barriers. If the specified page size or block size is not a power of two, then it is rounded up to the next power of two. The `page` and `blk` arguments can appear in any order and either one may be omitted. As with other numeric arguments, if the size argument begins with "`0x`", the size is interpreted as a hexadecimal number.

`-m` *mem_size* This option can be used to control the total amount of private memory allocated to each new process. The size is the sum of the space needed for the data and bss sections of the object program plus the heap and the stack. This option is no longer needed in most cases since there are separate options for controlling the heap and stack sizes. If *mem_size* is less than the amount needed for the data, bss, heap, and stack, then an error message will be printed. If *mem_size* is larger than needed, then the extra space will be added to the heap. One possible use of this option is to align the per-processor private memory regions on *mem_size* boundaries but in most cases this is not necessary. As with other numeric arguments, if the size argument begins with "`0x`", the size is interpreted as a hexadecimal number.

`-O0` Turn off optimization. Normally, blocks of instructions are executed natively by creating specialized functions in memory at the beginning of the simulation.

`-O1` Execute blocks of instructions natively instead of interpreting each instruction with a function call. This is the default when executing on MIPS-based machines. It is not yet supported on the SPARC.

`-P` *port* This option specifies the port number to use for socket connections. It defaults to the first available port starting at 19030.

`-p` *procs* This option specifies the maximum number of processes that may be simulated. This also controls the amount of memory that is allocated for private data.

`-q` This option ("quiet") suppresses the printing of warning messages.

`-r` This option specifies that the back-end uses release consistency. Release consistency allows a processor to continue executing instructions past the end of a critical section without waiting for the lock release to finish (such as might happen if the lock release is buffered in a write buffer or is waiting for acknowledgement messages in a network system). MINT itself has no concept of write buffers or networks, but in order to support release consistency a couple new functions are required: `RC_unlock()` and `RC_vsema()`. These functions are called by the back-end to unlock a lock or to increment a semaphore (and wake up a waiting process, if necessary). These functions separate out the continuation of a process after a return from `sim_lock_release()` from the release of the lock. Since the use of these functions changes the semantics of `sim_lock_release()` and `sim_vsema()`, these functions must be used only if the "-r" command line option is given to MINT. The global variable `Release_consist` is true (1) if this option was specified.

`-S` Spin instead of blocking when trying to acquire a lock. If this option is selected, then an `E_LOCK_ATTEMPT` event will be generated repeatedly until the lock becomes free. If this option is not selected, then only one `E_LOCK_ATTEMPT` will be generated, and if the lock is not free, the process will block. Events for `E_LOCK_ACQUIRE` and `E_LOCK_RELEASE` remain unaffected by this option.

-s *shmem_size* This option controls the size of the shared memory segment. The default size is 16 megabytes. If the size argument begins with "0x", the size is interpreted as a hexadecimal number.

-t i This option instructs MINT to generate events for every instruction.

-t n This option instructs MINT to generate no events.

-t r This option instructs MINT to generate events for every memory reference. This is the default so it need not be specified.

-t s This option instructs MINT to generate events for every shared memory reference.

-v Print the version number for MINT.

-V This option turns on protocol verification. The back end must supply a value for all read events in the event structure. The value for write events is passed to the back end in the event structure.

-W This option forces MINT to check the target address of a write against any pending load-linked addresses. By default, MINT checks the load-linked addresses only on store-conditional instructions. This is faster but may not be the correct behavior for some programs. If this option is specified and the target address of a write matches a pending load-linked address, then a subsequent store-conditional to that address by the processor (or processors) that performed the load-linked will fail.

8 Application functions replaced by MINT

Some of the library functions that the object program calls are replaced by MINT with its own corresponding version. For example, if the object program calls fork(), MINT replaces the function pointer that would normally be called to interpret the first instruction of the fork() routine with a pointer to an internal function that simulates the fork().

The list of functions that MINT understands (and replaces with its own version) are given below. These functions are callable from within the object program. The names of these functions may also be used within a file that specifies the number of cycles each operation should take. (See the command line option [-c *file*] in section 7.)

Memory allocation: calloc(), free(), malloc(), realloc(), share_malloc(), shmalloc()

Locks: uscas(), uscsetlock(), usctllock(), usdumplock(), usfreelock(), usinitlock(), usnewlock(), ussetlock(), ustestlock(), usunsetlock(), uswsetlock()

Barriers: barrier(), free_barrier(), init_barrier(), new_barrier()

Semaphores: `closepollsema()`, `cpsema()`, `ctlsema()`, `dumpsema()`, `freepollsema()`, `freesema()`, `newpollsema()`, `openpollsema()`, `testsema()`, `usinit()`, `usinitsema()`, `usnewsema()`, `uspsema()`, `usvsema()`

System calls: `_exit()`, `access()`, `cachectl()`, `chdir()`, `chmod()`, `chown()`, `close()`, `connect()`, `creat()`, `dup()`, `fchmod()`, `fchown()`, `fcntl()`, `fork()`, `fstat()`, `getdtablesize()`, `getegid()`, `geteuid()`, `getgid()`, `gethostname()`, `getpid()`, `getsockopt()`, `gettimeofday()`, `getuid()`, `ioctl()`, `kill()`, `link()`, `lseek()`, `lstat()`, `open()`, `oserror()`, `perror()`, `pipe()`, `read()`, `readlink()`, `recv()`, `recvfrom()`, `recvmsg()`, `semctl()`, `semget()`, `semop()`, `select()`, `send()`, `sendmsg()`, `sendto()`, `setoserror()`, `setsockopt()`, `sginap()`, `shmat()`, `shmctl()`, `shmdt()`, `shmget()`, `signal()`, `sigsetmask()`, `sigvec()`, `socket()`, `sproc()`, `stat()`, `symlink()`, `times()`, `umask()`, `unlink()`, `wait()`, `wait3()`, `write()`

Miscellaneous: `_isatty()`, `clock()`, `generate_event()`, `m_fork()`, `time()`

9 Performance

The time to simulate a program run using MINT depends on several factors, such as the instruction mix, the number of simulated processors, the types of events that are generated, and the complexity of the target machine simulation. To measure the overhead of MINT itself (rather than the combined overhead of MINT plus the user-defined back end), MINT was linked with simple back-end functions that only count the number of reads and writes.

Compile-time macros allow the current version of MINT to represent simulated processor time as either "unsigned long" or "double" values. Using "double" is slower since every reference to a time value requires 2 loads or stores instead of one. It also makes searching hash tables indexed by time somewhat less efficient. The maximum range of values representable by 32-bit integers, however, is only about 4 billion, and at 400,000 simulated cycles per second, a simulation that ran for 3 hours would exceed the maximum representable time. Using doubles, however, gives 53 bits of accuracy. At the same rate of 400,000 simulated cycles per second, it would take over 700 years before the simulated time exceeds a value that cannot be represented exactly using double-precision floats. All of the MINT simulations were made using double-precision floats to represent time values. The Tango and PROTEUS runs, however, used 32-bit integers to represent time values.

Table 1 summarizes the memory reference properties of the test programs. The first three are matrix multiplication programs that differ only in the type of elements stored in the matrix. The application "mm" uses integers; "mm-sp" uses single-precision floats; and "mm-dp" uses double-precision floats. The input matrices were of size 300×300. Since integer multiplies require more instructions than floating point multiplies, the total number of instructions decreased for "mm-sp" compared to "mm". Using 8-byte "doubles", however, increases the number of loads and stores so "mm-dp" had many more memory references than "mm" did. The "gauss" program performs Gaussian elimination on a 400×400 matrix of single-precision floats. "MP3D" is a three-dimensional wind tunnel simulator from the SPLASH suite [11]. MP3D was run with 20,000 molecules for 50 time steps.

Appl.	Total refs ($\times 10^6$)	Shared refs ($\times 10^6$)	Total instr ($\times 10^6$)	Instr/ref
mm	82	82	241	2.94
mm-sp	82	82	187	2.28
mm-dp	137	136	242	1.77
gauss	65	64	146	2.25
MP3D	59	35	173	2.94

Table 1: Memory reference characteristics of the test programs. The "Total refs" column contains the total number of memory references (in millions). "Shared refs" is the number of references to shared memory. The total number of instructions (in millions) is in the column labelled "Total instr", and the average number of instructions per memory reference is in the column labelled "Instr/ref".

Appl.	Native (secs)	MINT (secs)	Slow-down	Simulated cycles per second
mm	9.8	335	34	720000
mm-sp	7.4	329	44	571000
mm-dp	28.4	517	18	469000
gauss	4.2	244	58	601000
MP3D	4.0	260	65	664000

Table 2: The execution time (in seconds) to simulate each of the application programs and the slowdown compared to the native execution time. MINT was linked with empty back-end functions and each application used 1 simulated process. Private as well as shared memory references were simulated.

As can be seen in Table 1, the ratio of instructions to memory references varies across programs. This ratio significantly affects the execution time of the simulation. A high ratio means that there is a high proportion of instructions that do not cause events. Programs such as integer matrix multiply with high instruction-to-memory-reference ratios can be simulated faster than programs with low ratios since there are fewer events to process for a given number of instructions.

The overhead of MINT varies significantly across programs. Table 2 lists the slowdown for each of the application programs. The slowdown for a given program is the cpu time (user plus system) that MINT used to simulate a 1-processor execution of the program divided by the cpu time for a 1-processor native execution on the same machine. All measurements were made on one node of a 12-processor SGI Challenge with 100-MHz R4400 processors. When generating events for every memory reference, a MINT simulation typically takes between 20 and 70 times longer than native execution on the same machine. This is comparable to the slowdown for uniprocessor MIPS code interpreters such as SPIM [8], for which a slowdown of 20 has been reported.

High cache miss rates explain why the native execution time for mm-dp is much larger than for mm and mm-sp. The relatively high percentage of memory reference instructions means that MINT has more work to do per instruction executed, so the "simulated cycles per second" rate is low for mm-dp. A load or store that misses in the cache under native execution is costly. Under the MINT simulation, however, all loads and stores take the same time to simulate with the simple back end we used. Thus, the overall slowdown ratio is smaller when simulating programs with poor memory performance than it is for programs with low cache miss rates.

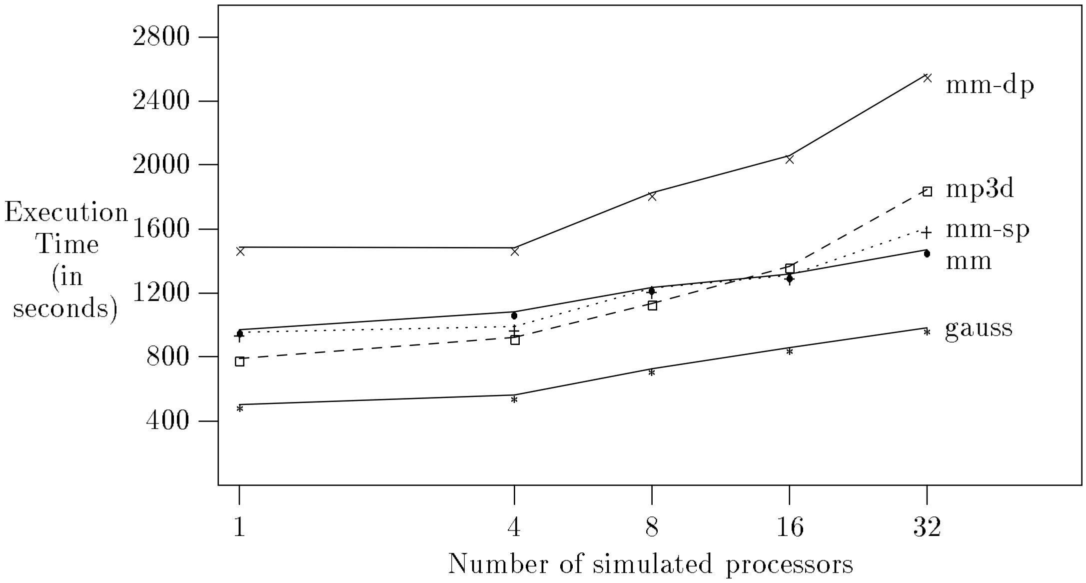

Figure 8: The execution time (in seconds) when using MINT with a realistic target system simulator.

The time required to simulate a program using a realistic target system simulator de-

Appl.	MINT (shared refs) min:sec	Tango (shared refs) hr:min:sec	times slower	MINT interpreting Tango hr:min:sec
mm	4:56	14:15:51	173	2:24:06
MP3D	3:20	5:26:00	98	0:19:49

Table 3: A comparison of the execution time overhead of MINT and Tango for 8-processor simulations.

pends on how much work the simulator performs per memory reference and this, in turn, usually depends on how well the application runs on the simulated target system. If the application has good locality and infrequent synchronization, then most of the loads and stores will be cache hits and the back-end simulator functions can return quickly. On the other hand, if the application runs poorly on the simulated machine, then the simulator functions have to do more work and the simulation takes longer. Figure 8 shows the time to simulate each application using MINT linked with a realistic target system simulator. The simulation models the memory hierarchy of a DEC 10000 multiprocessor. Primary and secondary caches are modelled as well as write buffers. Contention for the bus and lockout from the secondary cache are also modelled. Compared to MINT runs with empty back-end functions, the realistic simulations took about two to six times as long. This implies that most of the time is spent in the back end for a realistic simulation.

Since MINT is a sequential program, the total amount of work that MINT performs when simulating larger numbers of processors can be expected to increase at least as fast as the processor-time product of the simulated program. Figure 8 illustrates that MINT simulations scale efficiently as the number of simulated processors is increased.

Tango and PROTEUS also run on MIPS-based computers, so it is possible to do a direct comparison with MINT. To compare just the overhead of each simulation system, each was linked with empty back-end functions. For the MINT runs, the sources of the parallel programs were compiled and linked and then interpreted by MINT. For the Tango runs, the same sources were compiled with the modifications required by Tango and linked with the Tango libraries. For the PROTEUS runs, the application sources were annotated as required by PROTEUS and then compiled and linked with the PROTEUS support routines. (The built-in cache and bus simulator code was excised.) Events were generated for every shared memory reference. (With MINT, the type of events to generate is selected by a command line option at run time. With Tango and PROTEUS, the type of events to generate must be decided at compile time.)

The time required for MINT to simulate the MP3D program was about 3 minutes. (See Table 3.) By contrast, Tango took over 5 hours. For matrix multiplication, MINT took about 5 minutes, while Tango took over 14 hours.

Since the Tango simulators are themselves executable files, MINT can interpret them just like any other executable. This produces the same effect as running the Tango simulation.

Applications	MINT (shared refs) min:sec	PROTEUS (shared refs) min:sec	× slower
mm	4:56	28:34	5.8
MP3D	3:20	10:52	3.3

Table 4: A comparison of the execution time overhead of MINT and PROTEUS for 8-processor simulations.

The only difference is that MINT is interpreting the instructions and simulating the multiple Tango processes within a single address space. (For this experiment only, Mint simulated the multiprocessor execution without calling the back-end functions on memory references.) Surprisingly, MINT interpreting Tango simulating a program is faster than Tango running natively simulating the same program. The reason for this is that Tango uses System V semaphores to synchronize between Unix processes, which causes kernel traps and expensive context switches. In MINT, these expensive semaphore calls are simulated by a simple function call within the same address space. A new, unreleased version of Tango that uses lightweight threads should significantly reduce the overhead of this synchronization.

A comparison of MINT and PROTEUS is given in Table 4. MINT is over 3 times faster than PROTEUS when simulating MP3D, and almost 6 times faster than PROTEUS when simulating matrix multiply. PROTEUS, like MINT, represents simulated processes within a single address space and thus avoids the high cost of synchronization between Unix processes. The difference in performance between MINT and PROTEUS is largely due to the relative costs of switching between thread contexts. PROTEUS saves and restores all the registers on a context-switch; MINT simply changes a pointer, since all registers are kept in memory.

10 A simple cache simulator

This section demonstrates more of the features of MINT by explaining the design of a simple cache simulator. This program simulates an infinite size cache using a write-invalidate protocol. A read miss in the cache causes a bus read request to be issued. A write miss causes an exclusive read request to be issued, and a write hit to a shared block causes an invalidation request to be issued. Each request must compete for the bus. If the bus is busy, each request retries when the bus becomes free. The source for this program is in the files "cache.c" and "bus.c" in the "examples" directory.

10.1 Data structures

Each cache block has state associated with it that needs to be kept consistent across all processors. All the information about a cache block is stored in a centralized data structure. This makes updating and retrieving the state of the cache block simple and efficient. The cache state includes a present bit, a shared bit, and a dirty bit for each processor. A bit vector is used to store the values of the bits for each processor. For example, bit i in the "present" bitvector is 1 if the cache block is valid in processor i's cache. In the current implementation, a single 32-bit integer is used to store a bit vector so the number of processors is limited to 32. This limit can be increased by using an array of integers. All bit operations are performed using macros. This causes the operation to be inlined efficiently, makes it easy to increase the processor limit by replacing the macro definitions, and makes certain kinds of debugging easier because run-time checks can be inserted into the macro definitions. The bit vector macros are defined in "bit.h" in the "include" directory.

A hash table is used to store the cache block structures. The table is indexed by cache block number, and a pointer to the corresponding structure containing information about the cache block is returned. The first field of the cache block structure is used to store the cache block number, and the second field is used to link cache blocks that hash to the same entry in the hash table. The structure definition is given below:

```
/* define a structure to hold information about a cache block */
typedef struct info_t {
    unsigned long blocknum;
    struct info_t *next;
    bitvector_t present;
    bitvector_t shared;
    bitvector_t dirty;
} info_t, *info_ptr;

#define ENTRY_SIZE sizeof(struct info_t)
```

The hash table is initialized by calling `hash_init()`, passing it the number of entries to create in the hash table as the first argument. (The number of entries will be rounded up to a power of two.) The second argument is the size of the user-defined structure. This function returns a pointer to a hash table object. The hash table functions are provided as part of the MINT library and are independent of the particular type of structures stored in the hash table. The only requirement is that the structure has at least two fields: the first field is used to store the key (in this case the cache block number) that is used to index into the hash table, and the second field is a pointer.

The declaration of the hash table to store cache blocks is:

```
/* hash table size must be a power of two */
#define HASH_TABLE_SIZE (64 * 1024)

/* Cache is a pointer to the hash table data structure for storing
 * cache blocks.
 */
hash_tab_ptr Cache;
```

and it is initialized with:

```
Cache = hash_init(HASH_TABLE_SIZE, ENTRY_SIZE);
```

To retrieve the information about a cache block (and insert a new entry if not found):

```
pinfo = (info_ptr) hash_insert(Cache, blocknum);
```

If the cache block number is not found, then a new entry of size `ENTRY_SIZE` is allocated, initialized to zero, and inserted into the hash table. The return value from `hash_insert()` is a pointer of type `hash_entry_ptr`. This points to a generic hash table entry (which contains only the first two required fields) so the return value must be cast to type `info_ptr`.

10.2 sim_read()

The code for `sim_read()` is given below. This function is called on every read event. By default, a read event is generated for every load instruction but by using the command line option "`-t s`" read events will be generated only for load instructions that read shared memory.

```
sim_read(task_ptr ptask)
{
    int pid, addr, blocknum;
    event_ptr pevent;
    task_ptr newtask;
    info_ptr pinfo;

    pid = ptask->pid;
    pevent = ptask->pevent;
    addr = pevent->paddr;

    /* shift the addr to produce a cache block number */
    blocknum = addr >> Blockshift;

    /* lookup the entry for the cache block containing address addr */
    pinfo = (info_ptr) hash_insert(Cache, blocknum);

    /* if the block is present, it's a read hit */
    if (ISVALID(pinfo->present, pid)) {
        Data.read_hit++;
        return T_ADVANCE;
    }

    /* block is not present, issue a bus read request */
    issue_bus_request(ptask, read_retry, read_done, pinfo);

    Data.read_miss++;
```

```
    return T_FREE;
}
```

The memory address being read is extracted from the event structure and then right-shifted to obtain a cache block number. This cache block number is used to index the hash table.

The macro `ISVALID()` is used to test the bit in the "present" bit vector corresponding to process `pid`. If the bit is on (equals 1), then the cache block entry is present in this processor's cache, so we allow the processor to continue by returning `T_ADVANCE`.

If the cache block is not present, then we have to read it using the bus. All bus operations are requested by calling `issue_bus_request()`. The first argument is the task pointer, which contains the current time. The second and third arguments are function pointers. The first function pointer points to a "retry" function. If the bus is busy, the bus module schedules a task to execute the retry function when the bus next becomes free. The second function pointer points to a "done" function. If the bus is free, the bus module schedules a task to execute the done function when the bus operation completes. Using these function pointers simplifies the bus module and allows it to be independent of any particular cache implementation. The cache module is responsible for implementing the appropriate retry and done functions. The last argument is the `pinfo` pointer. This pointer is passed in so that it can be stored in the tasks that the bus module creates. This pointer is then extracted by the retry and done functions in order to avoid another hash table lookup.

10.3 `read_retry()`

The `read_retry()` function is simple. It issues a bus read request and returns `T_FREE` to cause the processor waiting for this read to complete to remain blocked.

```
read_retry(task_ptr ptask)
{
    info_ptr pinfo;

    /* retrieve the pinfo pointer from this task */
    pinfo = (info_ptr) ptask->uptr;

    /* retry the read request */
    issue_bus_request(ptask, read_retry, read_done, pinfo);
    return T_FREE;
}
```

10.4 `read_done()`

The `read_done()` function changes the state of the cache block for processor `pid`. It sets the present bit, clears all dirty bits, and if any other processor has this block present, it sets the shared bit. This function returns `T_ADVANCE` to allow the processor to continue.

```
read_done(task_ptr ptask)
{
    info_ptr pinfo;
    int pid;

    /* retrieve the process id and the pinfo pointer from this task */
    pid = ptask->pid;
    pinfo = (info_ptr) ptask->uptr;

    /* make this block present in pid's cache */
    SETVALID(pinfo->present, pid);

    /* clear the dirty bits for everyone */
    CLRALLBITS(pinfo->dirty);

    /* if anyone else has a copy then this block is known to be shared */
    if (!ISEXCL(pinfo->present, pid))
        SETEQUAL(pinfo->shared, pinfo->present);
    else
        CLRALLBITS(pinfo->shared);

    return T_ADVANCE;
}
```

10.5 sim_write()

The `sim_write()` function is a little more complicated than `sim_read()` because it has to issue an invalidation on a write hit if the cache block is present and shared. If the block is present but not shared, it simply sets the dirty bit. If the cache block is not present, it issues an exclusive read request. The processor may continue if this is a write hit to a private block; otherwise, the processor must remain blocked.

```
sim_write(task_ptr ptask)
{
    int pid, addr, blocknum;
    event_ptr pevent;
    task_ptr newtask;
    info_ptr pinfo;

    pid = ptask->pid;
    pevent = ptask->pevent;
    addr = pevent->paddr;

    /* shift the addr to produce a cache block number */
    blocknum = addr >> Blockshift;
```

```
    /* lookup the entry for the cache block containing address addr */
    pinfo = (info_ptr) hash_insert(Cache, blocknum);

    /* if the block is present, it's a write hit */
    if (ISVALID(pinfo->present, pid)) {
        Data.write_hit++;
        if (ISVALID(pinfo->shared, pid)) {
            /* block is shared, issue an invalidation */
            issue_bus_request(ptask, inv_retry, inv_done, pinfo);
            return T_FREE;
        } else {
            /* block is not shared, set the dirty bit */
            SETVALID(pinfo->dirty, pid);
            return T_ADVANCE;
        }
    }

    /* block is not present, issue a bus exclusive-read request */
    issue_bus_request(ptask, excl_retry, excl_done, pinfo);

    Data.write_miss++;
    return T_FREE;
}
```

10.6 inv_retry()

This inv_retry() function is called if the invalidation request could not be handled immediately because the bus was busy. This function is scheduled by the bus module to execute at the next time the bus becomes free. Since this block may have been invalidated by some other processor while we were waiting for the bus, we must check if the block is still valid. If the block is still valid, we retry the invalidation (and keep trying until we either get the bus or some other processor invalidates this block). If the block is no longer valid, we need to treat this as a write miss, so we issue an exclusive-read request.

```
inv_retry(task_ptr ptask)
{
    info_ptr pinfo;
    int pid;

    /* retrieve the process id and the pinfo pointer from this task */
    pid = ptask->pid;
    pinfo = (info_ptr) ptask->uptr;

    /* if this block is still present, then re-issue the invalidation request */
```

```
    if (ISVALID(pinfo->present, pid)) {
        issue_bus_request(ptask, inv_retry, inv_done, pinfo);
        return T_FREE;
    }

    /* this block is no longer present, so exclusive-read the cache block */
    issue_bus_request(ptask, excl_retry, excl_done, pinfo);
    return T_FREE;
}
```

10.7 `inv_done()`

The `inv_done()` function is called when an invalidate completes. This function changes the state of the cache block. The cache block is made present, dirty and non-shared for processor `pid`. The bits for all other processors are cleared.

```
inv_done(task_ptr ptask)
{
    info_ptr pinfo;
    int pid;

    /* retrieve the process id and the pinfo pointer from this task */
    pid = ptask->pid;
    pinfo = (info_ptr) ptask->uptr;

    /* make this block present only in pid's cache */
    SETEXCL(pinfo->present, pid);

    /* set my dirty bit, clear everyone else's dirty bit */
    SETEXCL(pinfo->dirty, pid);

    /* this block is not shared so clear the shared bits */
    CLRALLBITS(pinfo->shared);

    return T_ADVANCE;
}
```

The functions for `excl_retry()` and `excl_done()` are similar to the functions for `read_retry()` and `read_done()` and can be found in the file "cache.c".

10.8 `issue_bus_request()`

The bus module uses a global variable to keep track of how long the bus is busy. The declaration of this variable is:

```
/* time that bus is busy until */
static mint_time_t Bus_busy_until;
```

Variables that are used to store time values should be declared using type `mint_time_t`. In the current implementation of MINT, this is a typedef for `double`. Although operations on an `unsigned long` are more efficient than operations on a `double`, the use of type `double` allows longer simulation runs without overflowing the precision of the time variable. If the extra precision of type `double` is not needed, MINT can be recompiled so that it uses type `unsigned long` by defining `ULONG_TIME`. This change typically reduces the time for a simulation by 3 to 8 percent.[4]

Future implementations of MINT may use 64-bit integers (if supported by the compiler and underlying hardware) for storing time values so the back end should not make any assumptions about the type of `mint_time_t`. When printing time values using `printf()`, the back end must know what format to use. In this case, it is safest to use "`%.0f`" and cast the time value to type `double`.

The bus module code is short and simple. If the bus is free, it schedules a task to execute `func_done` when the bus operation completes. If the bus is busy, it schedules a task to execute `func_retry` when the bus becomes free. The `func_done` function is scheduled at priority 10, which is higher than the default of zero, so that if a "done" function and a "retry" function are scheduled for the same time slot, the "done" function will execute first. This is necessary for correctness, since the "done" function may change some state that the "retry" function depends on.

The pointer `uptr` is stored in the newly scheduled task so that the "retry" and "done" functions can use it. The type of `uptr` is `void *` so that the bus module does not need to know anything about the actual type of the pointer.

```
void
issue_bus_request(task_ptr ptask, PFTASK func_retry, PFTASK func_done,
                  void *uptr)
{
    task_ptr newtask;

    /* If the current time is past the time the bus is busy until,
     * then the bus is free.
     */
    if (ptask->time >= Bus_busy_until) {
        /* bus is free, make it busy for BUS_BUSY cycles */
        Bus_busy_until = ptask->time + BUS_BUSY;

        /* schedule the done function BUS_BUSY cycles from now */
        newtask = pri_sched_task(ptask, func_done, Bus_busy_until, 10);
```

[4] A special hand-optimized assembly language routine is provided when using type `double` for time values so this is actually faster now than using `unsigned long`. Support for time values of type `unsigned long` will probably be discontinued in the future.

```
        newtask->uptr = uptr; /* save the user-defined pointer */
    } else {
        /* bus is busy, schedule the retry function at Bus_busy_until */
        newtask = sched_task(ptask, func_retry, Bus_busy_until);
        newtask->uptr = uptr; /* save the user-defined pointer */
    }
}
```

To compile the cache simulator, type (in the "examples" directory):

```
make cache
```

11 Acknowledgements

Special thanks are due to Leonidas Kontothanassis and Ricardo Bianchini who, being the first users of MINT (other than myself), discovered some bugs in the early versions and helped contribute to the evolution of the design. Leonidas also helped write some of the code for managing the task queues. Mike Marchetti wrote the TCL-based graphic display that uses the socket interface to display performance data from a running simulation.

References

[1] James Boyle, Ralph Butler, Terrence Disz, Barnett Blickfeld, Ewing Lusk, and Ross Overbeek. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, 1987.

[2] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, MIT, sep 1991.

[3] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice Parallel Processing Testbed. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 4–11, 1988.

[4] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of the 1991 Conference on Parallel Processing*, pages II–99–II–107, aug 1991.

[5] Susan J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy. Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 37–47, 1990.

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

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

[8] James R. Larus. SPIM S20: A MIPS R2000 Simulator. Technical Report TR966, Computer Sciences Department, University of Wisconsin-Madison, sep 1990.

[9] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Columbia University, 1992.

[10] 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 *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, may 1993.

[11] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *ACM SIGARCH Computer Architecture News*, 20(1):5–44, 1992.

[12] Craig B. Stunkel and W. Kent Fuchs. TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation. In *Performance Evaluation Review, 17(1)*, pages 70–78, May 1989.