Appears in: "Proceedings of the 1993 ACM SIGMETRICS Conference," May 1993.
Reprinted by permission of ACM.

The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers*

Steven K. Reinhardt, Mark D. Hill, James R. Larus,
Alvin R. Lebeck, James C. Lewis, and David A. Wood
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA
wwt@cs.wisc.edu

Abstract

We have developed a new technique for evaluating cache coherent, shared-memory computers. The Wisconsin Wind Tunnel (WWT) runs a parallel shared-memory program on a parallel computer (CM-5) and uses execution-driven, distributed, discrete-event simulation to accurately calculate program execution time. WWT is a virtual prototype that exploits similarities between the system under design (the target) and an existing evaluation platform (the host). The host directly executes all target program instructions and memory references that hit in the target cache. WWT's shared memory uses the CM-5 memory's error-correcting code (ECC) as valid bits for a fine-grained extension of shared virtual memory. Only memory references that miss in the target cache trap to WWT, which simulates a cache-coherence protocol. WWT correctly interleaves target machine events and calculates target program execution time. WWT runs on parallel computers with greater speed and memory capacity than uniprocessors. WWT's simulation time decreases as target system size increases for fixed-size problems and holds roughly constant as the target system and problem scale.

*This work is supported in part by NSF PYI Awards CCR-9157366 and MIPS-8957278, NSF Grant CCR-9101035, Univ. of Wisconsin Graduate School Grant, Wisconsin Alumni Research Foundation Fellowship and donations from A.T.&T. Bell Laboratories and Digital Equipment Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the Univ. of Wisconsin Graduate School.

1 Introduction

The architecture of a parallel computer is a specification of an interface between software and hardware. The complex interactions in a system of this type are best studied by running and measuring real applications. Paper studies, analytic models, and simulations of abstract workloads can uncover architectural flaws. Nevertheless, running real applications and system software can expose software and hardware problems that other techniques cannot find and encourages improvements through successive refinement [9].

The two well-known methods for evaluating parallel computer architectures with real workloads are execution-driven simulation and hardware prototyping. Call the computer under study the *target machine* and the existing computer used to perform the evaluation the *host machine*. With execution-driven simulation, researchers write a host machine program, called a *simulator*, that interprets a target machine program, mimics the operation of the target machine, and estimates its performance [9]. By contrast, with hardware prototyping, researchers build a copy (or a few copies) of the target system, execute target programs, and measure their performance [20].

Execution-driven simulation and hardware prototyping offer largely complementary advantages and disadvantages. Simulators can be constructed relatively quickly, can be modified easily, and can compare radically different alternatives. However, simulators often are too slow and run on machines without enough memory to simulate realistic workloads and system parameters. Parallel target machines exacerbate the deficiencies of execution-driven simulation, since a single host must simulate many processors. A uniprocessor workstation host with 128 megabytes of memory, for example, cannot simulate even 64 processors with 4 megabytes each. Hardware prototypes, on the other hand, run fast enough to execute real work-

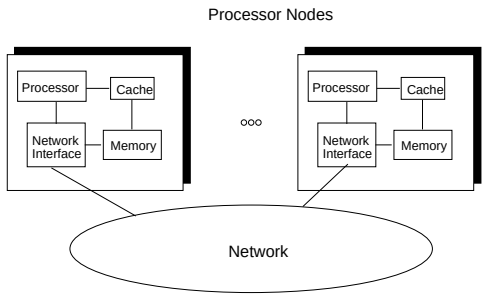

Figure 1: Organization of a parallel computer. Many parallel computers are composed of processor nodes, each of which contains a CPU, cache, and local memory. The nodes are connected by an interconnection network (e.g., a 2D mesh or fat tree).

loads, but are expensive, require years to construct, and are difficult to modify.

Our evaluation method combines the advantages and mitigates the disadvantages of execution-driven simulation and hardware prototyping by exploiting the commonality in parallel computers. The common hardware base (see Figure 1) consists of a collection of processing nodes connected by a fast network. Each node contains a processor, memory module, network interface, and often a cache. Of course, computers differ in network topologies; processor architectures; and, most important, in the primitives built on the base hardware. For example, although interconnection networks exchange messages, some machines expose this facility (e.g., Intel Paragon) while others use it to construct a shared-memory abstraction (e.g., Kendall Square KSR-1).

A new approach to exploiting this commonality underlies our system for evaluating cache-coherent, shared-memory computers. The Wisconsin Wind Tunnel (WWT) runs a parallel shared-memory program on a parallel message-passing computer (a CM-5) and concurrently calculates the program's execution time on the proposed system. We call WWT a *virtual prototype* because it exploits similarities between the system under design (the target) and an existing evaluation platform (the host). In WWT, the host directly executes all target instructions and memory references that hit in the target cache. *Direct execution* means that the host hardware executes target program operations—for example, a floating-point multiply instruction runs as a floating-point multiply [5, 6, 9, 11]. Simulation is only necessary for target operations that a host machine does not support. Direct execution runs orders of magnitude faster than software simulation [9].

A unique aspect of WWT is that it directly executes memory references that hit in a target cache. Other direct-execution simulators test for a cache hit before every memory reference. WWT uses the CM-5 memory's error-correcting code (ECC) as valid bits to build a fine-grained extension of Li's shared virtual memory [21]. Only the memory references that miss in the target cache trap to WWT, which simulates a cache coherence protocol by explicitly sending messages.

WWT, like uniprocessor execution-driven simulators, correctly interleaves target machine events and calculates target program execution times. Because WWT runs on a parallel computer, it requires two new techniques to integrate distributed simulation on a parallel computer with direct execution of target processes. First, WWT manages target node interactions by dividing execution into lock-step quanta to ensure all events originating on a remote node that affect a node in the current quantum are known at the beginning of the quantum. Second, WWT orders all events on a node for the current quantum and directly executes the process up to its next event.

The key contributions of this work are the development and demonstration of the two techniques necessary to build virtual prototypes of parallel, cache-coherent, shared-memory computers. The first technique is fine-grained shared virtual memory, which enables shared-memory programs to run efficiently on non-shared-memory computers (Section 3). The second technique is the integration of direct execution and distributed, discrete-event simulation, which together efficiently run a parallel shared-memory program and accurately compute its execution time (Section 4). Section 5 compares WWT with a uniprocessor simulation system (Stanford's Tango/Dixie) simulating the Stanford DASH multiprocessor. Section 6 describes related work, Section 7 describes extensions and future work. Section 8 presents our conclusions.

2 Background

This section provides background by briefly reviewing machines that we want to simulate (Section 2.1), important characteristics of the host machine (Section 2.2), and simulation techniques (Section 2.3).

2.1 Cache-Coherent, Shared-Memory Target Machines

Figure 1 illustrates the essential features of a target machine. It has N processors, where N currently cannot exceed the number of host processors (however,

see Section 7). To permit high-bandwidth access, memory is divided into N memory modules, each of which is physically adjacent to a processor. A processor, memory module, cache, and network interface form a processing node. Nodes are connected by an interconnection network that delivers point-to-point messages.

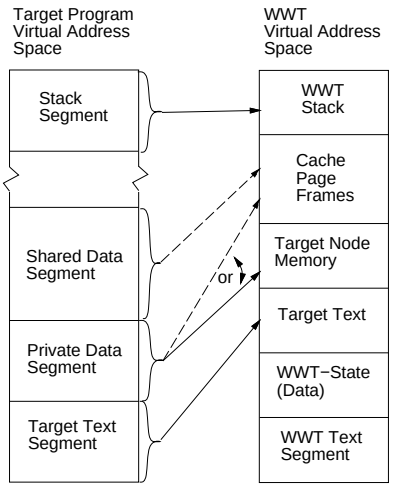

Figure 2: The target program's address space is shown on the left. The virtual prototype's (WWT) address space (right) is a separate SPARC context. The arrows show how WWT maps target data into its address space. Dashed lines indicate that a mapping exists only for pages containing cached blocks. The mappings are logical and implemented with virtual address aliases to common physical pages.

The target machine supports shared memory by allowing all processors to address a single virtual address space. The current target programs are single-program-multiple-data (SPMD) shared-memory programs that run one process per node. Each process's virtual address space contains four segments—text (code), private data, shared data, and stack—as illustrated on the left side of Figure 2. All processes identically map the shared-data segment. Other segments are mapped to physically local memory, so distinct processes access different private locations. Shared data is interleaved by placing successive pages ($\geq$ 4K bytes) on successive nodes.

To reduce average memory latency, each cache holds blocks ($\geq$ 32 bytes) recently referenced by its processor. A processor node handles a cache miss by sending a message to the referenced block's home node and receiving the block's contents in a reply message. Processors use a directory protocol to keep cache and memory blocks coherent [1]. Typically, a directory is distributed so a directory entry and memory block reside on the same node. The directory protocol determines the contents of a directory entry and the messages used to maintain coherence.

We assume that the execution time for each instruction is fixed. Instruction fetches and stack references require no cycles beyond the basic instruction time. Other memory locations are cached in a node's cache. A cache hit requires no additional cycles, while a cache miss invokes a coherence protocol that sends messages, accesses a directory entry, etc. All messages and cache or directory transitions have a cost. A cache or directory processes messages in first-come-first-serve (FCFS) order. Queuing delay is included in the cache miss cost. Network topology and contention are ignored, and the latency of all messages is fixed at T cycles.

2.2 TMC CM-5 Host

The host platform for WWT is a Thinking Machines CM-5 [33]. The architecture supports from 32 to 16,384 processing nodes. Each node contains a SPARC processor, cache, memory management unit (MMU), custom vector/memory controller units, up to 128 MB of memory, and a custom network interface. Nodes are connected by two user-accessible networks. The data network allows a program to send short (five-word) point-to-point messages, but does not guarantee order of arrival. The control network performs fast barriers, broadcasts, and parallel prefix computations. WWT does not use the vector units.

2.3 Distributed, Discrete-Event Simulation

In *discrete-event simulation*, the target system (sometimes called the *physical system*) is modeled by a set of state variables that make discrete transitions in response to events [12]. A uniprocessor host computer performs discrete-event simulation by removing the first event from an event list (ordered by target time), processing the event to change the system state, and scheduling zero or more events. *Distributed, discrete-event simulation* partitions the target system's state and event lists among the multiple processing nodes of a parallel host system. Generally, each node processes events in a manner similar to a uniprocessor and sends messages to other nodes to schedule events that affect remote state. The fundamental difficulty is to determine the next event. The first event in the local event list may not be the event that should be processed first, because a message may subsequently arrive with an event from an earlier target time.

Aggressive distributed, discrete-event simulation algorithms optimistically process the local event list and exchange information that causes rollbacks of prematurely-processed events. By contrast, *conservative* algorithms ensure causality by exchanging information to determine when the first item in the local list can be processed safely. A node processes its first event only when no node will subsequently send an earlier event. Section 4.1 describes WWT's conservative technique and Section 6 compares it with related algorithms.

3 Parallel Direct Execution of Shared-Memory Programs

This section describes how WWT directly executes shared-memory programs on the CM-5. The next section provides details of the performance calculation. Non-shared-memory machines, such as the CM-5, do not provide hardware mechanisms to handle cache misses on remote memory references. WWT simulates this mechanism with a software trap handler that is invoked on target cache misses. Our approach is a fine-grained extension of Li's shared virtual memory. With this technique, WWT directly executes all instructions and all memory references that hit in the target cache. Only target cache misses require simulation.

Li's *shared virtual memory (SVM)* [21] implements shared memory on a distributed-memory system. Pages local to a processor are mapped into the processor's virtual address space. A remote page is left unmapped so a reference to it causes a page fault, which invokes system software to obtain the page.

Although SVM allows direct execution of memory references, sharing is limited to page granularity (typically 4K bytes), which is much larger than target cache blocks (e.g., 128 bytes). WWT refines SVM to support fine-grained sharing by logically adding a valid bit to every 32 bytes in the host pages underlying the target cache. An access to an invalid block traps and software simulates a cache coherence protocol that obtains the block. The CM-5 does not provide memory valid bits, but it does provide precise interrupts on double-bit error correcting code (ECC) errors. WWT marks invalid blocks with a bad ECC value, which causes a trap on loads and stores.

WWT target programs are SPARC binaries that run directly on the CM-5's SPARC processors. As described in Section 2.1, each target process sees the virtual address space illustrated on the left side of Figure 2. WWT runs in a different SPARC context. The right side of Figure 2 illustrates WWT's virtual address space, all of which is mapped to local memory. The text segment holds WWT code and the stack segment contains WWT's and the target program's stack. WWT's data is divided into four regions: WWT-state, target text, target node memory, and cache page frames (CPFs). WWT-state holds WWT-specific data structures. Target text is a copy of the target program. Target node memory holds all data that resides in the target node's local memory module and is logically equivalent to the target node's physical memory.[1] Finally, WWT uses CPFs to model the target node's cache, as explained below.

To understand how WWT models a cache, let's follow the steps (under a simple cache coherence protocol) when a memory reference misses in the cache. Initially, the target's private and shared data segments are unmapped so the first reference to a page generates a fault that the CM-5 passes to WWT. This reference would be a cache miss on the target shared-memory machine. WWT regains control on all cache misses so it can simulate the target cache.

Assume for a moment that the target cache's block size equals the host page size (4K bytes). In this case, WWT handles a cache miss in a similar manner as shared virtual memory (SVM) [21]. Upon a cache miss, WWT allocates a page from the pool of CPFs and changes the target's page tables so the faulting address maps to the CPF. WWT then simulates the target cache-coherence protocol by sending messages to other processors to obtain the referenced block and update directory information. Finally, WWT writes the block to the allocated page and resumes the target program. Subsequent references to the same block execute directly. WWT removes a block from the cache—either because of cache replacement or a coherence message—by unmapping the CPF and returning the block's data to its home memory module. As a further refinement, we simulate a translation lookaside buffer (TLB) by selectively unmapping CPFs.

Cache blocks in foreseeable systems will be much smaller than 4K bytes, so WWT simulates multiple blocks per host page. WWT logically adds a valid bit to each block in a mapped CPF page. On a target cache miss to a previously unrefenced page, WWT maps the page into a CPF (as above), writes the block's data into a valid block, and marks as invalid all other blocks on the page. Subsequent references to these blocks cause an "invalid" trap that WWT

[1] WWT can model machines in which private and shared data is stored in the same cache—by storing private data in CPFs—or stored separately—by mapping private data to target node memory.

services similarly.

Since the CM-5, like most machines, does not provide valid bits, we synthesize them from existing hardware. We divide the CPF pages into blocks of the same size as those in the CM-5's SPARC cache (32 bytes). WWT marks a block invalid by using a CM-5 diagnostic mode to change the error correcting codes (ECC) for the block's memory double-words to invalid values that cause a double-bit ECC error. By running the CM-5's SPARC cache in write-back mode, all target memory references appear to the memory controller as 32-byte cache fills. A cache fill of a valid block loads the data into the SPARC cache. An invalid block incurs a double-bit ECC error, which causes the memory controller to interrupt the processor and transfer control to WWT.

Target cache blocks larger than 32 bytes are easily handled by having WWT manipulate aligned groups of 32-byte blocks. Read-only blocks require setting the block's page protection to read-only, which causes protection violations on writes. Regrettably, protection must be read-only if any block in a page is read-only. A write to another, writable block in the page causes a protection violation, but WWT determines that the write is permissible and completes it.

Conservative users may be surprised to learn that WWT's use of ECC bits has little effect on the CM-5's standard ECC coverage. Most memory is outside the CPFs and unaffected by our use of the ECC bits. Valid blocks within the CPFs have valid ECC. On an ECC error, WWT checks a block's validity. If the block is invalid, WWT simulates a cache miss. Otherwise, WWT signals a true ECC error. However, a bit error could make an invalid block appear valid. Writing invalid ECC codes in the two double words in a block ensures that two single-bit errors are necessary to cause this problem.

4 Calculating Performance

The discussion above described how WWT accurately and efficiently mimics the functionality of a cache-coherent, shared-memory computer. Functional simulation without performance metrics, however, is insufficient to study and compare computer architectures. A program's running time is the single most important metric in determining how well the program runs on a computer. This section describes how WWT computes a program's execution time on a target machine.

WWT adds logical clocks to target processes, protocol messages, directory hardware, etc. to enable WWT to model latencies, dependencies, and queuing. The result is an event-driven simulation of a parallel target machine that runs on a parallel host machine. WWT differs from other distributed, discrete-event simulators [12] because its workload is an executing program. Driving a simulation from an executing target program requires new techniques because of the frequency with which the program modifies the target machine's state. By contrast, a queuing network simulation only modifies target state on job arrival and departure. Treating every instruction execution as an event would be prohibitively costly and would forfeit the benefits of the efficient functional simulation described above.

WWT uses two techniques to combine the distributed simulation of a parallel computer with direct execution. First, WWT manages interprocessor interactions by dividing program execution into lock-step quanta to ensure all events originating on a remote node that affect a node in the current quantum are known at the quantum's beginning. Second, WWT orders all events on a node for the current quantum and directly executes the process up to its next event. These two techniques result in an efficient simulation that delivers accurate, reproducible results without approximations. Section 4.1 describes how WWT simulates interactions among processor nodes. Section 4.2 shows how WWT integrates direct-execution of target programs with discrete-event simulation.

4.1 Inter-Node Simulation

In this section, we ignore the details of WWT's processor simulation. From this perspective, WWT performs a distributed, discrete-event simulation that exploits characteristics of the target and host machines. First, rollback is impractical because directly executed programs can modify any part of a target processor's state (e.g., registers and memory). Second, determining the local time of nodes that could possibly send an event is expensive because any target node can communicate with any other target node. Third, our host system hardware (the CM-5) does not guarantee that messages from one host node to another node arrive in order. However, the CM-5 does provide hardware support for efficient reductions.

WWT coordinates simulation of processor nodes by breaking the target program's execution into lock-step quanta Q target machine instruction cycles long. Every quantum ends with a CM-5 reduction that ensures all outstanding messages containing events are received. Setting $Q \leq T$ (where T is the minimum latency of the target machine's interconnection network) guarantees that at the beginning of a quan-

tum, a host node has received all remotely-generated events that could affect the target node in the quantum, because:

- all messages containing events that originated during the previous quantum have been delivered, and
- all events produced during the current quantum cannot affect target state on other nodes during this quantum.

We choose this approach because it avoids rollbacks, makes good use of the efficient CM-5 reductions, and tolerates unordered data messages. In practice, the technique works well, as the numbers in Section 5 demonstrate. Section 6 discusses how the technique relates to other distributed, discrete-event simulation algorithms.

4.2 Node Simulation

This section explains how WWT coordinates direct program execution with the distributed, discrete-event simulation. On each host node, WWT runs the target program up to the first event, processes the event, and continues running the program up to the next event or until the quantum expires. If the local program produces events (e.g., cache misses), they are easily added to the event list.

WWT directly executes many instructions consecutively by identifying the points in a program at which events can interact with the program and only checking for a pending event at these points. Instructions that do not access memory only affect a processor's local state and neither cause nor are directly affected by shared-memory events. These instructions execute directly. A memory reference produces an event when it misses in the target machine's cache. Fortunately, fine-grained shared virtual memory (Section 3) efficiently identifies these memory references and returns control to WWT. Finally, shared-memory references can interact with events caused by other processor nodes. For example, another processor can steal a cache block from the local node. These events must be carefully coordinated so state changes occur at the correct point with respect to the local program's execution. Because WWT knows all non-local events at the beginning of a quantum, it runs the program up to the first event, changes the cache or directory state as dictated by the event, and continues the program.

Each WWT host node performs the discrete-event simulation of a target node by repeatedly processing the first event in the local event list and performing an appropriate action. Target machine coherence protocol messages are sent via CM-5 data messages. WWT timestamps each message with its sending time plus T (to model the target network latency). When a data message is received, WWT inserts its event on the local event list. WWT also schedules a quantum-expiration event every Q target cycles. This event causes a host node to wait for all other nodes to reach a barrier and all data messages to reach their destination. WWT uses the CM-5's fast reductions to determine when all messages are delivered.

WWT currently processes requests on a target directory at the request's arrival time rather than enqueuing the request until its service time. This optimization, advocated by Ayani [3], is possible because the directory uses first-come-first-serve queuing in which an event is unaffected by events that arrive after it. The optimization is important because directory service and completion events need not be enqueued and the running program is interrupted less often. WWT processes external requests on a target cache in the same fashion as directory requests.

```
— Initialization not shown
while (simulation not complete) do
    begin
        while (time (cpu) ≥ time (event_queue.head)) do
            begin
                — Process interprocessor events that occur
                — before the next instruction execution.
                — Remember: quantum expiration is event.
                first_event = dequeue (event_queue);
                process_event (first_event);
            end

        — Run the target program up to the next event,
        — Δ_permitted cycles in the future.
        Δ_permitted = time (first_event) - time (cpu);
        — Δ_actual is the actual time that the program ran.
        Δ_actual = run_program (Δ_permitted);
        time (cpu) = time (cpu) + Δ_actual;
    end
```

Figure 3: WWT's event loop.

Figure 3 shows WWT's event processing loop. The routine *run_program* runs the target program and returns control to WWT at the first target cache miss or before the first memory reference that executes at or after $\Delta_{permitted}$ cycles. The program need not stop precisely at $\Delta_{permitted}$ cycles, because additional instructions that do not reference memory cannot interact with external events. The function returns the amount of time that the program actually ran (Δ_{actual}). On a cache miss, $0 \leq \Delta_{actual} \leq \Delta_{permitted}$,

so the cache miss occurs before the first event in the local event list. If the memory access references the local memory module, its event is inserted before the former (and still unprocessed) first event and the loop continues. If the miss references a remote memory module, WWT sends a message, suspends the program, and processes the event list since the response to the message cannot arrive until the next quantum.

run_program returns control of the host processor to the target program. WWT ensures that the target program always returns before the next event by using a modified version of *qpt* [4] to add quick tests to the target program's binary executable. Before resuming the target program, WWT sets a counter in a global register to $\Delta_{permitted}$. Conceptually after each instruction, the instrumented target program decrements the counter by the instruction's static cycle cost. For memory references, this cost assumes a cache hit. On a cache miss, WWT regains control and accounts for the miss penalty. To improve performance, the cycle counter is actually decremented only before non-private references and at the end of basic blocks. The instrumented program only tests the counter before non-private references. If the counter is negative, the program returns control to WWT, which processes the next event before executing the memory reference. The decrement and test each require a single instruction and consequently do not greatly affect the program's execution time.

5 A Comparison

To improve upon direct-execution uniprocessor simulators, WWT must obtain results of comparable accuracy and run large parallel programs more quickly. This section shows that WWT produces results similar to Stanford's Tango/Dixie, a direct-execution uniprocessor simulator. It also shows how WWT scales to simulate large target systems more effectively than a uniprocessor simulator.

5.1 Validation

WWT is a complex system for modeling complex software and hardware systems. To build confidence in WWT, we compared it against an existing simulator—a process that is often optimistically called *validation*. The similar—though not identical, for reasons discussed below—results increased our confidence in WWT. Even so, the process did not prove WWT correct, because testing never proves software to be correct. The validation only showed that the two systems produced similar results. It did not demonstrate that either system correctly modeled a real computer.

Name	Application	Input Data Set	Size (lines)
cholesky	Sparse matrix cholesky factorization	bcsstk14	1888
mp3d	Hypersonic flow simulation	10000 mols 20 iter	1607
tomcatv	Parallel SPEC benchmark	128×128 10 iter	404

Table 1: Applications used in validation.

These three benchmarks, two from the SPLASH benchmark suite [29] and a parallelized version of a SPEC benchmark [31], were chosen because they make little use of the math libraries. We could not simulate these libraries with Tango, because we did not have MIPS math library sources.

We compared WWT against *Dixie*, a Tango-based simulation of the DASH multiprocessor [26]. Tango is a direct-execution simulation system that runs on a uniprocessor MIPS-based workstation. Tango instruments a target program's assembly code with additional code that computes instruction execution times and calls the simulator before loads and stores. The Dixie memory system simulation models the DASH prototype hardware [20]. Together, Dixie and Tango execute parallel shared-memory programs and estimate system performance.

Our original intent was to perform a black box comparison by writing our own DASH simulator for WWT (WWT/DASH) from only published details and reproducing DASH performance results. Unfortunately, this goal proved impossible because of myriad unpublished, subtle, and detailed design choices that significantly affect shared-memory system performance. For example, differences in memory allocation (e.g., in *malloc* and *sbrk*) resulted in a nearly 100% discrepancy in cache performance.

Instead, we "opened the black box" and modified both WWT/DASH and Tango/Dixie until they modeled similar systems. We made most changes to our description of DASH. The principal exception was the memory hierarchy. The DASH multiprocessor has a four-level memory hierarchy: primary cache, write buffer, secondary cache, and remote access cache. To reduce the complexity of our DASH simulation, we disabled most levels of the hierarchy and simulated a system with a single direct-mapped cache, no clusters, and $T = 100$ cycles to send a coherence protocol message.

During the validation, we found several major dif-

Application	Proc	Message Counts			Simulated Cycles		
		WWT	T/D	Diff.	WWT	T/D	Diff.
cholesky	4	1253504	1259467	-0.47%	103379891	107713586	-4.02%
cholesky	8	1756122	1747374	0.50%	93356751	99380412	-6.06%
cholesky	16	2197179	2220362	-1.04%	87566479	94468953	-7.31%
mp3d	4	2823164	2799511	0.84%	95464151	104226210	-8.41%
mp3d	8	3276119	3267250	0.27%	64579958	68684983	-5.98%
mp3d	16	3778487	3769719	0.23%	47764673	49901197	-4.28%
tomcatv	4	332995	332181	0.25%	52214338	43086273	21.19%
tomcatv	8	330091	328362	0.53%	26468922	22029613	20.15%
tomcatv	16	197397	195722	0.86%	12606286	9991406	26.17%

Table 2: Results of Validation.

WWT/DASH (WWT) and Tango/Dixie (T/D) are very close in message counts. The percent difference ($100\% \times (\text{WWT} - \text{T/D})/\text{T/D}$) is less than 1.1% in all cases. The simulated cycle counts—execution times—are not as close since the two host systems have different instruction sets, different compilers, and use different instrumentation. Both systems are simulating direct-mapped, 64K byte caches with 32-byte blocks.

ferences in the way that WWT and Tango/Dixie manage memory. First, Tango/Dixie does not simulate a translation lookaside buffer (TLB), while WWT does. For large virtual address spaces and large data sets, TLB performance can be important. To minimize this difference, we simulated a very large, fully-associative TLB in WWT. Second, because Tango shares the same address space as an application program, Tango's data structures are intermixed with the application's private data. In WWT, an application has its own address space. This difference caused non-trivial differences in cache performance. To remedy it, we ensured that both simulators cached only shared data and allocated shared data contiguously.

After making these changes, we ran the three parallel benchmarks listed in Table 1. Table 2 summarizes the simulated message and target cycle counts. We simulated target systems with 4–16 processors, because Tango/Dixie could not handle larger systems on our workstations. The message counts are close, with less than 1.1% deviation in all cases. The estimated cycle counts—target execution times—are not quite as close. Some deviation is unavoidable because Tango/Dixie runs on a MIPS, while WWT runs on a SPARC. Cmelik et al. showed that programs on these architectures typically perform within 10% for similar compilers [8]. To minimize instruction set differences, we assumed all instructions execute in a single cycle. A problem, however, is that Tango/Dixie counts instructions in compiler-generated assembly code. Because the MIPS assembler expands pseudo-instructions, assembly code differs from executed binary code. This was particularly important for *Tomcatv*, whose inner loop expanded from 969 pseudo-instructions to 1394 binary instructions (on the SPARC, the inner loop takes 1329 binary instructions). Despite these differences, the cycles counts for *MP3D* and *Cholesky* are within 10%. *Tomcatv's* differences are 20-26%, due to the large assembler expansion. Finally, Tango/Dixie did not measure the math library routines, whose sources were unavailable. WWT, because it instruments and runs executables, measures all code in a program.

5.2 Performance Evaluation

Another goal of WWT is to simulate programs faster than direct-execution uniprocessor simulators such as Tango/Dixie. It is tempting to compare directly the performance of Tango/Dixie and WWT/DASH. For at least three reasons, this comparison is not meaningful. First, the Dixie memory system simulator models the DASH implementation more completely than WWT/DASH. In particular, although we disabled the effects of Dixie's four-level memory hierarchy, much of the code still executed. Second, isolating the effect of different instruction sets, compilers and operating systems is difficult. Third, WWT/DASH is newer than Tango/Dixie. Unreleased versions of Tango/Dixie might perform better.

For these reasons, we focus on how performance changes as we increase the number of processors in a target system, both without and with changing the program's size. The WWT numbers were collected on a 64-processor CM-5 running pre-release (beta) CMOST 7.2 S4.

Consider first the simulation time when the num-

Application	Procs	WWT			Tango/Dixie		
		Time (sec.)	Rate (cyc./sec.)	Slowdown (host/target)	Time (sec.)	Rate (cyc./sec.)	Slowdown (host/target)
cholesky	4	586	176416	187	5659	19034	2101
cholesky	8	353	264466	124	8474	11727	3410
cholesky	16	236	371044	88	16695	5658	7068
cholesky	32	182	469111	70	n/a	n/a	n/a
cholesky	64	168	504118	65	n/a	n/a	n/a
mp3d	4	725	131674	250	3891	26786	1493
mp3d	8	322	200558	164	4048	16967	2357
mp3d	16	179	266841	123	4601	10845	3688
mp3d	32	114	335560	98	n/a	n/a	n/a
mp3d	64	79	414663	79	n/a	n/a	n/a
tomcatv	4	114	458020	72	2705	15928	2511
tomcatv	8	58	456360	72	2902	7591	5269
tomcatv	16	20	630314	52	3039	3287	12166
tomcatv	32	13	533214	61	n/a	n/a	n/a
tomcatv	64	14	304765	108	n/a	n/a	n/a

Table 3: WWT/DASH and Tango/Dixie simulation speed and slowdown.

This table shows simulation time, the number of target cycles simulated per second, and the host cycles required to simulate each target cycle. WWT runs on 33-Mhz SPARCs in a CM-5. Tango/Dixie runs on a 40-Mhz DECstation 5000/240. Results for Tango/Dixie with 32 and 64 processors are unavailable (n/a), because we could not run Tango/Dixie on these target systems on our workstations.

ber of processors in a target system increases and the program's size is fixed. Uniprocessor simulators are, at best, unaffected by the change since the program, at best, performs the same operations on more target processors. In practice, additional context switching overhead and host TLB and cache contention increase simulation time. Since WWT runs on a parallel host, we increase the number of host nodes (up to the size of the host) as the target system grows. Consequently, WWT's simulation time should decrease, rather than increase, because each target (and host) node performs fewer operations. Table 3 and Figure 4a clearly demonstrate this behavior. For *MP3D* and *Tomcatv*, Tango/Dixie's execution time increases slightly from 4 to 16 processors. WWT's elapsed time decreases nearly linearly for *MP3D* and *Tomcatv* as the target system size increases. In *Cholesky*, moreover, processes spin on a shared variable, which increases the number of target instructions executed as the number of target processors increases. Since Tango/Dixie simulates all processors serially, its execution time triples from 4 to 16 processors. WWT's elapsed time again decreases.

The advantage of WWT is more striking for scaled speedup. Gustafson [14] and others have argued that parallel speedup is uninteresting: give scientists a larger machine and they will solve larger problems in the same amount of time. This argument calls for scaling a problem's size approximately linearly with the number of processor nodes. On uniprocessor simulators, like Tango/Dixie, scaling a problem increases simulation time at least linearly. On WWT, as illustrated in Figure 4b, simulation time increases much more slowly.

6 Related Work

WWT is a tool for evaluating large-scale, parallel computer system software and hardware. The two closely-related methods for studying large-scale parallel computers are hardware prototyping and software simulation. The methods are not mutually-exclusive as simulation usually precedes prototyping.

Stanford DASH [20], MIT Alewife [7], and MIT J-Machine [10] are projects that built hardware prototypes. WWT offers at least four advantages over hardware prototyping. First, WWT ran programs after two months, ran large applications four months later (instead of multiple years), and produced accurate cycle counts in two more months. Second, WWT modules can be modified in days or weeks (instead of months or years). Third, WWT can be built in a university without the infrastructure for hardware development. Finally, WWT can be distributed to

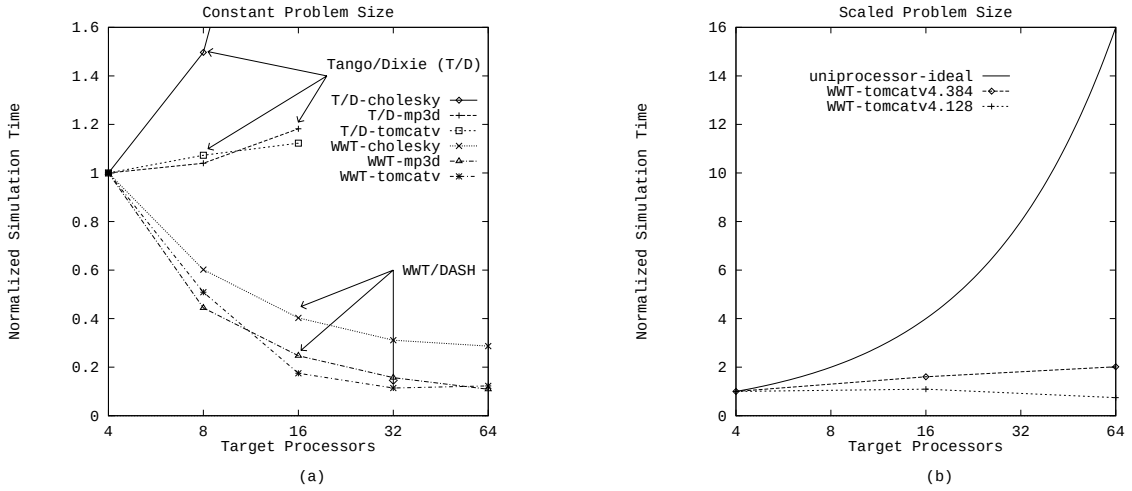

Figure 4: Normalized simulation time vs target system size.

This figure plots the relative simulation times as the target system size increases for both constant problem size (a) and scaled (linearly with target system size) problem size (b). Each curve is normalized against the respective simulator time for 4 processors. For constant problem size, Tango/Dixie's simulation time increases with target system size, while WWT's time decreases nearly linearly. For scaled problem size, WWT's simulation time increases slowly, while the simulation time on a uniprocessor host must increase at least linearly (exponentially on this semi-log plot). The tomcatv4.128 line corresponds to 4, 16, and 64 processors and 128×128, 256×256, and 512×512 matrices, respectively. The tomcatv4.384 line corresponds to 4, 16, and 64 processors and 384×384, 768×768, and 1536×1536 matrices, respectively.

other sites.

The disadvantages of WWT include not providing participants with experience in building actual hardware and the ever-present question as to how closely a simulation reflects the actual system. Of course, even academic hardware prototypes are not completely realistic, because of constraints on infrastructure, money, and availability of experienced designers.

An alternative to hardware prototyping is software simulation. MIT Proteus [6], Berkeley FAST [5], Rice Parallel Processing Testbed [9], and Stanford Tango [11] are simulation systems that simulate parallel machines by running actual programs (as opposed to distribution-driven or trace-driven workloads). All use direct execution and run on a uniprocessor host. WWT improves on these systems by extending direct execution to all target operations except shared-memory references that *miss* and by running on a parallel host with gigabytes of main memory. These differences enable WWT to evaluate larger and more realistic workloads including real applications, data sets, and system software.

A disadvantage of WWT with respect to uniprocessor simulators is the high cost of WWT's host. Nevertheless, WWT provides two advantages. First, it can perform more realistic evaluations, because the CM-5 provides more main memory than can be placed on most workstations (and this memory accounts for a significant fraction of the CM-5's cost). Second, it facilitates successive refinement of software and hardware by providing a much faster response time.

Other researchers [12, 16, 18, 24, 25, 3, 28, 27] have studied simulation systems for parallel hosts. These systems perform a discrete-event simulation of a parallel target using a discrete-event workload. These systems and WWT exploit the parallel host by avoiding a fine-grained global simulation clock that advances in lock-step.

The key advance of WWT over these systems is that WWT supports dynamic execution of target program binaries. As discussed by Covington, et al. [9], execution-driven simulation of computer systems produces more detailed results than either distribution- or trace-driven simulation. To the best of our knowledge, all other distributed, discrete-event simulation systems simulate queuing networks or computer systems with stochastic workload models, not real programs.

WWT's method for coordinating nodes for the exe-

cution time calculation is a conservative, synchronous method. *Synchronous* algorithms use the target times of some or all neighboring nodes to ensure causality [3, 24, 28, 27]. The difference between the target times of two nodes is called *lag* and the minimum target time for one node to affect another is called *distance*. Synchronous algorithms ensure the lag between each pair of nodes is always less than the distance.

WWT ensures that lag is less than distance with barriers at fixed target intervals smaller than the minimum distance between any two target nodes. Our approach is closely related to the methods of Ayani, Lubachevsky and Nicol. Ayani uses barriers, but runs on a shared-memory host machine and simulates at most one event per node per barrier. Lubachevsky [24] repeatedly broadcasts the time of the slowest node (to maintain a bounded lag), but allows the lag to be greater than the distance between some nodes. Nicol [27] uses barriers at variable intervals. After each barrier, nodes cooperate to determine the next barrier time. WWT eliminates this phase by using a fixed time between barriers.

Konas and Yew and Lin et al. perform parallel simulations of multiprocessors, but neither system directly executes target programs. Konas and Yew [18, 19] use distribution-driven workloads and simulation algorithms that rely on a shared-memory host. Lin et al. [22] use trace-driven workloads in which the trace and interactions among processors are unaffected by a memory-reference miss (as if a cache hit and miss take the same time). WWT models reference times more accurately to compute target execution time and allows memory system behavior to affect program execution.

Finally, WWT allocates cache page frames in 4K-byte blocks, but usually uses smaller blocks ($\geq$ 32 bytes) for data transfers and coherence. The IBM 360/85 cache [23] used different block sizes for allocation and transfer. Goodman [13] discusses the use of different block sizes for allocation, transfer, and coherence. These ideas are also used more recently by Tamir [32] and Kendall Square [17].

7 Extensions and Future Work

Currently, WWT is a novel simulator of cache-coherent, shared-memory computers that runs on a Thinking Machines CM-5. We believe that virtual prototyping is very general concept and can be used to model *any* target system and to extend WWT to run on other hosts. The primary issue is how efficiently a specific target feature can be modeled. As an example, WWT could easily support message-passing programs. This only requires a library of message-passing primitives that use WWT timestamps to compute message latency. A more expensive change would enable WWT to model interconnection network topology and contention. WWT could charge a minimum latency determined by network topology and add a stochastic contention penalty. Finally, WWT could, at great cost, model contention exactly by using CM-5 nodes to simulate network routing.

WWT can model most target node features. Instruction set extensions—e.g., a `compare-and-swap` instruction—are implemented by existing instructions (either in-line, in a call, or with a simulator trap). *qpt* models the instruction's performance by charging an assumed cycle count. *qpt* can also form the basis of a binary-to-binary translator that permits non-SPARC instruction sets [30]. This technique could model a dynamic pipeline of arbitrary complexity. At some point, however, the complexity of translating an instruction set outweights the benefits of direct execution and an instruction simulator is a better alternative.

Fine-grained shared-virtual memory efficiently simulates caches that use pseudo-random replacement. A second-level cache can be simulated during the traps for level-one cache misses. Algorithms that cause state changes on some cache hits, such as LRU replacement, require a full address trace, which *qpt* can provide.

Currently, WWT runs one target node on a host node. Future versions will run multiple target nodes on a host node, which will permit evaluation of larger systems. WWT will use a separate context (i.e., virtual address space) for each target node. During a quantum, a host node will run each target node, in turn, for Q cycles. Causality is preserved because the communication latency of the target nodes running on a host node is still $T \geq Q$ cycles. In addition, we could switch contexts more frequently when modeling a target system with low intracluster communication latency. Finally, this improvement will enable us to simulate multithreaded architectures that context switch frequently.

WWT could run on machines besides a CM-5. The key CM-5 features exploited by WWT are precise interrupts on ECC errors, fast user-level messages, and efficient reductions. Without the ECC mechanism, fine-grained shared virtual memory could be implemented with memory tag bits (e.g., Tera [2]). On a machine with neither mechanism, *qpt* can add additional code to a program to directly test if a shared-memory reference will miss, as in Tango/Dixie. The performance of this approach relative to the ECC

mechanism depends on its overhead on cache hits, since the cost of a cache miss dwarfs the tests and traps.

WWT can run on machines without the CM-5's fast user-level messages and reductions. High latency messages will slow WWT's execution. On a shared-memory host, WWT could directly write to remote event lists to add events. Again, the efficiency depends how quickly the host's memory system perform these accesses. WWT quanta end with a reduction that completes when all outstanding messages are received. A barrier would suffice on a machine that has another mechanism for ensuring that remote events are scheduled (e.g., shared memory). On machines without barrier or reduction hardware, WWT could use software reductions and barriers, or the system could be radically changed to use another distributed, discrete-event simulation technique.

This paper contains some results for a simulation of a shared-memory machine similar to Stanford DASH. We are using WWT to compare existing and new cache-coherence protocols. WWT facilitates these studies, because it clearly separates the modules that specify a target machine's cache coherence protocol and the rest of WWT. To date, WWT runs DASH [20], Dir_iNB for $i = 1 \ldots N$ [1], Dir_iB for $i = 0 \ldots N$ [1], and Dir_1SW variants [15, 34].

8 Conclusions

This paper describes the Wisconsin Wind Tunnel (WWT)—a system for evaluating cache-coherent, shared-memory computers on a Thinking Machines CM-5. WWT runs parallel shared-memory binaries and concurrently calculates the programs' execution times on the target hardware with a distributed, discrete-event simulation. The non-shared-memory host directly executes all target instructions and memory references that hit in the target cache. WWT's shared memory uses the CM-5 memory's error-correcting code (ECC) as valid bits to build a fine-grained extension of Li's shared virtual memory.

WWT calculates target program execution times on the parallel host with a distributed, discrete-event simulation algorithm. WWT manages the interactions among target nodes by dividing execution into lock-step quanta that ensure all events originating on a remote node that affect a node in the current quantum are known at the quantum's beginning. On each node, WWT orders all events in a quantum and directly executes the process up to the next event.

We showed that WWT produces results that are close to Tango/Dixie for a Stanford DASH-like system. Finally, we examined how WWT's performance scales as target systems increase in size. WWT's execution time decreases for fixed-size problems running on larger target systems and increases slowly as problems are scaled to run on larger systems. Neither result is surprising, because WWT uses the processors and memory of a parallel host. Nevertheless, they demonstrate that WWT can support evaluations of much more realistic parallel workloads than previously possible without building hardware.

Acknowledgements

Dave Douglas, Danny Hillis, Roger Lee, and Steve Swartz of Thinking Machines provided invaluable advice and assistance in building WWT. Satish Chandra, Glen Ecklund, Shubhendu Mukherjee, Subbarao Palacharla, and Timothy Schimke helped develop WWT and applications. Richard Fujimoto and David Nicol provided readings in distributed, discrete-event simulation. Sarita Adve, Richard Fujimoto, Stephen Goldschmidt, Rahmat Hyder, Alain Kägi, Edward Lazowska, David Nicol, and John Zahorjan provided helpful comments that greatly improved this paper.

We would also like to acknowledge the invaluable assistance provided by the Stanford DASH project. Helen Davis, Kourosh Gharachorloo, Stephen Goldschmidt, Anoop Gupta, John Hennessy, and Todd Mowry wrote and generously provided Tango and Dixie. Singh *et al.* [29] wrote and distributed the SPLASH benchmarks.

References

[1] Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, June 1990.

[3] Rassul Ayani. A Parallel Simulation Scheme Based on the Distance Between Objects. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 113–118, March 1989.

[4] Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[5] Bob Boothe. Fast Accurate Simulation of Large Shared Memory Multiprocessors. Technical Report CSD 92/682, Computer Science Division (EECS), University of California at Berkeley, January 1992.

[6] Eric A. Brewer, Chrysanthos N Dellarocas, Adrian Colbrook, and William Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991.

[7] David Chaiken, John Kubiatowics, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234, April 1991.

[8] Robert F. Cmelik, Shing I. Kong, David R. Ditzel, and Edmund J. Kelly. An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 290–302, April 1991.

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

[10] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Nuth, Scott Wills, Paul Carrick, and Greg Flyer. The J-Machine: A Fine-Grain Concurrent Computer. In G. X. Ritter, editor, *Proc. Information Processing 89*. Elsevier North-Holland, Inc., 1989.

[11] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing (Vol. II Software)*, pages II99–107, August 1991.

[12] Richard M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[13] James R. Goodman. Coherency for Multiprocessor Virtual Address Caches. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 408–419, October 1987.

[14] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May 1988.

[15] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 262–273, October 1992.

[16] David R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[17] Kendall Square Research. Kendall Square Research Technical Summary, 1992.

[18] Pavlos Konas and Pen-Chung Yew. Parallel Discrete Event Simulation on Shared-Memory Multiprocessors. In *Proc. of the 24th Annual Simulation Symposium*, pages 134–148, April 1991.

[19] Pavlos Konas and Pen-Chung Yew. Synchronous Parallel Discrete Event Simulation on Shared-Memory Multiprocessors. In *Proceedings of 6th Workshop on Parallel and Distributed Simulation*, pages 12–21, January 1992.

[20] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[21] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[22] Y.-B. Lin, J.-L. Baer, and E. D. Lazowska. Tailoring a Parallel Trace-Driven Simulation Technique to Specific Multiprocessor Cache Coherence Protocols. Technical Report 88-01-02, Department of Computer Science, University of Washington, March 1988.

[23] J. S. Liptay. Structural Aspects of the System/360 Model 85, Part II: The Cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[24] Boris D. Lubachevsky. Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks. *Communications of the ACM*, 32(2):111–123, January 1989.

[25] Jayadev Misra. Distributed-Discrete Event Simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[26] Todd Mowry and Anoop Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12:87–106, June 1991.

[27] David Nicol. Conservative Parallel Simulation of Priority Class Queueing Networks. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):398–412, May 1992.

[28] David M. Nicol. Performance Bounds on Parallel Self-Initiating Discrete-Event Simulations. *ACM Transactions on Modeling and Computer Simulation*, 1(1):24–50, January 1991.

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

[30] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary Translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[31] SPEC. SPEC Benchmark Suite Release 1.0, Winter 1990.

[32] Yuval Tamir and G. Janakiraman. Hierarchical Coherency Management for Shared Virtual Memory Multicomputers. *Journal of Parallel and Distributed Computing*, 15(4):408–419, August 1992.

[33] Thinking Machines Corporation. The Connection Machine CM-5 Technical Summary, 1991.

[34] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993.