# Chapter 5
# Simulation Methodology

This chapter describes the simulation methodology used for the protocol comparisons in this study. Since this research examines the scalability of cache coherence protocols to 128 processor machines, it must simulate applications that can scale to that number of processors. The applications used, the multiple variants of the same application that were simulated, and the problem sizes of the applications are discussed in Section 5.1. Section 5.2 describes the FLASH simulator, including both the processor model, and FlashLite, the detailed memory system simulator for the Stanford FLASH multiprocessor. The chapter concludes with a discussion of synchronization in Section 5.3.

## 5.1 Applications

To properly assess the scalability and robustness of cache coherence protocols it is necessary to choose applications that scale well to large machine sizes. This currently translates to the realm of scientific applications but does not limit the applicability of this study. As will be shown in Chapter 6, this study finds that protocol performance is important over a reasonable spectrum of parallel applications, and that the optimal cache coherence protocol can change with the application or the architectural parameters of the machine.

The applications are selected from the SPLASH-2 application suite [62]. In particular, this study examines FFT, Ocean, Radix-Sort, LU, Barnes-Hut, and Water. All applications except Barnes-Hut and Water use hand-inserted prefetches to reduce read miss penalty [35]. To further improve scalability, the applications use software tree barriers rather than traditional spin-lock barriers. For some applications this improved performance by over 40% at large processor counts.

So that the applications achieve reasonable parallel performance, their problem sizes are chosen to achieve a target minimum *parallel efficiency* at 128 processors. Parallel efficiency is simply defined as speedup divided by the number of processors, and always varies between 0 and 1. An application's problem size is determined by choosing a target minimum parallel efficiency of 60% for the best version of the application running the

best protocol at 128 processors. Table 5.1 lists the problem sizes used for each of the applications in this study.

**Table 5.1. Applications and Problem Sizes**

| Application | Description | Problem Size |
|---|---|---|
| FFT | Radix $\sqrt{n}$ Fast Fourier Transform | 1M Points |
| Ocean | Multi-grid ocean simulation | 514x514 grid |
| Radix-Sort | Integer radix sort | 2M keys |
| LU | Blocked dense lu decomposition | 768x768 matrix |
| Barnes-Hut | N-body galaxy simulation | 16384 bodies |
| Water | Molecular dynamics simulation | 2048 molecules |

Of the six applications in Table 5.1, three (Ocean, Barnes-Hut, and Water) are complete applications and three (FFT, Radix-Sort, and LU) are computational kernels. All of the applications are highly-optimized so that they can scale to large machine configurations. The applications are taken from the SPLASH-2 application suite, with a few modifications. The global error lock in the multi-grid phase of Ocean has been changed from a lock, test, and set structure to a test, lock, test, and set structure. This improved performance by 34% at 64 processors. Radix-Sort uses a tree structure to communicate the ranks and densities more efficiently than the linear structure in the SPLASH-2 code.

In addition, multiple versions of each application are examined, varying from highly-optimized to less-tuned versions of each application. Most of the applications have two main optimizations that are selectively turned off: data placement, and an optimized communication phase. All of the most optimized versions of the applications include data placement to optimize communication. This study looks at the relative performance of cache coherence protocols for versions of each of these applications both with and without data placement. For FFT, Radix-Sort, LU, and Barnes-Hut, it is also possible to run a less-tuned communication phase by changing compile-time constants. Table 5.2 describes the changes in these un-optimized versions with respect to the base optimized application.

As another architectural variation, the less-tuned versions of the applications are also run with smaller 64 KB processor caches. Since this cache size is smaller than the working sets of some of our applications [43], these configurations place different demands on the

**Table 5.2. Description of Less-Tuned Application Versions**

| Application | Description |
|---|---|
| FFT | Unstaggered versus staggered transpose phases |
| Radix-Sort | O(P) global histogram merge phase, versus O(log P) |
| LU | Explicit barriers between the three communication phases: diagonal factor, perimeter update, and interior update |

cache coherence protocols than the large-cache configurations, and lead to some surprising results.

Chapter 6 presents results for the most optimized version of each application, and then progressively “de-optimizes” the application in a manner similar to the way most applications are optimized. Typically the last thing done for shared memory programs is data placement, so that is turned off first. After presenting the results without data placement, results are presented for the application run without both data placement *and* the optimized communication phase (since it typically does not make sense to perform data placement in the versions with the less-tuned communication phase). Finally the cache size is reduced to examine the impact that architectural parameter has on the choice of cache coherence protocol.

## 5.2 The FLASH Simulator

At the time of this writing there is a 4-processor FLASH machine running in the lab. The machine is stable and is running the dynamic pointer allocation protocol. When this research began, however, a FLASH machine did not exist. Moreover, there is still no FLASH machine larger than 4 processors, and there may never be a 128 processor machine. Consequently, the results for the performance, scalability, and robustness of cache coherence protocols presented in Chapter 6 are obtained from detailed simulation. This section describes the FLASH simulation environment, including both the processor model described in Section 5.2.1, and the memory system model described in Section 5.2.2.

### 5.2.1 Processor Model

Execution-driven simulation is used to produce the results in this study. The processor simulator is Mipsy, an emulation-based simulator that is part of the SimOS suite [41] and interfaces directly to FlashLite, our system-level simulator. Mipsy models the processor and its caches, while FlashLite models everything from the processor bus downward.

In this study Mipsy simulates a single-issue 400 MHz processor with blocking reads and non-blocking writes. Although the MIPS R10000 is a superscalar 200MHz processor, the FLASH simulator is run with a single-issue processor model running 4 times faster than the memory system rather than 2 times faster. This faster speed effectively sets the average instructions per clock (IPC) to 2.0, approximating the complex behavior of the R10000 and yielding a more realistic interval between memory references.

Mipsy controls both the first and second-level caches, which are parameterized to match the R10000. The processor model has split first-level instruction and data caches of 32 KB each and a combined 1 MB, 2-way set-associative secondary cache with 128 byte cache lines. There is a four entry miss handling table that holds all outstanding operations including read misses, write misses, and prefetch operations. Though Mipsy has blocking reads, this study uses prefetched versions of the applications to simulate a more aggressive processor and memory system. Moreover, all the protocols operate in the relaxed EAGER consistency mode that allows write data to be returned to the processor before all invalidation acknowledgments have been collected [18]. To run in EAGER consistency mode, all unlock operations use the R10000 `sync` instruction to ensure that all previous writes have completed before the unlock operation. In addition, some flag writes in Barnes-Hut needed to perform `sync` operations before the write to the flag.

### 5.2.2 FlashLite

FlashLite is the lightweight, threads-based, system-level simulator for the entire FLASH machine. FlashLite uses Mipsy as its execution driven processor model [41] to run real applications on a simulated FLASH machine. FlashLite sees every load and store miss in the application and lets each memory reference travel through the FLASH node, giving it the proper delay as it progresses. FlashLite allows easy modeling of functional units and independent state machines that may interact in complex ways.

```
main() {
   create_task(Consumer, ...);
   create_task(Producer, ...);
   await(&FinishEvent, 1);
   printf("Program finished successfully\n"); // here at time 15
}

void Producer(void) {
   pause(10);                 // wait 10 cycles
   advance(&ProducerReady);   // advance Consumer thread
   await(&ConsumerDone, 1);   // wait for response
   advance(&FinishEvent);     // advance main()
}

void Consumer(void) {
   await(&ProducerReady, 1); // wait for producer to produce
   pause(5);                 // wait 5 cycles
   advance(&ConsumerDone);   // unblock producer
}
```

*Figure 5.1.* FlashLite threads package code example

Figure 5.1 shows pseudo-code for a simple producer-consumer program written with the FlashLite threads package. The event-based threads package has three main routines: `advance`, `await`, and `pause`. In this example the `main` routine creates the `Producer` and `Consumer` threads, and then waits on the event FinishEvent. Each event is simply a counting semaphore with an initial eventcount of zero. When a thread wants to wait on an event, it calls `await` and passes the count to wait for as the second argument. An `await` call will block (de-schedule the active thread) if the eventcount of the specified event is less than the count passed in as the second argument. When the `main` thread blocks, either the `Producer` thread or the `Consumer` thread will be run since they are both ready. The `Consumer` immediately waits on the ProducerReady event, making the `Producer` thread the only ready thread. The `Producer` thread wants to do 10 cycles worth of work before notifying the `Consumer` thread that it is done. The `pause` call de-schedules the `Producer` thread for 10 cycles. There is a single global eventcount for time that is a standard part of the threads package. Since there are no other active threads at this point, time is increased by 10 cycles and the `Producer` thread is scheduled again. This time it calls `advance` on ProducerReady which increments its eventcount to 1, and then blocks on an `await` call on the ConsumerDone event. Since the `Consumer` thread was waiting

for the ProducerReady eventcount to be 1, it is marked as a ready thread at the point of the `advance`. However, the `Producer` thread does not yield control until it calls await. This is an important and powerful feature of the threads package—all actions inside a thread are atomic until the thread calls either `await` or `pause`. Only on an `await` or a `pause` will a thread yield control to another available thread. This makes it easier to write critical sections and manage potentially complex thread interactions without giving up any of the power of parallel threads. To finish our example, the `Consumer` thread then does 5 cycles worth of work and informs the `Producer` that it has finished. The `Producer` wakes up and informs `main` that it is finished as well. Finally, the `main` routine wakes up and ends gracefully at time 15.

Though the example above is purposefully simple, it is possible to model very complex interactions quite easily with the threads package. To make it easy to model points of arbitration (like scheduling a shared bus): the internal simulator clock runs at twice the clock frequency of the system. It is easiest to think of this as running the simulation at half-cycle granularity (or alternatively as being analogous to two-phase hardware design). All normal system threads call `advance`, `await`, or `pause` on integral cycle counts. If two threads both want a shared bus on the same cycle, both may signify this with an advance of some event. Normally an arbiter thread would wake up when this event is advanced and that arbiter thread would run a scheduling policy to decide which of the two requesters actually gets the bus. But the arbiter thread cannot run on integral cycle multiples, because it needs to wait until all possible requests for the bus have been posted to make the proper scheduling decision, and there is no implicit thread ordering within the same cycle. To solve this problem, all arbitration threads run on half-cycle multiples. In this manner an arbiter thread can now see everyone who made requests for the bus on the previous integral cycle, and make an informed decision as to who really gets the bus. After making the decision, the arbiter thread waits a half-cycle to re-align itself to whole system clocks and advances the eventcount of the winner.

Accuracy in the simulation environment is critical in assessing the performance differences of cache coherence protocols. This is especially true in the model of the communication controller, in this case the MAGIC chip and in particular, the protocol processor. For example, rather than simulating the exact execution of the protocol code on the proto-

col processor, one could calculate an approximate average protocol processor occupancy per handler, and then simulate using that fixed occupancy. The normal FlashLite simulation will yield the average protocol processor occupancy for that run. The same simulation run with that average occupancy in the fixed occupancy version of the simulator produces a much different result. The results indicate that the fixed occupancy runs can overestimate performance by up to 22%. This result shows that averages are misleading, and that modeling occupancy properly can reveal hot-spots in the memory system, and result in more accurate performance comparisons.

FlashLite models everything in the system from the processor bus downward, including the MAGIC chip, the DRAM, the I/O system, and the network routers. The MAGIC chip alone is comprised of 14 independent FlashLite threads, as shown in Figure 5.2, capturing the parallelism within the MAGIC chip as well as the contention both within the chip and at its interfaces. FlashLite's parameters are taken directly from the Verilog RTL description of the FLASH machine [23], although the FlashLite thread models do not contain enough information to be cycle accurate with the Verilog model. The MAGIC chip runs at 100 MHz. The memory system has a bandwidth of 800 MB/s and a 140 ns access time to the first double word. The network also has a bandwidth of 800 MB/s and a per-hop

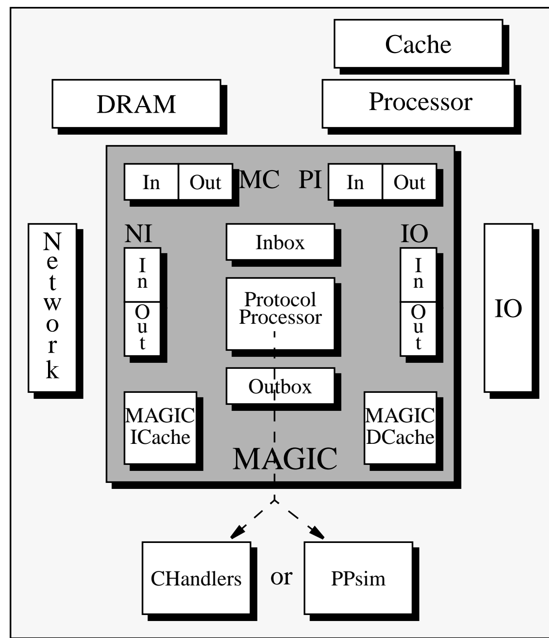


*Figure 5.2.* FlashLite threads modeling the entire FLASH system. FlashLite's protocol processor thread is itself a low-level instruction set emulator (PPsim).

**Table 5.3. MAGIC Latencies (10ns system cycles)**

| Internal MAGIC Operation | Latency |
|---|---|
| **Processor Interface**: | |
| Inbound processing | 1 |
| Outbound processing | 4 |
| Retrieve state reply from processor cache | 15 |
| Retrieve first double word of data from processor cache | 20 |
| **Network Interface**: | |
| Inbound processing | 8 |
| Outbound processing | 4 |
| **Inbox**: | |
| Queue selection and arbitration | 1 |
| Jump table lookup | 2 |
| **Protocol Processor**: | |
| Handler execution | Varies |
| MAGIC data cache miss penalty | 29 |
| **Outbox:** | |
| Outbound processing | 1 |
| | |
| **External System Operations** | |
| **Router**: | |
| Network transit, per-hop | 4 |
| **Memory System**: | |
| Access, time to first 8 bytes | 14 |

latency of 40 ns. FlashLite accurately models the network routers, using the hypercube topology employed by the real machine. The delays through each external interface of the MAGIC chip are shown in Table 5.3.

The protocol code itself is written in C and compiled and scheduled for the dual-issue protocol processor. Protocol data is accessed via the 1 MB direct-mapped MAGIC data cache. The protocol code used in the simulator is the *exact* code run on the real FLASH machine, and is the output from the protocol development tool chain detailed in Section 4.1.1. FlashLite's protocol processor thread is PPsim, the instruction set emulator for the protocol processor. PPsim emulates the same protocol code, providing cycle-accurate pro-

tocol processor timing information and precise MAGIC cache behavior. To factor out the effect of protocol instruction cache misses, a perfect MAGIC instruction cache is simulated in this study, rather than the normal 16 KB MAGIC instruction cache. Due to hardware implementation constraints, the real MAGIC instruction cache is undersized. As the protocol code sizes in Table 4.15 suggest, even a 32 KB instruction cache would capture the important handler working sets in all four of the protocols. While SCI and COMA do have larger code sizes than bit-vector and dynamic pointer allocation, the number of instruction cache misses can be reduced with a more realistic instruction cache size and with more aggressive instruction cache packing techniques [57][63].

## 5.3 Synchronization

The scientific applications in this study initially used LL/SC barriers when the application required a barrier, but the simulations reproduced the well-known result that LL/SC barriers are not scalable. As an alternative, the applications were re-coded to employ software tree barriers. There is very little difference between LL/SC barriers and tree barriers on 16 processor systems, but for larger machine sizes the tree barrier code is significantly faster. For example, in Ocean the tree barrier code is 1.1 times faster on 32 processor systems, 1.3 times faster on 64 processor machines, and an eye-popping 2.9 times faster on 128 processor machines. The results presented in Chapter 6 use software tree barriers to help the applications naturally scale to 128 processors. This enables the measurement of the performance impact of the different cache coherence protocols on a FLASH machine that is fundamentally operating well.

Aside from allowing multiple cache coherence protocols, the flexibility of the MAGIC chip enables the implementation of efficient synchronization primitives and custom synchronization protocols. Other work has focused on implementing tree barriers using only MAGIC chips, and implementing a custom scalable lock protocol [22]. As that research proceeded concurrently with this, the results presented here do not use the special MAGIC synchronization primitives. Although MAGIC synchronization primitives would slightly improve the performance of the most-optimized versions of the applications, their affect on the less-optimized application versions is less clear. Many of those applications have large read and write stall times which would not be affected by the new MAGIC synchro-

nization. However, some of the less-optimized applications do suffer from severe hot-spotting during global synchronization, a condition which could be improved by using MAGIC synchronization.

From the perspective of this research, it is not critical which scalable synchronization methods are used as long as the machine is operating well for the most optimized applications. Either method of synchronization would still produce the results in the next chapter that highlight the differences between the four cache coherence protocols as the machine size scales, application characteristics change, or architectural parameters vary.