

# Parallel Simulation of Chip-Multiprocessor Architectures

**MATTHEW C. CHIDESTER AND ALAN D. GEORGE**

*High-performance Computing and Simulation (HCS) Research Laboratory*  
Department of Electrical and Computer Engineering, University of Florida  
P.O.Box 116200, Gainesville, FL 32611-6200

Primary Contact Author

Dr. Alan D. George  
*High-performance Computing and Simulation (HCS) Research Laboratory*  
Department of Electrical and Computer Engineering  
University of Florida  
216 Larsen Hall, P.O. Box 116200  
Gainesville, FL 32611-6200

Phone: (352)392-5225  
Fax: (352)392-8671  
e-mail: [george@hcs.ufl.edu](mailto:george@hcs.ufl.edu)

Additional Author Contact Information

Dr. Matthew C. Chidester  
*Intel Corporation*  
8104 SW Charlotte Drive  
Beaverton, OR 97007  
  
Phone: (503)259-0831  
e-mail: [matthew.c.chidester@intel.com](mailto:matthew.c.chidester@intel.com)

# Parallel Simulation of Chip-Multiprocessor Architectures

Matthew C. Chidester and Alan D. George

**Abstract**—Chip-multiprocessor (CMP) architectures present a challenge for efficient simulation, combining the requirements of a detailed microprocessor simulator with that of a tightly-coupled parallel system. In this paper, a distributed simulator for target CMPs is presented based on the Message Passing Interface (MPI) designed to run on a host cluster of workstations. Microbenchmark-based evaluation is used to narrow the parallelization design space concerning the performance impact of distributed vs. centralized target L2 simulation, blocking vs. non-blocking remote cache accesses, null-message vs. barrier techniques for clock synchronization, and network interconnect selection. The best combination is shown to yield speedups of up to 16 on a 9-node cluster of dual-CPU workstations, partially due to cache effects.

**Index Terms**—Chip-multiprocessor, parallel simulation, MPI, SCI, Myrinet

## 1 INTRODUCTION

Simulation is an important tool in the development of new microprocessors. A design must be rigorously validated prior to implementation to ensure that it is both functionally correct and performs well. A large range of test cases must be explored while many configurations are compared to optimize the design. As microprocessors become more complex, encompassing both a larger number of transistors and numerous architectural features for increased performance, the design space that must be evaluated through simulation explodes.

In the design of parallel systems, simulation times are increased further by the need to simulate multiple processors, a still wider range of inputs, and larger datasets. One technique for reducing the simulation time is to scale datasets down in size [8], but this approach introduces inaccuracies and necessitates a detailed analysis of each workload to determine which part(s) can be safely scaled. Another performance enhancement involves simplified processor models with an emphasis on accurate memory subsystem and interconnect simulation [24]. However, these simplified models do not reflect the behavior of modern superscalar, out-of-order processors. A third method to address simulation explosion is to make use of parallel simulation.

Parallel simulation employs multiple processing nodes to increase the simulation rate. A common approach is to perform separate simulations for different settings of parameters—i.e. cache sizes, workloads, datasets, etc.—simultaneously on different processors of a parallel system. While this approach can greatly increase the throughput of the simulations, it does not reduce the latency required for a single simulation to finish. Often, a designer desires rapid feedback regarding a specific change in order to guide future decisions. For such situations, low-latency turnaround time is preferable to high throughput. Parallel simulation of event-driven models such as logic- and circuit-level evaluation has been successful in reducing these computationally intensive tasks. However, many of the architectural design choices for modern uniprocessors must be made early in the design cycle with the aid of a fast “performance model” that is often written in C or C++ [28]. Parallel simulation has been applied to such systems in the past [29],[24], but has been limited to modeling loosely-coupled, distributed shared-memory (DSM) systems and used direct-execution to model the individual processing elements.

Future systems are likely to be composed of multiple processors integrated on a single die known as chip-multiprocessors (CMPs). The CMP has been the subject of many research projects [9],[17],[13],[3] as well as commercial implementations such as IBM’s Power4 [20] and HP’s Mako [19]. CMPs reduce the impact of interconnect delay on clock frequency [1] and reduce design time by making use of repeated, regular structures. A CMP also lends itself to parallel simulation: the processors in a large target CMP can be distributed to different nodes of a host parallel machine and simulated simultaneously. However, the tight coupling of the processors on a CMP with a shared L2 cache and the low latency of such accesses serve to increase the communication demands of a parallel CMP simulator over that of a DSM system.

This paper explores the parallel simulation of a CMP on a distributed host system consisting of commercial off-the-shelf (COTS) workstations connected with a high-speed network [2]. For

portability, the simulator uses the Message Passing Interface (MPI) [23] for inter-node host communication. Several parallelization schemes are evaluated, including a distributed target cache model vs. a centralized scheme, blocking vs. non-blocking accesses to the remote cache, and the use of null-messages vs. barriers to maintain clock synchronization. Microbenchmarks are used to evaluate the performance of each alternative prior to constructing the full simulator. Two popular system-area networks (SANs), the Scalable Coherent Interface (SCI) [31] and Myrinet [4], are also evaluated as the interconnect for the host platform.

In the next section, conventional performance modeling of a CMP and common parallel simulation methods are introduced. Section 3 describes several parallelization approaches considered for the CMP simulation. In Section 4, the alternate approaches are studied through the use of MPI-based microbenchmarks running on the target platform to quantify the tradeoffs associated with each. Section 5 demonstrates the performance of a parallel CMP simulator based on the algorithm selected from the microbenchmarks. Section 6 provides a discussion of related research. Finally, Section 7 contains conclusions and avenues for future research.

## **2 BACKGROUND**

In order to understand the design challenges of a parallel simulator for a CMP, it is important to consider the state-of-the-art in sequential simulations of uniprocessor and multiprocessor systems. Also, there are many approaches that can be used to parallelize a sequential simulation. In this section, both issues are addressed separately.

### **2.1 Sequential Simulation of a Microprocessor**

Traditional, sequential simulation for performance modeling of a microprocessor typically achieves the fastest simulation rate by employing some form of trace-driven simulation. In trace-driven simulation, application code is run through a fast, functional simulator or on an existing machine (perhaps even of a different architecture from that being simulated). A “trace” of

important activities is logged, consisting of a sequence of decoded instructions with specified input and output dependencies, memory accesses, and branch instructions. The trace is then fed into a timing simulator to determine the execution time of the instruction sequence given stalls due to dependencies, cache misses, and branch mispredictions.

Because the trace files have a tendency to grow rather large, in practice they can be generated dynamically at the front-end of the simulator. This approach also allows simulation of mispredicted instructions, an effect that has been shown to cause cache pollution and significantly impact performance [32]. Figure 1 shows a block diagram of a sequential simulator such as SimpleScalar [5].

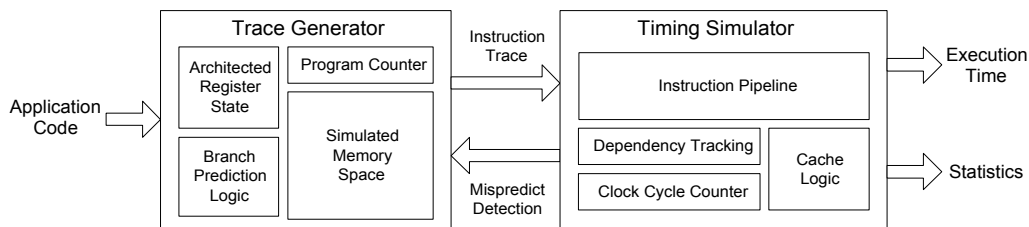


Figure 1. Trace-driven simulation of a uniprocessor using a sequential simulator.

The front-end of the simulator maintains the programmer-visible register state including the program counter and the memory space occupied by the simulated application code. In order to simulate mispredicted instructions, branch prediction is also performed by the trace generator. In this respect, trace generation performs the same duties as the fetch stage in a pipelined microprocessor. The back-end consists of a timing simulator which feeds the trace through the complete pipeline, stalling on register dependencies and cache misses when appropriate. The timing simulator notifies the trace generator when a mispredicted instruction is retired so that a new trace can begin from the correct path.

The instruction trace is generated by executing each instruction as soon as it is fetched. By executing instructions in-order, the outcome of all branches can be immediately known. This

technique allows simulation of key boundary conditions, such as perfect branch prediction. More importantly, it allows the remainder of the timing simulation to know a priori when an instruction is on a mispredicted path, greatly simplifying the recovery of speculative state when a mispredicted branch is resolved.

## 2.2 Sequential Simulation of a CMP

A chip-multiprocessor consists of multiple processors integrated on a single die. Unlike conventional symmetric multiprocessors (SMPs) that share only a common memory space, the processors in a CMP interface at a common cache level such as the L2 cache [25]. An example of a 4-processor CMP is shown in Figure 2.

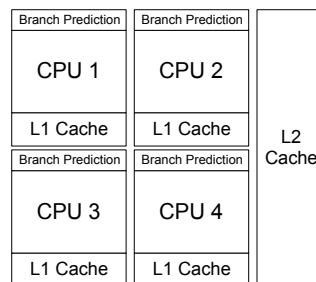


Figure 2. Organization of a CMP.

To enable simulation of such a CMP, the simulation of each processor is interleaved on a clock-cycle basis. In a sequential simulation, the outermost loop simulates a single clock cycle for each CPU in the CMP before incrementing the clock-cycle counter and repeating the loop. This approach enforces time-ordered *consistency* between each processor in the CMP. By incrementing the cycle counter only when all processors have completed execution of that cycle, any external effects which one processor may have on another are maintained.

In the CMP configuration shown in Figure 2, the only external effects which one processor may have on another involve accesses to the shared L2 cache and, by extension, the global memory. Maintaining *coherency* of each processor's view of this shared resource is another major issue in parallel simulation.

For the CMP architectures evaluated in this paper, a Modified-Shared-Invalid (MSI) protocol [18] is used to enforce coherency between the L1 caches and the shared L2 cache. A given cache line in the modified state indicates that only one L1 cache contains that data and both read and write access is permitted. A line in the shared state may be present in multiple L1 caches and is therefore read-only. If a processor desires write access to a shared line, it must first request exclusive access through the L2. The L2 cache will invalidate any other shared lines and grant the access. A line that is not present in any cache is said to be in the invalid state.

### 2.3 Parallel Simulation Techniques

Research in the area of parallel simulation techniques focuses primarily on event-driven simulation. In a parallel, event-driven simulation, each parallel process maintains a queue for locally generated events and separate queues for events generated by each remote process that will influence the local process. Events are processed in-order, with the event having the smallest timestamp of those in any queue, local or remote, processed first.

The challenge in parallel simulation is insuring that events are processed in globally consistent order. That is, an event can only be processed by the destination node when it can be sure that no events with an earlier timestamp will arrive at a later time. One way to ensure such consistency is to wait until all remote input queues contain at least one event before selecting the one with the lowest timestamp.

Such a scheme can easily lead to deadlock situations when there are no events generated for a particular remote process. A common method to avoid this situation is to employ *lookahead* and *null messages* [6]. If a process is deadlocked waiting for an event from one or more remote processes, it will send a null message to the other processes in the system. The null message represents an event that does not require any action but indicates the earliest timestamp for which an actual event may be received from the source node.

This timestamp is computed by taking the current time step of the source node plus a lookahead value. The lookahead value is system-dependent and represents any processing time that would be applied to incoming events before an outgoing event can be generated. An example of null-message-based synchronization applied to a CMP will be provided in Section 3.3.

The use of lookahead and null messages can result in a significant level of generated null-message traffic, particularly if the lookahead is small or if the processes infrequently generate events destined for remote nodes [10]. Another method is to use *barrier synchronization* [14]. In this approach, all processes process events freely up to a certain timestep and then wait for all other processes to reach the same timestep. If the barrier interval is selected based on the lookahead value, each process is guaranteed to have received any remote events that may influence the current barrier-bounded interval prior to the barrier at the beginning of the interval.

### **3 PARALLEL SIMULATION APPROACHES FOR A CMP**

In this section, the parallel simulation techniques presented in the previous section are combined with simulation of multiprocessors to produce several alternatives for parallel simulation of a CMP. For performance reasons, architecture-level simulations of microprocessors are cycle-driven rather than event-driven. Typically, hundreds of events occur each clock cycle in a speculative processor, making the overhead of event queues too costly. Therefore, traditional event-driven parallel simulation is not directly applicable to a CMP.

However, as noted previously, a CMP can be viewed as a collection of independent processors that can only affect one another through the memory hierarchy. If the architecture of Figure 2 is assumed, all such external events will occur through the shared L2 cache. While a single processor may access memory multiple times per clock cycle, modern L1 caches provide hit rates in excess of 95%, so accesses to the L2 cache are relatively infrequent.

In this paper, we adopt a scheme similar to [29] where the processor and L1 caches are modeled with conventional, cycle-driven simulation techniques, but parallel, event-driven simulation is applied between each L1 cache and the shared L2. However, the processor model is more detailed along the lines of those in [5] or [26] than the direct execution used in [29]. Because L2 accesses are infrequent and have a high latency in terms of the simulated timespace, the simulation can be parallelized using a message-passing approach, allowing use of cost-effective and scalable distributed systems as the underlying simulation platform.

The following discussion assumes that the parallel simulation will be divided into one or more threads for concurrent execution. The mapping of threads to nodes on the simulation platform will be considered later.

### 3.1 Centralized vs. Distributed L2 Parallelization

One design choice for parallelization involves the shared L2 cache. The most straightforward approach is to simulate the target's L2 cache in a dedicated thread on the host and each target processor in a separate thread, as shown for a  $p$ -processor CMP in Figure 3a. The dotted lines in this figure indicate that each target processor and associated L1 cache are simulated in a *processor thread* while the L2 and memory subsystems are handled by the *L2 thread*. The L2 thread also models the desired interconnect and contention between processors. It is possible for each processor thread to handle more than one target processor/L1 pair when  $p$  exceeds the number of processor threads,  $N$ , although in this typical example,  $N = p$  and the total number of threads is  $p + 1$ .

The centralized approach has several disadvantages. First, by requiring a thread dedicated to L2 transactions, the parallel efficiency is reduced. Parallel efficiency is defined as the speedup divided by the number of processors in the simulation platform, where speedup is the execution time of a sequential simulation divided by the execution time of the parallel simulation. Since the

speedup is limited to  $N$ , the parallel efficiency is at most  $N / (N+1)$  which is poor for small  $N$ . For large  $N$ , the centralized L2 becomes a bottleneck. To alleviate the first problem, one of the processor threads can take on the dual role of simulating one or more target processor/L1 pairs and the L2 cache. However, this approach limits the performance of the target processor simulations of that thread, requiring the other processor threads to wait on the slower thread.

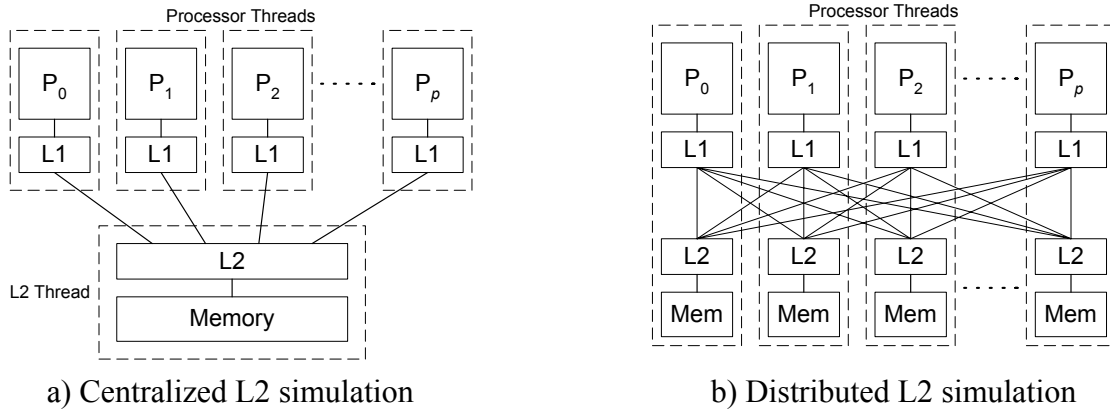


Figure 3. Centralized vs. distributed simulation of the L2 cache.

An alternative approach is to distribute the simulation of the L2 cache across all processor threads and eliminate the L2 thread as in Figure 3b. While this figure assumes  $p = N$ , each host thread can simulate multiple target processor/L1 pairs as well as part of the L2 cache when  $p > N$ . The memory space is interleaved on a cache-block basis with each thread having responsibility for all accesses to a particular bank of memory. If lower-order interleaving is used, the accesses should be relatively evenly distributed, allowing the performance to scale as  $N$  is increased.

One drawback to a distributed L2 simulation is the potentially high cost to simulate contention. If the number of target L2 cache ports must be limited to fewer than  $N$  or if modeling a non-banked cache is required, each portion of the L2 must communicate with the other portions to arbitrate for access. Another drawback involves maintaining a global view of the clock. With a centralized L2 simulation, all-to-one and one-to-all communication can be used to maintain clock synchronization, but a distributed L2 simulation requires costly all-to-all communication.

The performance of either approach depends on the frequency of cache accesses and clock synchronization events in the target system as well as the number of threads tasked to the simulation on the host. These tradeoffs will be evaluated in greater detail in Section 3.3.

### 3.2 Blocking vs. Non-blocking L2 Requests

Another design decision with a large potential effect on parallel simulation performance involves the accesses to the target L2 cache. Traditional, sequential simulators applying a trace-driven approach as in Figure 1 decouple the memory access from cache hit checking. In simulations of CMP architectures, the cache access is integral to the data access due to the need to maintain coherency. Therefore, the trace generator must perform memory transactions through the cache logic. With either centralized or distributed L2 parallelization, a remote access is required on the host to obtain the data for every L2 access in the target.

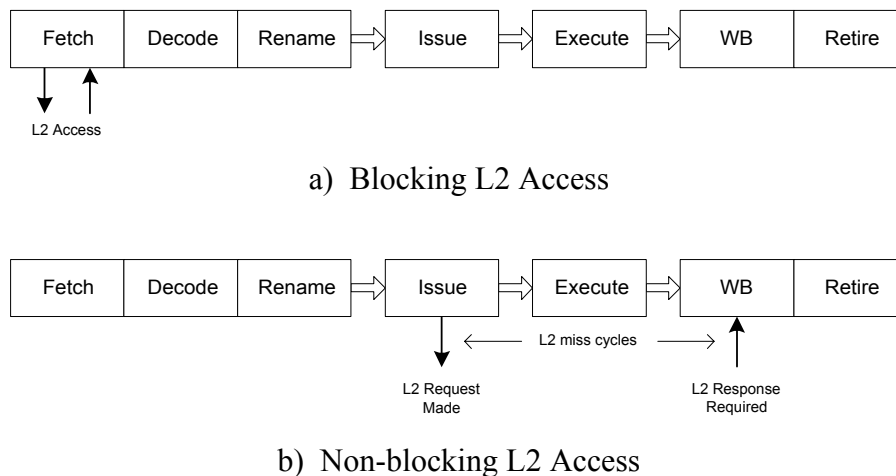


Figure 4. Blocking vs. non-blocking accesses to the L2 cache.

Supporting trace-driven cache accesses requires the accesses to be blocking. That is, each target memory reference is processed in order by the front-end of the simulator to satisfy dependencies before moving on to the next instruction. Figure 4a demonstrates the placement of the L2 accesses of such a simulation on a processor with 7 pipeline stages similar to that which

will be used later in this paper. Trace generation occurs during the fetch stage of each instruction, so the processor thread must block until a response from the remote L2 is received.

As mentioned in Section 2.1, trace-driven simulation greatly simplifies the task of handling mispredicted instructions. However, requiring blocking access to the L2 cache in a parallel simulation is a significant disadvantage. The L2 transactions now initiate a remote transaction and therefore have a much higher latency than in a sequential simulation. While the transaction is outstanding, no other instructions in the simulated processor can be fetched, limiting performance. Another disadvantage to trace-driven simulation is that it introduces inaccuracies in modeling parallel systems due to imprecise timing of events. Such inaccuracies can profoundly impact the outcome of shared accesses such as lock contention. Depending on the desired level of detail in the target platform, such a scheme may be undesirable from a purely functional standpoint.

An alternative approach is to allow non-blocking accesses to the L2 cache as in Figure 4b. This approach more closely resembles the timing and execution of instructions in hardware than the trace-driven approach. The request is performed when all input dependencies are satisfied and the instruction is issued. Since an L1 miss will have a latency of several cycles in the target processor even if it hits in the L2, the simulation can continue for a number of cycles before the response from the L2 is required in the writeback stage. In this manner, the host communication latency for the remote L2 transaction can be hidden.

The major disadvantage of a non-blocking approach is added simulator complexity. Trace-driven simulation is no longer employed and branch mispredictions are not readily identifiable.

### **3.3 Null-message vs. Barrier Synchronization**

Both the null-message and barrier synchronization approaches described in Section 2.3 can be straightforwardly applied to the interface between the target L1 and L2 caches in a parallel CMP

simulator. In both cases, the timestep for synchronization is based on the target L2 cache access time in terms of simulated clock cycles.

For example, in a CMP system where events are generated between the target processors and the shared L2 cache with directory-based coherency, the lookahead interval would be the target L2 cache access time,  $L$ , where “cache access time” refers to the minimum number of cycles from the time an L2 request is issued to the time that the L2 returns the data on a cache hit without any coherency misses for the desired target system. For simplicity, it is assumed that a coherence miss will not generate any traffic (i.e invalidate requests, state downgrades, etc.) before this time, but systems such as that of a snoopy bus could also be modeled if the lookahead is adjusted accordingly. If the L2 has processed all events up to and including time  $t$ , the earliest time that an event destined from the L2 to a processor (e.g. the data returning or a cache line invalidation) can be generated is  $t + L$ . The L2 could send a null message to each host processor thread indicating that it is safe to proceed up to time  $t + L$  without the possibility of receiving an event with an earlier timestamp from the L2.

Similarly, the barrier synchronization approach requires all host threads to wait at a barrier every  $L$  clock cycles. In this manner, requests issued prior to a barrier can only affect events after the barrier. As long as messages are received in-order and the L2 thread processes all outstanding requests before reaching the barrier, consistency is preserved between all host threads.

#### **4 EVALUATION OF SIMULATOR DESIGN ALTERNATIVES**

In order to select the best parallel simulator design from the alternatives presented in Section 3, each design decision must be studied with regard to performance tradeoffs. Developing a simulator for each alternative would be very costly in terms of design time and complexity. Instead, several MPI-based microbenchmarks were developed to enable study of the tradeoffs

associated with the parallel algorithm alternatives when executed on the target system. Key components of the full-fledged simulator are abstracted and provided as parameters to a much simpler parallel program that, in essence, simulates the behavior of the final application. The approach is much simpler than the analytical models of WWT found in [12], trading accuracy for rapid prototyping.

#### 4.1 Evaluation Platform

The purpose of the microbenchmarks is to model the expected communication behavior of the parallelized simulator with reasonable accuracy. In this section, we consider the parameters of the simulator necessary to describe its communication pattern, introduce the design space that will be explored, and describe the experimental platform on which the microbenchmarks and, ultimately, the parallel CMP simulator will be executed.

##### 4.1.1 Simulation Parameters

As in most parallel applications, the performance of a parallel CMP simulation is largely limited by the communication. The communication consists of two components: data-value communication between the separate L1 and shared L2 caches of the target system and clock-cycle synchronization. The latter is largely determined by the parallel algorithm and is a key component for microbenchmark evaluation. The former is a product of the simulated application and the timing simulator. For simplicity, we will abstract the cache-to-cache communication pattern using parameters measured from a sequential version of the simulator.

The simplest model for the cache transactions involves only two parameters: the time required to simulate a single clock cycle for a single processor of the CMP,  $t_c$ , and the average number of cache transactions generated each cycle,  $A$ . We define  $t_c$  in terms of the rate of a uniprocessor simulation in cycles/sec,  $f_c$ . The microbenchmark simulates the full application by simply delaying for  $t_c$  seconds per simulated clock cycle. Since experimental analysis shows  $A$  to

be less than one, the microbenchmarks generate cache requests in a given clock cycle only when a randomly chosen value between zero and one is found to be less than  $A$ .

Additional parameters to the microbenchmarks include the number of processors in the target CMP,  $p$ , and the number of host processor threads in the parallel simulation,  $N$ . The clock synchronization frequency depends on the lookahead, in this case the L2 cache access latency, and is defined as  $L$ . The parameters and their default values, taken from a sequential version of the CMP simulator, are summarized in Table 1.

Table 1. Parameters to microbenchmark simulations.

Parameter	Values	Description
$t_c$	$1 / f_c$ seconds	Simulation latency per clock cycle
$f_c$	100000 cycles/sec	Simulation rate of clock cycles
$A$	0.045 accesses/cycle	L2 cache accesses generated per cycle
$L$	12 cycles	Lookahead for parallel simulation
$p$	2 – 32 processors	Number of processors in target CMP
$N$	1 – 16 threads	Number of processor threads in host

The  $t_c$  value reflects the performance of the sequential simulator. Smaller values indicate a faster sequential simulator, therefore more difficult parallelization due to higher communication-to-computation ratio. The parameter  $A$  is comprised of four access types: L1 instruction-cache misses, L1 data-cache misses, L1 data-cache writebacks, and L1 coherence misses. Including a term for the variance of the  $A$  parameter would enable more accurate workload characterization, but it will be shown that even a simple model without variance has sufficient accuracy.

#### 4.1.2 Algorithm Alternatives

Using the microbenchmark tests, each of the three major design options presented in Section 3 will be explored: centralized vs. distributed L2 cache simulation, blocking vs. non-blocking L2 cache accesses, and null-message vs. barrier clock synchronization. Table 2 illustrates the naming convention we will use in the remainder of this section to distinguish between design alternatives.

Table 2. Parallelization alternatives.

<b>Name</b>	<b>L2 Simulation</b>	<b>L2 Accesses</b>	<b>Clock Synchronization</b>
CB	Centralized	Blocking	-
CBN	Centralized	Blocking	Null-message
CBB	Centralized	Blocking	Barrier
CN	Centralized	Non-blocking	-
CNN	Centralized	Non-blocking	Null-message
CNB	Centralized	Non-blocking	Barrier
DN	Distributed	Non-blocking	-
DNB	Distributed	Non-blocking	Barrier

Note that not all possible combinations are present in the table; when appropriate, the design space has been narrowed through the tradeoff analysis presented later in this section. Also, three incomplete designs are examined: CB, CN, and DN. These configurations do not conduct any form of clock synchronization and therefore could not be used in the full parallel simulator. They are presented as a baseline to establish the impact of clock synchronization communication relative to the data communication.

#### 4.1.3 *Experimental Platform*

In the parallel simulator, the vast majority of messages are either target cache requests, cache responses, or clock synchronization messages. A cache request contains an address field, a source processor identifier, and a timestamp for a total payload of 32 bytes. Messages that are considered cache requests include line fill and upgrade requests from an L1 to the L2 cache or line flush and downgrade requests from the L2 to an L1 cache. Cache responses consist of the same identifying fields as the cache request plus the associated data. Cache lines are 32 bytes, giving cache response messages a total payload of 64 bytes. Cache responses include both line fills from the L2 to a requesting L1 and writebacks from an L1 to the L2. Clock synchronization messages for null-message-based synchronization contain only a 16-byte timestamp, while barrier

synchronization allows use of zero-byte messages. The type of message is identified through the `MPI_TAG` field and therefore does not add to the payload length.

The SCI testbed consists of 9 nodes connected in a 3×3 unidirectional torus. Each node contains two 733-MHz Pentium-III processors using a Serverworks LE chipset and 256 MB of PC133 SDRAM. The SCI adapters are from Dolphin and Scali [30] and feature a link speed of 4.0 Gbps with a 32-bit, 33-MHz PCI interface. Version 2.1.2 of Scali's implementation of MPI, ScaMPI, provides the messaging layer.

The Myrinet testbed consists of 9 nodes connected through an M2L-SW16 16-port switch. Each node features dual 600-MHz Pentium-III processors and an i840 chipset with 256 MB of PC100 SDRAM. The Myrinet adapters have a link speed of 1.28 Gbps with a 64-bit, 66-MHz PCI interface. GMPI 1.2.3 is used for the messaging layer.

The disparity in processor clock frequency between the SCI and Myrinet testbeds is negated by the fact that the per-cycle computational delay,  $t_c$ , is measured in absolute seconds. Therefore, both platforms assume the same uniprocessor simulation rate. Any differences measured in performance will be due to the communication delay alone.

Figure 5 compares the latencies of SCI and Myrinet. The plotted data is for one-half of the round-trip time (RTT) of an MPI message of specified size. SCI provides a much lower latency than Myrinet, particularly for small packet sizes. The 32-byte cache request messages have a latency of about 7.1  $\mu$ s under SCI and almost 19  $\mu$ s with Myrinet. The larger cache response messages require latencies of 10.7  $\mu$ s for SCI and 19.1  $\mu$ s for Myrinet. The small clock synchronization messages have an extremely low latency with SCI at 5.2  $\mu$ s for an 8-byte message in the null-message scheme or 4.5  $\mu$ s for a zero-byte barrier message. Myrinet shows no advantage to very small messages, requiring about 17.6  $\mu$ s for either size.

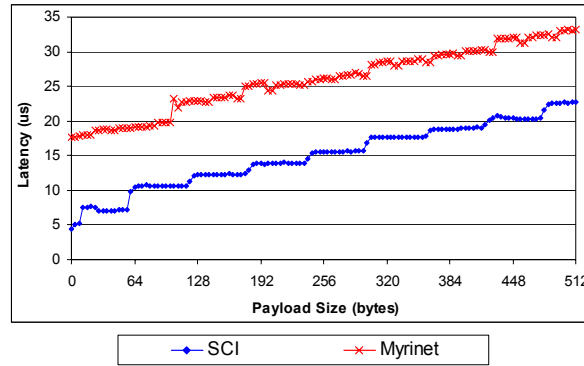


Figure 5. RTT/2 latencies for MPI messages over SCI and Myrinet.

## 4.2 Microbenchmark Results

In this section, the microbenchmarks are used to determine the best design alternative from those discussed above. First, the combinations of L2 simulation, access approach, and clock synchronization protocol will be examined through experiments run on the SCI testbed. Then, the performance of the optimal scheme will be compared against the same on the Myrinet testbed.

In the following experiments, it should be noted that  $N$  refers to the number of host processor threads. For the distributed L2 simulations, there are  $N$  total threads in the parallel simulation. For the centralized L2 simulations, an additional thread is used for the L2 cache requiring  $N + 1$  total threads. When parallel efficiencies are provided, the efficiency is in reference to the total number of threads as appropriate for the L2 simulation.

The mapping of host threads to nodes on the testbed is one-to-one except when  $N = 16$ . Because the SCI testbed is limited to 9 nodes, two processor threads are run on each SMP node, sharing a single SCI interface. When a centralized L2 is required, the L2 thread is run alone on the ninth node. The same configuration is used on the Myrinet testbed.

### 4.2.1 Parallelization Schemes

The first set of microbenchmark experiments makes use of a centralized L2 cache. Figure 6a shows the performance with blocking L2 accesses while Figure 6b demonstrates non-blocking

performance. Both figures compare the speedups obtained with null-message and barrier synchronization. Two baselines are provided: an ideal speedup and the speedup when no clock synchronization is performed. The ideal speedup is defined as the number of processor threads.

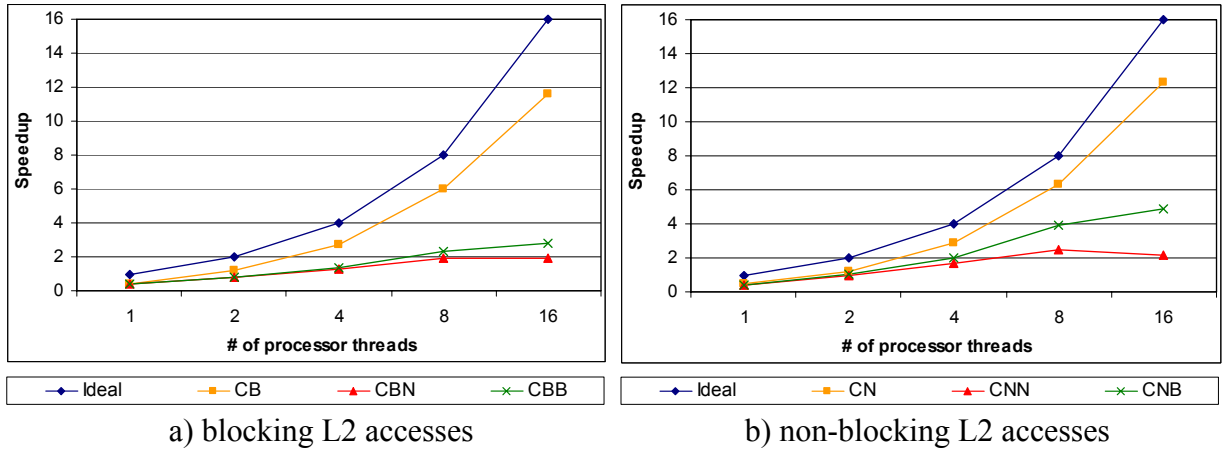


Figure 6. Microbenchmark performance predictions for centralized L2 cache schemes on SCI testbed ( $p = N, f_c = 10^5, A = 0.045, L = 12$ ).

The first trend evidenced in Figure 6 is that the CB and CN approaches both track well with the ideal. Furthermore, there is very little difference between CB and CN because the L2 requests from each processor are very infrequent. However, a large reduction in performance results when clock synchronization is added. This reduction is due to the fact that host processors must wait even when they do not have a target L2 request of their own when other target processors have requests. As  $p$  increases, the probability of at least one target processor performing a request and therefore requiring the others to wait is increased.

With null-message synchronization taken into account, Figure 6 shows a maximum speedup of about 2 for both CBN and CNN schemes with  $N = 16$ . Barrier synchronization fares much better, particularly with non-blocking accesses, showing a speedup of 3 for CBB and 5 for CNB. The benefit of non-blocking accesses in the barrier approach is that all host processors can proceed to the barrier even if there are outstanding requests. With blocking accesses, any processor making an L2 request will arrive at the barrier much later than those that do not make a

cache access. Once again, as  $p$  is increased, the probability of a single processor slowing the rest increases.

The difference between the performance with and without clock synchronization and the difference between the two types of synchronization can be explained by examining the type and number of messages that the L2 thread must handle. Figure 7 shows the number of messages sent and received by the L2 thread per simulated clock cycle. Figure 7a shows the number of cache requests, cache responses, and null messages processed in the CNN scheme, while Figure 7b shows the requests, responses, and barrier messages for CNB. In the simplified microbenchmark approach, the number of messages sent by the L2 equals the number received for each type. Also, the messages are evenly distributed between each of the processor threads.

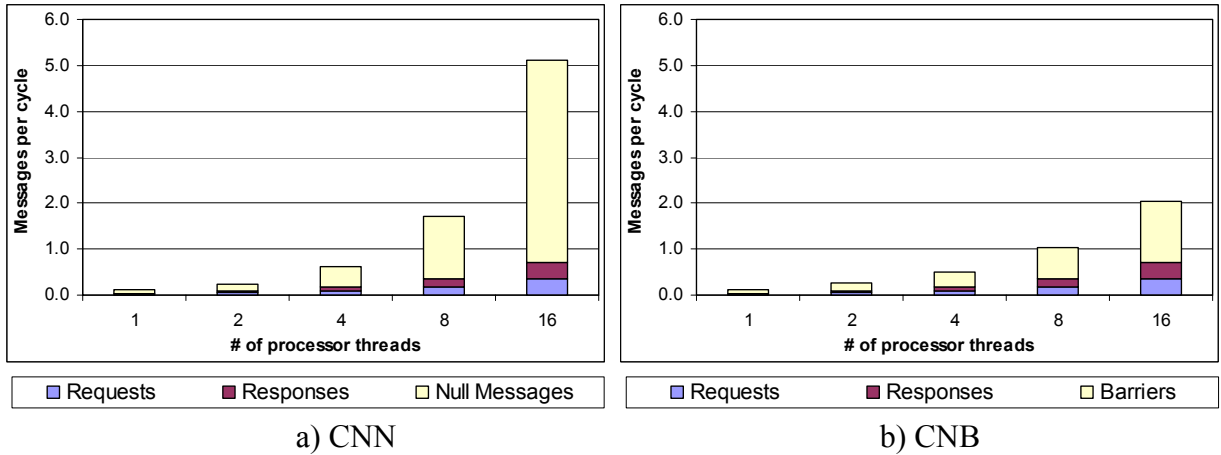


Figure 7. Messages handled per simulated clock cycle by centralized L2 cache with non-blocking accesses on SCI testbed ( $p = N, f_c = 10^5, A = 0.045, L = 12$ ).

As Figure 7 illustrates, the amount of traffic for clock synchronization greatly exceeds the cache traffic, particularly for large values of  $N$ . As described in Section 4.1.3, SCI provides a latency of  $10.7 \mu\text{s}$  for the large cache responses and lower latencies for the other messages. If all messages were of the larger variety, up to 93000 messages could be serviced by the host per second or  $93000 / f_c = 0.93$  messages per target clock cycle. Thus, the performance becomes communication-bound at  $N \geq 8$  where more messages per cycle are required.

Figure 7 also illustrates a large difference between the number of null messages generated in CNN versus the number of barriers in CNB. The number of barriers scales linearly with  $N$  and is also dependent on  $L$ . Since the cache traffic also scales with  $N$ , the barrier messages account for a constant 65% of the total traffic for all values of  $N$  for the combination of  $L$  and  $A$  used.

By contrast, null messages are sent whenever a thread reaches a limit based on the lookahead. To understand why the number of null messages grows faster than  $N$ , consider a simple system with two processor threads, both initially at simulated clock cycle  $t_0$ . The target processors in the first thread do not make any cache accesses and proceed to cycle  $t_0 + L$  before the host sends a null message. A target processor in the second thread makes a cache request in cycle  $t_0 + 1$ . The L2 must wait until the first thread sends its null message before deeming it safe to process the second thread's request. At that time, it can respond to the first thread's null message with permission to proceed to cycle  $t_0 + 1 + L$  (this is because a request may still arrive from the second thread with time stamp  $t_0 + 2$  that may affect the first thread). Therefore, the null message grants the first thread permission to simulate only one additional clock cycle before another null message will be required.

As the number of threads in the system is increased, the probability of at least one thread making a request in any given clock cycle increases. Therefore, due to the effect described above, the null messages in CNN effectively grant permission for fewer clock cycles of simulation for increased  $N$ . Since the number of threads sending null messages increases and the null messages become more frequent, the number of null messages increases faster than  $N$ . For the data in Figure 7, null messages account for only 58% of the total traffic in a 2-thread system. In the 16-thread system, the null messages account for 86% of the total traffic.

The results from Figure 6 indicate that the combination of non-blocking L2 accesses and barrier synchronization perform best. In Figure 8, the distributed L2 approach is compared to the

centralized L2 by using DNB and CNB schemes, respectively. For a baseline comparison without clock synchronization, DN and CN performance is also shown. Figure 8a illustrates the parallel efficiency while Figure 8b shows the speedup.

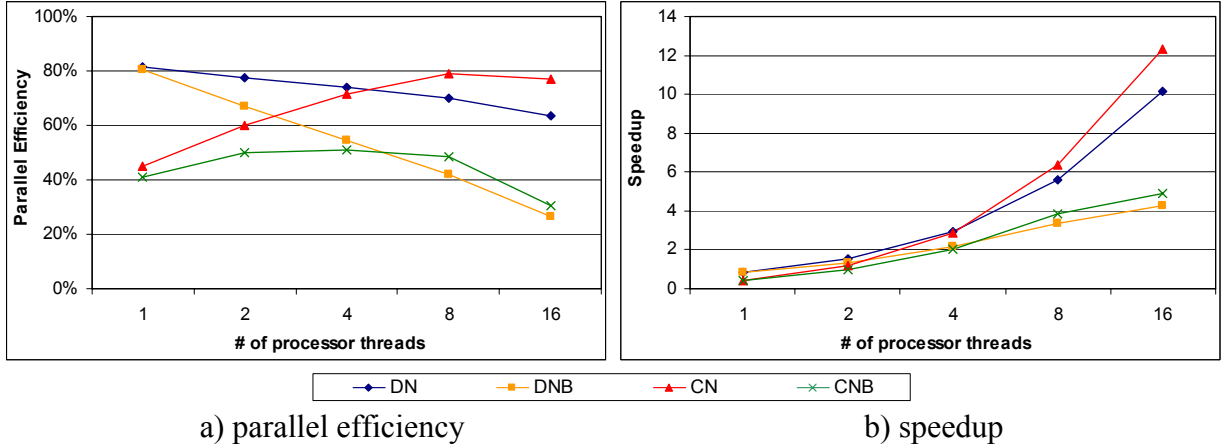


Figure 8. Microbenchmark performance predictions for centralized vs. distributed L2 with non-blocking accesses and barrier synchronization on SCI testbed ( $p = N, f_c = 10^5, A = 0.045, L = 12$ ).

The efficiency of the distributed approach declines as the number of threads is increased, even for the DN scheme that does not perform clock synchronization. This decline in efficiency is due to the fact that the L2 processing task is distributed between all of the processor threads, thereby slowing the target processor simulation. The CN scheme, by contrast, increases in efficiency due to the  $N / (N + 1)$  scaling effect described earlier.

When barrier synchronization is added, the efficiency of the distributed approach decreases rapidly. This effect is due to the need to communicate a barrier message with every other thread in the system in DNB rather than with only the centralized L2 thread in CNB. This all-to-all synchronization requires  $N$  times as many transactions as the centralized approach.

The resulting conclusion is that the distributed L2 is slightly more efficient for a small number of host threads,  $N \leq 4$ . At larger thread counts, the centralized L2 has a slight performance and efficiency advantage. This outcome, coupled with the ability to efficiently

model contention effects allowing more flexibility in exploring target architectures, gives the edge to a centralized L2 design.

#### 4.2.2 Interconnect Selection

With the conclusion that the CNB approach shows the greatest performance potential, we now turn to a comparison of the underlying network for the parallel simulation platform. In Figure 9, the parallel efficiency and speedup of the CNB-based microbenchmarks are shown for both the SCI and Myrinet testbeds. The CN results are also provided as a baseline.

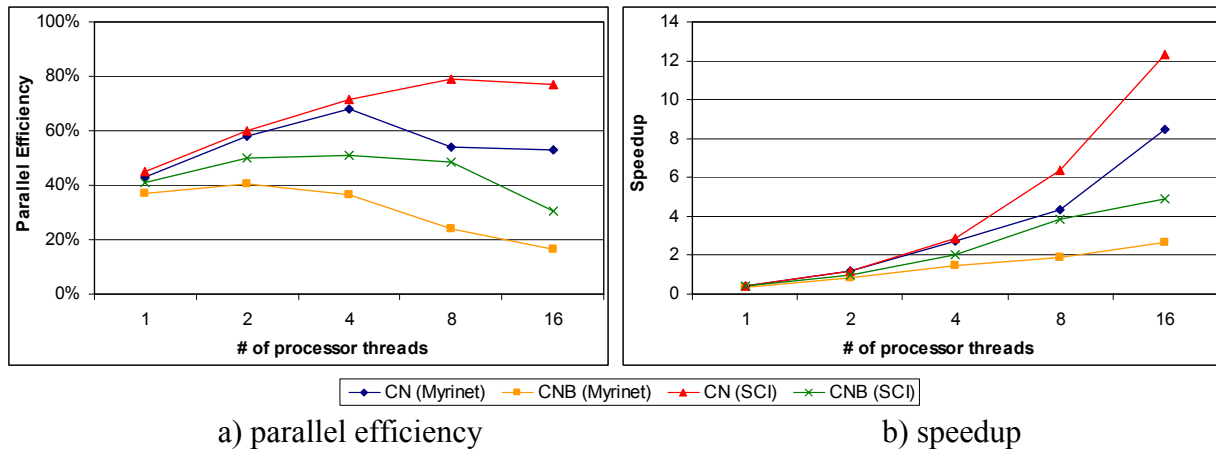


Figure 9. Microbenchmark performance predictions for SCI vs. Myrinet testbed with centralized L2 and non-blocking accesses ( $p = N$ ,  $f_c = 10^5$ ,  $A = 0.045$ ,  $L = 12$ ).

The performance of both interconnects with the CN baseline is similar up until  $N = 8$ , where the Myrinet platform shows a decrease in parallel efficiency. From Figure 7, it can be seen that at this system size, the L2 must process 0.36 request and response transactions per simulated clock cycle. As previously discussed, the 10.7  $\mu\text{s}$  latency of SCI allows the host up to 0.93 transactions per simulated target clock cycle. Myrinet, however, shows a latency of 19.1  $\mu\text{s}$ , allowing only 0.52 host transactions per simulated cycle. Though this appears to be sufficient capacity for the required target cache traffic, in practice the effective latency is increased due to contention when different hosts make multiple requests simultaneously. At 0.36 transactions per clock cycle, the

load on the Myrinet network exceeds 50% and contention is frequent. The SCI performance trails off similarly at  $N = 16$  where 0.72 cache transactions are made per target clock cycle.

When barrier synchronization is added, the performance advantage of SCI becomes even more pronounced. Not only can the SCI network handle a higher load due to its decreased latency, but the majority of the traffic consists of zero-byte barrier messages. As shown in Section 4.1.3, ScaMPI provides a 4.5  $\mu\text{s}$  latency for these messages while Myrinet requires 17.6  $\mu\text{s}$ . The result is a potential speedup at  $N = 16$  of 4.9 using SCI and only 2.7 with Myrinet.

## 5 PARALLEL SIMULATOR RESULTS AND ANALYSIS

Based on the results from the previous section, the SimpleCMP simulator [7], a derivative of SimpleScalar [5], was parallelized using the CNB scheme and subsequently executed on the SCI testbed. In this section, the simulation platform is described, parallel performance is explored, and the measured performance is compared to the microbenchmark predictions from the previous section to demonstrate the validity of the microbenchmark approach.

### 5.1 Simulation Platform

SimpleCMP performs trace-driven simulations of a 32-bit, MIPS-like instruction set [27], complete with register renaming, out-of-order execution, superscalar issue, and detailed cache and branch prediction logic. In addition to the CNB parallelization scheme, the simulator was also modified to more closely resemble the hardware of an Alpha 21264 [21] by replacing the register update unit (RUU) with a reorder buffer and rename logic. Separate issue queues are provided for integer and floating-point instructions, and the pipeline depth matches the design of the 21264. The dual, synchronized register-file of the 21264 was not implemented. The cache subsystem is directory-based with point-to-point links from each L1 to the shared L2. Table 3 shows the key architectural parameters for each individual processor in the simulations.

Table 3. Processor architecture parameters.

Parameter	Value
Fetch/Issue/Retire Width	4 instructions
Rename Registers	128
Integer Issue Queue	64 instructions
Load/Store Issue Queue	32 instructions
Integer ALUs	4
L1 cache ports	2
Branch Predictions/cycle	1
Branch Predictor	<i>Global</i> : 12-bit history, 4k × 2-bit saturating counters <i>Local</i> : 1k × 10-bit history, 1k × 2-bit saturating counters <i>Select</i> : 12-bit global history, 4k × 2-bit saturating counters
Branch Target Buffer	2048 entries, 2-way associative
Return Address Stack	32 entries
Minimum Misprediction Latency	8 cycles
L1 I-Cache	64 kB, 2-way associative, 32-byte lines, 1-cycle latency
L1 D-Cache	64 kB, 2-way associative, 32-byte lines, 1-cycle latency
Unified L2 Cache	8 MB, 4-way associative, 32-byte lines, 12-cycle latency
Memory Access Latency	100 cycles, 8 bytes/cycle

Table 4. Selected SPLASH-2 benchmark components.

Benchmark	Input Dataset	Instructions	8-processor Parameters	
			$f_c$	$A$
LU	256x256 matrix, 16k blocks	66 M	131k	0.0147
Ocean	258x258 grids, 4 time-steps	330 M	102k	0.0463
Radix	256k integers, radix = 1024	16 M	131k	0.0355
Raytrace	teapot.env	282 M	98k	0.0103
Average	-	174 M	107k	0.0267

In order to evaluate the parallel simulator under a variety of workload conditions, a representative subset of the SPLASH-2 benchmarks [34] was selected. SPLASH-2 provides a suite of shared-memory benchmarks for parallel systems and includes several kernel and application components. Table 4 lists the four benchmarks used, two kernels and two applications: *LU*, *ocean*, *radix*, and *raytrace*. The table also provides the measured values for  $f_c$  and  $A$  when executed on an 8-processor CMP using the sequential version of SimpleCMP. These values vary slightly for different degrees of parallelism, with  $f_c$  generally decreasing and  $A$  increasing with higher target processor counts.

All statistics are gathered only from the parallel portion of the benchmarks, neglecting any sequential start-up code and generation of working-set data. This common practice assumes that, with realistic workloads, the initialization code is insignificant compared to the overall execution time. In practice, the non-parallel part can be sped up considerably by using only the front-end of the trace generator and not the timing simulator, or even eliminated entirely through checkpointing. Table 4 provides the approximate number of instructions that comprised the parallel portion of each benchmark.

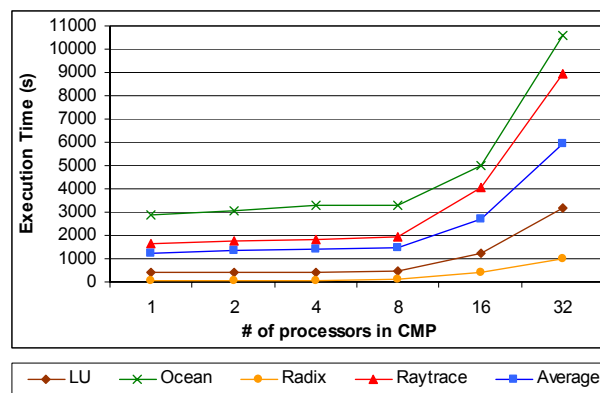


Figure 10. Sequential execution time for SPLASH-2 benchmarks.

As a baseline for comparison, Figure 10 shows the execution time for the sequential version of SimpleCMP for each benchmark and the overall average measured on one of the SCI testbed nodes. The execution time remains relatively constant up to an 8-processor CMP despite an increase in target complexity because fewer cycles are necessary due to the benchmarks' near-linear speedup. A sharp increase for 16- and 32-processor CMPs occurs as the simulator working set exceeds the 256 kB L2 cache of the host systems for large target processor counts.

The sequential simulation scheme described in Section 2.2 results in highly reduced locality at large processor counts due to the sequential iteration over each processor for every simulated clock cycle. In addition, the capacity required to hold all frequently used data increases. The key data elements for each iteration are the decoded instructions which are 236 bytes in length. Each

processor can have up to 128 such instructions in-flight, for a total of 29.5 kB of data. With eight processors in the simulated CMP, 236 kB is required to hold just the decoded instructions. In addition, the L1 caches and branch predictors are also frequently accessed.

A 16-processor CMP simulation has twice as large a working set as an 8-processor CMP and therefore much poorer data locality. It is important to note that even the 32-processor CMP simulation fits in the 256 MB main memory of one of the testbed nodes, so swapping to disk is not an issue. Such a performance limitation would be relatively easy to fix by increasing the memory capacity, but cache sizes are much more difficult and expensive to increase.

## 5.2 Parallel Performance

The SPLASH-2 components from Table 4 were simulated on the parallel CMP simulator for a variety of target CMP sizes and processor thread counts. Figure 11 shows the speedup obtained over the sequential version of the simulator. As before, the horizontal axis refers to the number of processor threads and does not include the L2 thread.

Several trends are apparent from the results in Figure 11. First, for CMPs of 8 processors and less, the speedup is relatively modest, reaching almost 4 for 8 threads. There is a slightly increased speedup for an 8-processor CMP over a 4-processor CMP even when both systems are simulated with the same number of host threads. This increase is due to the fact that the synchronization overhead is a lower percentage of the total communication time when each host thread simulates more than one target processor. In effect, twice the number of instructions and twice the number of L2 cache requests can be simulated per barrier synchronization.

For 16-processor CMPs, the speedup shows a somewhat larger improvement. At 32-processors, the performance is markedly improved. This effect can be explained by the improved cache locality in the parallel simulation. As illustrated in Figure 10, cache locality becomes problematic in a uniprocessor simulation of CMPs of this size. In the parallel approach, however,

a 16-thread simulation requires only one or two target processors per host thread for a 16- or 32-processor CMP, respectively. The parallelization effectively scales the size of the host L2 cache with the number of threads, improving the performance beyond the simple computational overlap.

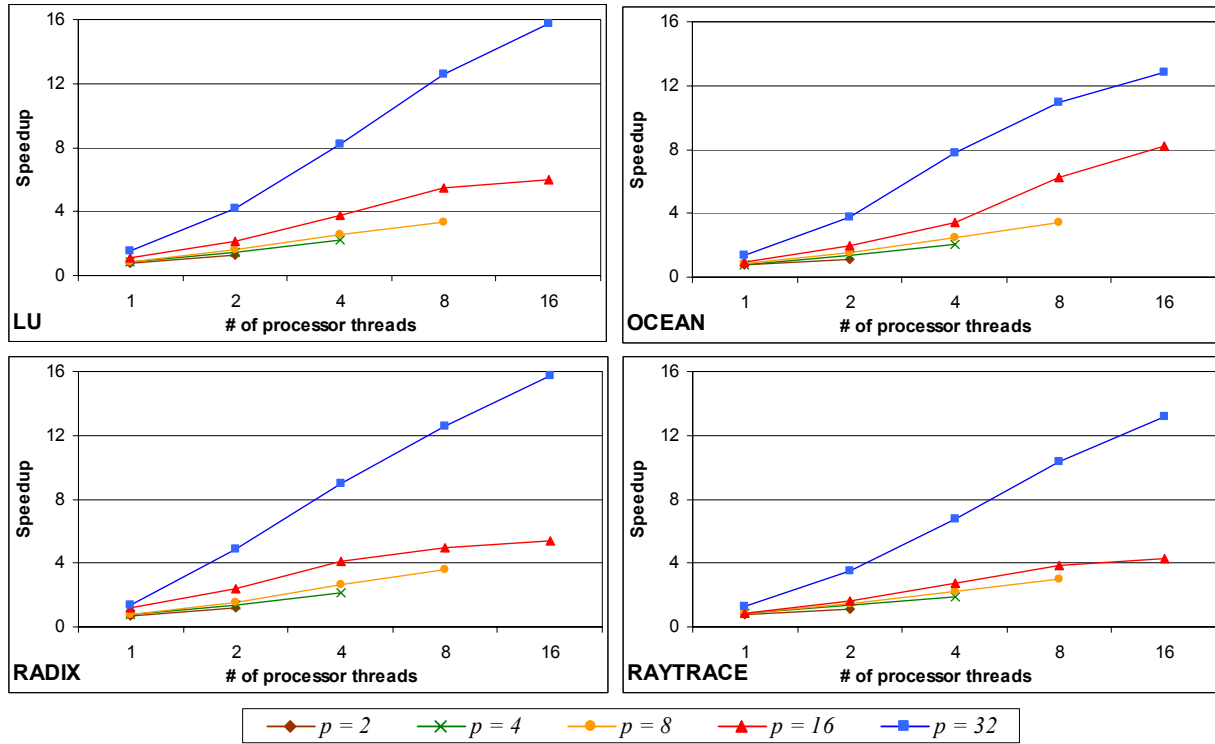


Figure 11. Speedup for selected components of SPLASH-2 for target CMPs of varying size.

Figure 12 better illustrates this effect by showing the parallel efficiency (relative to the total number of threads, including the L2 thread). For system sizes of between 2 and 8 threads, parallel simulation of the 32-processor CMP shows a superlinear speedup due to the increased cache hit rate. At 16 threads, the reduced efficiency predicted by the microbenchmarks and illustrated in Figure 8 reduces the performance to sublinear levels.

Interestingly, the improved cache locality shows benefit even in the simplest case of a single processor thread plus the centralized cache node. The theoretical maximum efficiency is 0.5 in such a system, but the 32-processor CMP simulation exceeds this efficiency for all benchmarks while the 16-processor CMP simulation does so for the *LU* and *radix* benchmarks.

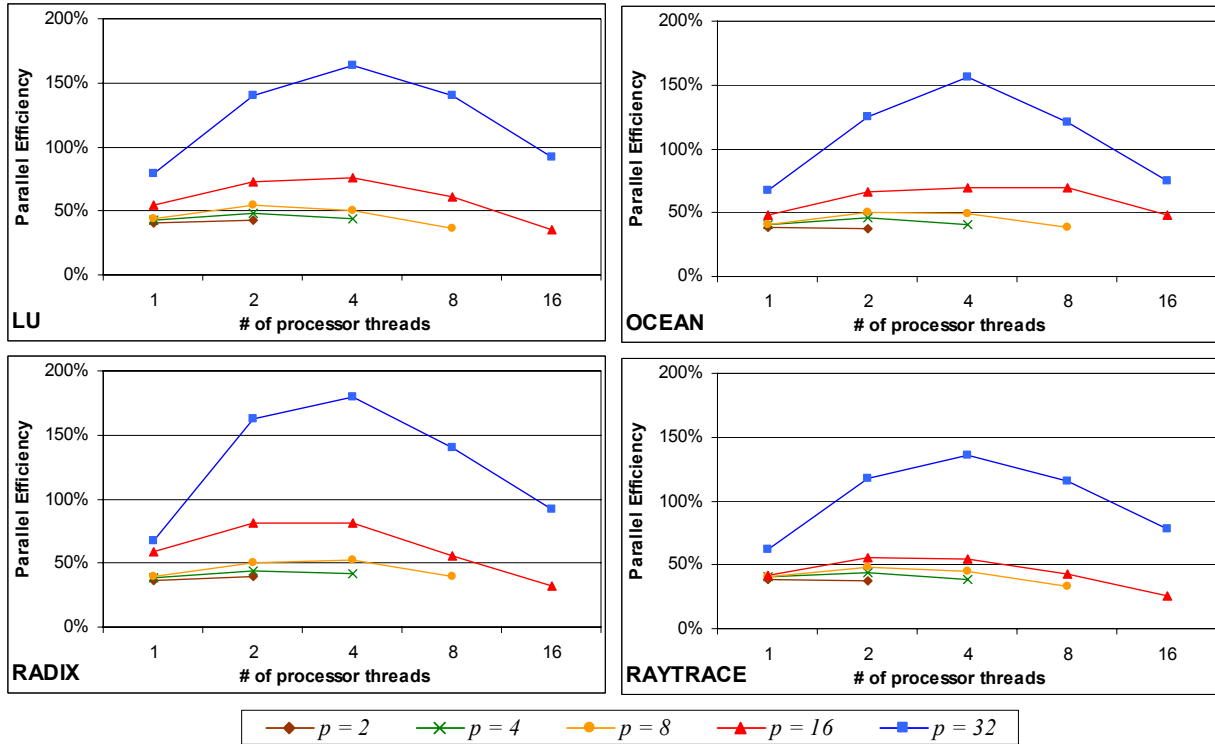


Figure 12. Parallel efficiency for selected components of SPLASH-2 for CMPs of varying size.

### 5.3 Comparison to Microbenchmarks

In order to assess the accuracy of the microbenchmarks in predicting the parallel simulator performance, microbenchmarks were run for each  $f_c$  and  $A$  value in Table 4 corresponding to one of the SPLASH-2 benchmarks and the results averaged. Due to the cache effects with CMP configurations of greater than 8 target processors for which the microbenchmarks do not account, comparisons are made for an 8-processor CMP with host thread counts from  $N = 1$  to  $N = 8$ . The average result for the CNB-based microbenchmarks and the measured performance on the fully implemented simulator are shown in Figure 13.

The CNB microbenchmark does not track particularly well with the actual parallel simulator performance. At  $N = 1$ , the difference is only 9.8%, but the disparity increases to 25% for  $N = 8$ . The disparity was found to be largely due to a neglected effect that has a non-negligible impact on the overall performance. Namely, target cache requests from the L1 caches to the L2 cache can

cause the L2 to require a writeback of dirty data from a second L1 in order to satisfy the first request. In the implementation used in the parallel simulator, the L2 blocks on such a request until the dirty line is returned by the second L1. Though the writeback transaction is included in the  $A$  parameter, the L2 cache blocking effect was not accounted for in the CNB microbenchmark.

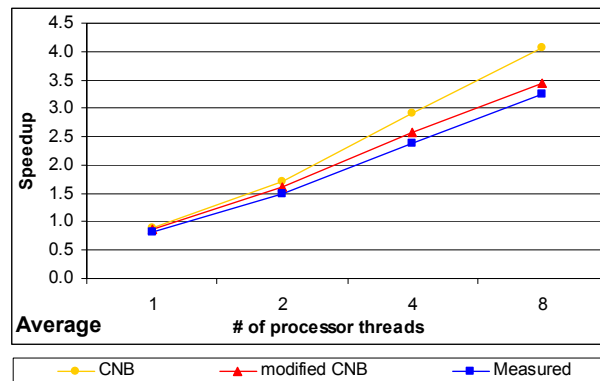


Figure 13. Comparison of microbenchmark prediction and actual parallel simulator performance.

The CNB microbenchmark was modified to accept another parameter: the probability that an L1 request will require a writeback request and concomitantly block in the L2. For the benchmarks studied, this probability ranged from 0.3% to 14% with an average of 9.3%. Using the measured probabilities for each SPLASH-2 component, the modified microbenchmarks were executed and the results plotted in Figure 13.

As illustrated in the figure, the modified CNB microbenchmark predicts the performance of the parallel simulator with much greater accuracy: within 10% of the measured results for the system sizes shown. The remaining difference can be attributed to the fact that the actual execution of the benchmarks frequently results in bursty L2 accesses, often synchronized with barriers or locks in the parallel code, reducing the performance as contention increases the latency for each transaction. By contrast, the microbenchmarks yield a more uniform distribution of cache accesses. Including a variance term in the microbenchmarks could account for this effect, but the results are already within acceptable limits.

## 6 RELATED RESEARCH

Much existing research exists in the field of parallel simulation of multiprocessors. The Wisconsin Wind Tunnel simulates a parallel, CC-NUMA system on various parallel systems including a Myrinet-connected cluster of workstations [29],[24]. Synchronized Active Messages provide the messaging layer rather than MPI as in our study. Also, WWT makes use of direct-execution simulation with each parallel process running on a separate CPU, capturing only the memory accesses and sending them through a sequential timing simulation of the memory hierarchy. This approach does not offer flexibility in the processor model and is unable to model effects such as issue widths, speculative memory accesses, and out-of-order execution. Analytical modeling has been used to approximate the performance of WWT for a variety of system sizes [12]. Interestingly, [12] shows an optimal cost/performance ratio around 16 to 32 processors in the target system due largely to memory costs, mirroring the cache effects noted in our study.

The Integrated Simulation Environment (ISE) takes a similar approach for simulation of MPI-based parallel programs [16]. In ISE, MPI code executes natively on distributed nodes in a cluster with the MPI function calls being intercepted and sent to a centralized, sequential simulation of the interconnect. This “software-in-the-loop” approach allows flexibility in selecting the interconnect type and topology but is limited in performance by the relatively slow interconnect simulation.

In [15], parallel simulation is applied to a parallel digital signal processor (DSP) system. In this study, the parallel programming language Linda is applied in a clustered environment to achieve speedup on a multi-DSP simulation despite the use of low-performance, 10 Mbps Ethernet as the interconnect.

The RSIM [26] simulation environment provides great flexibility in the configuration of the individual processors in a simulated multiprocessor, but the simulation itself runs only on a

uniprocessor. Rather than parallelizing the simulator, DirectRSIM [11] provides a performance improvement by adapting RSIM to be trace-driven instead of execution-driven. Using an approach similar to RSIM, the MINT [33] multiprocessor simulator has even been modified to simulate a CMP [22], but neither approach involves parallel simulation.

## 7 CONCLUSIONS

Simulation of parallel systems is an important tool in evaluating design alternatives. As the complexity of such systems increases down to the chip level with the advent of CMP-based systems, the simulation overhead becomes increasingly prohibitive. In this paper, an MPI-based parallel simulation environment was presented to reduce the execution time of performance-level CMP simulations running on a cluster of workstations connected by a high-speed interconnect.

The parallel simulation is a natural extension of conventional, sequential simulation of a uniprocessor combined with traditional, event-driven parallel simulation techniques. The proposed shared-L2 design of future target CMP architectures requires a tighter coupling of shared communications in the host platform than traditional DSM-based target systems, making the selection of parallel algorithm and a high-performance network very important for the host.

Several design alternatives were considered, including centralized vs. distributed simulation of the target L2 cache, blocking vs. non-blocking L2 accesses, and null-message vs. barrier clock synchronization. Through MPI-based microbenchmarks, the optimal combination was found to be a CNB scheme combining centralized L2, non-blocking accesses, and barrier synchronization with a predicted speedup of around 5 for 16 threads. The microbenchmarks also demonstrate that the low-latency communication afforded by SCI makes it a more suitable interconnect for the host platform than Myrinet.

Performance results for a fully implemented parallel simulator were presented for a variety of workloads. Due to increased cache capacity of the parallel platform, the results show higher-than-predicted speedups of between 12 and 16 when simulating a large CMP architecture with 16 processor threads on 9 dual-CPU cluster nodes. The parallel simulation results are shown to differ somewhat from the microbenchmark predictions even without the cache effect, but the difference is largely accounted for with a slight modification to the microbenchmarks.

Future work in this area could pursue several avenues. Larger system sizes, in terms of the numbers of simulated CMP processors in the target or processor threads and cluster nodes in the host, could be studied. CMP processor counts were restricted to 32 in this study due to the fact that limitations in VLSI technology will likely yield CMPs of this size or smaller in the near future, but further advances or simpler processor designs may allow chip-level architectures with hundreds of processors. The cluster node counts and, by extension, processor threads in this study were limited by the hardware available.

Further study could also be directed at more efficient parallel simulation algorithms. For example, a more aggressive clock synchronization scheme might speculatively execute instructions past the null-message or barrier clock cycle, rolling back the computation if an earlier-stamped message is received. The speculative nature of the processor pipeline being simulated would facilitate such an approach with little extra overhead.

Parallel simulation should increase in importance as parallel systems become larger and more complex. Fortunately, the existence of such systems will also enable parallel simulators to study the next-generation design alternatives with greater speed and efficiency. As the simulation application itself becomes more complex, techniques such as microbenchmark-based evaluation of design alternatives will become increasingly valuable in assessing the most effective parallelization technique for a given system.

## 8 ACKNOWLEDGMENTS

This work was funded in part by the Department of Defense and by an NSF Graduate Fellowship (Chidester). Support was also provided through equipment donations from Nortel Networks, Intel Corporation, Dolphin Interconnect LLC, and Scali AS. We also wish to thank the anonymous reviewers for their valuable feedback.

## REFERENCES

- [1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27<sup>th</sup> International Conference on Computer Architecture*, pp. 248-259, June 2000.
- [2] T. Anderson, D. Culler, and D. Patterson. A case for NOW. *IEEE Micro*, 15(1), pp. 54-64, Feb. 1995.
- [3] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27<sup>th</sup> Annual International Symposium on Computer Architecture*, June 2000.
- [4] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1), pp. 26-36, Jan. 1995.
- [5] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. Technical Report TR-1342, University of Wisconsin-Madison Computer Sciences Department, June 1997.
- [6] K. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5(5), pp. 440-452, 1979.
- [7] M. Chidester, A. George, and M. Radlinski. Multiple-path execution for chip-multiprocessors. Technical Report, HCS Research Lab, Department of Electrical and Computer Engineering, University of Florida, Apr. 2001.
- [8] D. Culler and J. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.
- [9] L. Codrescu, D. Wills, and J. Meindl. Architecture of the Atlas chip-multiprocessor: dynamically parallelizing irregular applications. *IEEE Transactions on Computers*, 50(1), pp. 67-82, Jan. 2001.
- [10] R. DeVries. Reducing null messages in Misra's distributed discrete event simulation method. *IEEE Transactions on Software Engineering*, 16(1), pp. 82-91, Jan. 1990.
- [11] M. Durbhakula, V. Pai, and S. Adve. Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pp. 23-32, Jan. 1999.

- [12] B. Falsafi and D. Wood. Modeling cost/performance of a parallel computer simulator. *ACM Transactions on Modeling and Computer Simulation*, 7(1), pp. 104-130, Jan. 1997.
- [13] M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee. The M-Machine Multicomputer. *International Journal of Parallel Programming*, 23(3), pp. 183-212, June 1997.
- [14] R. Fujimoto. *Parallel and Distributed Simulation Systems*. John Wiley & Sons, Inc., 2000.
- [15] A. George and S. Cook. Distributed simulation of parallel DSP architectures on workstation clusters. *Simulation*, 67(2), pp. 94-105, Aug. 1996.
- [16] A. George, R. Fogarty, J. Markwell, and M. Miars. An Integrated Simulation Environment for parallel and distributed system prototyping. *Simulation*, 75(5), pp. 283-294, May 1999.
- [17] L. Hammond, B. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9), pp. 79-85, Sept. 1997.
- [18] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.
- [19] D. Johnson. HP's Mako processor. *Microprocessor Forum 2001*, Oct. 2001.
- [20] J. Kahle. Power4: A dual-CPU processor chip. *Microprocessor Forum 1999*, Oct. 1999.
- [21] R. Kessler, E. McLellan, and D. Webb. The Alpha 21264 microprocessor architecture. In *Proceedings of the 1998 International Conference on Computer Design: VLSI in Computers and Processors*, pp. 250-259, June 1998.
- [22] V. Krishnan and J. Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip multiprocessor. In *Proceedings of the 1998 International Conference on Supercomputing*, pp. 85-92, June 1998.
- [23] *MPI: A Message-Passing Interface Standard*. Message-Passing Interface Forum, [www.mpi-forum.org](http://www.mpi-forum.org), 1994.
- [24] S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, M. Hill, D. Wood, S. Huss-Lederman, and J. Larus. Wisconsin Wind Tunnel II: A fast and portable parallel architecture simulator. *IEEE Concurrency*, 8(4), pp. 12-20, Oct. 2000.
- [25] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, K. Chung. The case for a single-chip multiprocessor. In *Proceedings of the 7<sup>th</sup> International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2-11, Oct. 1996.
- [26] V. Pai, P. Ranganathan, and S. Adve. *RSIM Reference Manual version 1.0*, Technical Report 9705, Department of Electrical and Computer Engineering, Rice University, Aug. 1997.
- [27] C. Price. *MIPS IV Instruction Set, Revision 3.1*. MIPS Technologies, Inc., Mountain View, CA, Jan. 1995.
- [28] M. Reilly and J. Edmondson. Performance simulation of an Alpha microprocessor. *IEEE Computer*, 31(5), pp. 50-58, May 1998.

- [29] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 48-60, May 1993.
- [30] Scali System Guide version 2.0, white paper. Scali Computer AS, [www.scali.com](http://www.scali.com), 2000.
- [31] *Scalable Coherent Interface: ANSI/IEEE Standard 1596-1992*. Piscataway, NJ: IEEE Service Center, 1993.
- [32] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark. Branch prediction, instruction-window size, and cache size: performance tradeoffs and simulation techniques. *IEEE Transactions on Computers*, 48(11), pp. 1260-1281, Nov. 1999.
- [33] J. Veenstra and R. Fowler. *MINT Tutorial and User Manual*, Technical Report 452, Department of Computer Science, University of Rochester, Aug. 1994.
- [34] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22<sup>nd</sup> International Symposium on Computer Architecture*, pp. 24-36, June 1995.