Fast and Portable Parallel Architecture Simulators: Wisconsin Wind Tunnel II

Shubhendu S. Mukherjee[1], Steven K. Reinhardt[2], Babak Falsafi[3], Mike Litzkow[4], Steven Huss-Lederman[4], Mark D. Hill[4], James R. Larus[5], and David A. Wood[4]

[1]Compaq Computer Corporation, 334 South Street, SHR3-1/S30, Shrewsbury, MA 01545, USA, 508-841-3467, Shubhendu.Mukherjee@compaq.com.
[2]EECS Department, University of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109-2122, U.S., 734-647-7959, stever@eecs.umich.edu.
[3]School of Electrical and Computer Engineering, Purdue University, 1285 Electrical Engineering Building, West Lafayette, IN 47907, U.S., 765-494-9064, babak@ecn.purdue.edu.
[4]Computer Sciences Department, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, Wisconsin 53706-1685, U.S., 608-265-3402, {mike,lederman,markhill,david}@cs.wisc.edu.
[5]Microsoft Research, One Microsoft Way, Redmond, WA 98052, 425-936-2981, larus@microsoft.com.

Abstract

Analysis of future parallel computers requires rapid simulation of target designs running realistic workloads. These simulations have been accelerated by two techniques: direct execution and the use of a parallel host. Historically, these techniques have been considered to lack portability. We identify four key operations necessary to make these simulations portable. This allows us to run the Wisconsin Wind Tunnel II (WWT II) readily on a wide range of SPARC platforms from a workstation cluster to a symmetric multiprocessor (SMP).

WWT II has good performance and scalability as shown on a range of benchmarks. WWT II achieves speedups between 8.6 and 13.6 on a 16 host processor SMP. Finally, we show that parallel simulation with WWT II is cost-effective.

Keywords: architecture, simulation, parallel, portable, cost-effectiveness

IEEE Concurrency

1 Introduction

Simulation is an important technique for studying computer architectures ranging from microprocessors to parallel computers. Simulation speeds design by enabling architects to evaluate computers without building hardware prototypes. However, simulating large problems—parallel machines with realistic workloads—requires vast amounts of computation and memory. Two techniques, direct execution and parallel simulation, make this approach feasible.

In direct execution [1], a program from the system under study (the *target*) runs on an existing system (the *host*). For example, a target's floating-point multiply executes as a floating-point multiply instruction on the host. The host calculates the target's execution time and only simulates operations unavailable on the host. Direct execution can run orders of magnitude faster than pure software simulation (which interprets every target instruction). This approach can accurately calculate the target execution time for statically scheduled processors with blocking caches [1].

Parallel simulation of a parallel computer further speeds simulation by exploiting the parallelism inherent in the target parallel computer and the large memory in a parallel host to hold the working set of the simulator without paging. The advent of low-cost parallel computers, such as symmetric multiprocessors (SMPs) and clusters of workstations (COWs), make parallel simulation very attractive. In contrast, Rice RSIM and Stanford SimOS use uniprocessor hosts.

Unfortunately, parallel, discrete-event, direct-execution simulators are complex pieces of software that can be difficult to build and port. In part, these simulators are not portable because they rely on machine-specific features. They are tied to specific instruction sets by the need to modify target executables or assembly code to calculate a target's execution time and simulate missing features. Some simulators [2, 3] also modify the operating system to detect target cache misses. In addition, parallel simulators often use machine-specific synchronization and communication features to achieve good parallel performance.

As the authors and users of two generations of parallel direct-execution simulators, we are painfully aware of these low-level dependencies. In building our tools, we have identified four key operations that underlie parallel, discrete-event, direct-execution simulation:

- calculation of target execution time,
- simulation of features of interest,

- communication of target messages, and
- synchronization of host processors.

We show that these four operations can be implemented in a fashion that minimizes the dependence of a parallel simulator on host-specific features. This is achieved with two tools, called *Elsie* and *Synchronized Active Messages (SAM),* that encapsulate these operations in a portable way. *Elsie,* which currently runs on SPARC instruction sets, is an editor that modifies executables to calculate target execution time and simulate a parallel computer's memory system. *SAM* is a messaging library that supports parallel simulation.

Using the available and portable versions of *Elsie* and *SAM*, we ported the *Wisconsin Wind Tunnel II (WWT II)*—the successor to the original *Wisconsin Wind Tunnel (WWT)* [2]—to a range of platforms, including desktop workstations, a SUN Enterprise server, and a cluster of SPARCstations. All platforms currently use the same SPARC instruction set architecture.

Our analysis has shown several important results including that *WWT II*:

- achieves portability without sacrificing performance,
- shows good parallel efficiencies across a range of host platforms, and
- is a cost-effective parallel simulator.

In summary, WWT II demonstrates a technology for parallel simulation of target multiprocessors with up to hundreds of in-order processors executing user-level code. Other simulators, however, have evolved to simulate richer parallel targets: Rice RSIM [4] (user-level out-of-order processors), Stanford SimOS [5] (user/system out-of-order processors), and Virtutech SimICS (user/system in-order processors) [6]. These simulators run on uniprocessor hosts, and, therefore, are painfully slow simulating large target multiprocessors. A future simulation challenge is use WWT II-like parallel simulation technology for accelerating the simulation of multiprocessors with out-of-order processors executing user and system code.

2 Operations

In this section we discuss alternative implementations of four key operations that underlie parallel, discrete-event, direct-execution simulation. These operations help isolate host-specific features, which makes it easy to port and tune the performance of a parallel simulator. The first two operations—calculation of target execution time and simulation of features of interest—relate to direct execution, while the last two—communication of target messages and synchronization of host processors—relate to conservative-window, parallel, discrete-event simulation.

2.1 Calculation of Target Execution Time

To evaluate the performance of a proposed architecture, a simulator must calculate elapsed time on the target machine as well as mimic the target's function. In simulators that interpret every target instruction, calculating the target execution time is simple: the simulator updates a clock variable after simulating each instruction. However, direct execution simulators derive their speed from directly executing blocks of target instructions without simulator intervention. Invoking the simulator to update the clock variable after every target instruction would nullify this performance advantage.

The cost of updating the target clock variable can be reduced in two ways. First, instead of invoking the simulator, the target itself can maintain and update its own target clock variable. This implies that the target code must be augmented with extra code that updates the target clock. We call this *target clock instrumentation*. Second, we can update the variable less frequently by combining the updates for a sequence of instructions.

Target clock instrumentation can be done at four levels: source code [1], assembly code [7], object code, and executable [2]. Unfortunately, the first three approaches require source, assembly, or object code, which may be hard to obtain for vendor-provided libraries or commercial operating systems and databases. Executable modification removes this restriction because target clock instrumentation is added directly to the executable. However, executable modification introduces two problems. First, it is complex to implement because the executable editor must handle machine-specific details (e.g., fix branch addresses after the introduction of target clock instrumentation code). Second, like assembly or object code modification, executable modification makes the simulator dependent on a specific instruction set.

Fortunately, researchers have recently developed executable editing tools that allow users to traverse the control-flow graph of a target executable and introduce foreign code in an almost machine-independent fashion. These tools relieve the writers of executable editors from worrying about low-level machine-specific details. *WWT II* uses one such tool, called EEL [8], to build an executable editor, called *Elsie,* to perform the target clock instrumentation on target executables. *Elsie* is described in *Section 3.*

2.2 Simulation of Features of Interest

Researchers build simulators to study proposed parallel architectures. Hence, simulators must allow researchers to simulate features which may or may not be currently available in a parallel host. For example, the original *WWT* simulated a hardware, cache-coherent, shared-memory machine on the Thinking Machines (TMC) CM-5, which is a message-passing parallel machine.

In direct execution, simulating missing features requires the target to jump into the simulator on specific target instructions. For example, to simulate the target memory system, the target must transfer control to the simulator on some target loads and stores.

Researchers have used two approaches to simulate features missing in the host. The first approach uses hardware and software mechanisms available in the host to transfer control. For example, *WWT* and *Tapeworm II* [3] marked host memory blocks that are absent in the target cache or TLB (Translation Lookaside Buffer) with bad ECC. Accesses to memory blocks with bad ECC generated traps that were vectored to the simulator via the operating system. This allowed *WWT* and *Tapeworm II* to simulate cache and TLB misses, respectively. Unfortunately, this method is not easily portable because it requires operating system modification to catch the ECC traps. Additionally, most dynamically-scheduled processors are unlikely to support precise exceptions on ECC error. Without precise exceptions, a simulator will not be able to correctly simulate target cache misses.

The second approach is to replace target instructions with code segments that transfer control to the simulator. This approach is more general than the previous approach but can incur a performance penalty for its generality. For example, to simulate target cache misses, all loads and stores must check the target cache state, unlike the *WWT* approach in which the simulator checked the target cache block state only on target cache misses.

Replacing instructions with new code segments introduces problems similar to those faced by target clock instrumentation. Hence, our solution is similar. We augment *Elsie* to replace target instructions to simulate features missing in the host. In our case, this feature is the target memory system.

2.3 Communication of Target Messages

Communication is inherent in parallel simulation because target nodes exchange messages with one another. However, the most efficient method of communication differs radically across parallel computers. Typically, massively parallel processors (MPPs) use a native message passing library, COWs use sockets, and SMPs use shared memory. Consequently, communication code written for one machine cannot be easily ported to another machine. To overcome this problem, we have developed a simple messaging library called *Synchronized Active Messages (SAM)*, which abstracts away the communication primitives from the mechanisms and techniques used in implementation. *SAM*, which also handles processor synchronization, is described in Section 4.

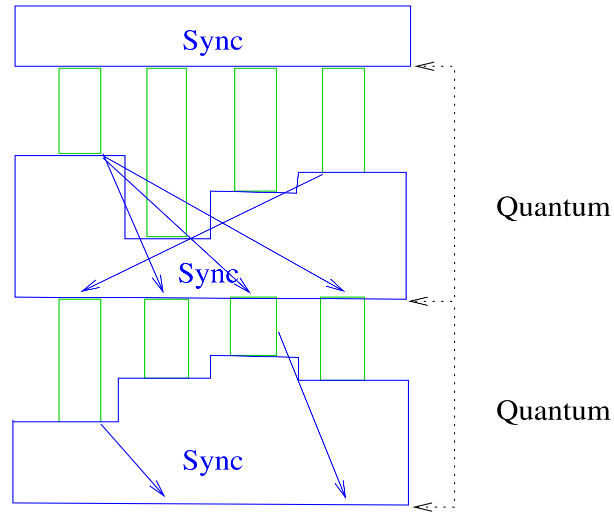

FIGURE 1. Graphical representation of quantum and messages sent for 4 processors. Blue regions are synchronization time while green areas are simulator processing times.

2.4 Synchronization of Host Processors

Parallel, discrete-event simulation that uses the conservative time bucket synchronization method [9] must rapidly synchronize host processors. In this method, target execution is broken up into lock-step intervals called *quanta* as shown in Figure 1. Target messages sent during one quantum can only affect target state in subsequent quanta. This is accomplished by setting the quantum length based upon the time necessary for a message to be delivered in the target (this is a lower bound so it is conservative). Since messages are guaranteed to be delivered before the start of the next quantum, the simulator makes sure that the receiving target is aware of the message before it can have any effect on the outcome of the target program.

Conservative-window, parallel, discrete-event simulation imposes three synchronization requirements. First, host processors must be able to detect when target execution reaches the end of a quantum. Second, when a quantum expires, host processors must synchronize among themselves using a barrier and calculate the duration of the next quantum interval. The duration of the next quantum interval is often calculated as the sum of the minimum target execution time across all host processors (conventionally called a reduction) and a fixed quantum length (e.g., 100 target processor cycles). The former represents the fact that the simulator often knows that all targets will not be interacting for a period of time so it can extend the next quantum. The latter represents the minimum time for message transmission once a message has been sent and is the minimum time for two targets to interact. Third, host processors must ensure that all messages sent in a quantum are received and processed before the beginning of the next quantum. This is shown in Figure 1 by the fact that messages sent are received at the end of the synchronization. A global reduction of the difference between the number of messages sent and received will be zero once delivery is complete. This allows a host processor to complete reception of all target messages before beginning the next quantum. The following three paragraphs discuss each of these three synchronization requirements.

There are two ways to detect the end of a quantum. First, the simulator can check for quantum expiration on each entry into the simulator. This approach works well if the target frequently returns control to the simulator. Because *WWT II* simulates every load and store, we use this approach. Second, if the simulator is invoked less frequently, global synchronization will be deferred and consequently other target nodes may be delayed. In this case, we can modify the target executable to check the target execution time more frequently (e.g, on target clock updates) and invoke the simulator if a quantum has expired. This method is more robust, but introduces additional overhead.

Different parallel computers provide different degrees of hardware support for barrier synchronization and reductions. For example, the TMC CM-5 supports both hardware barriers and hardware reductions, while the Cray T3E supports only hardware barriers. In contrast, the SUN Enterprise E6000 and our COW connected with an off-the-shelf network have no hardware support for either; hence, these machines must implement both in software. Lack of hardware support for barriers and reductions can degrade the performance of conservative-window, parallel, discrete-event simulation, particularly when the quantum intervals are short.

Most parallel computers do not provide hardware support to determine if all messages injected into a host network have been drained (the TMC CM-5 is a notable exception). However, there are a variety of ways of doing this in software. For example, we can collect acknowledgments for every message injected into the network. Alternatively, we can confirm message delivery at the end of the quantum, combining this operation with the barrier synchronization.The *SAM* package, described in Section 4, implements the necessary functionality while allowing for portability.

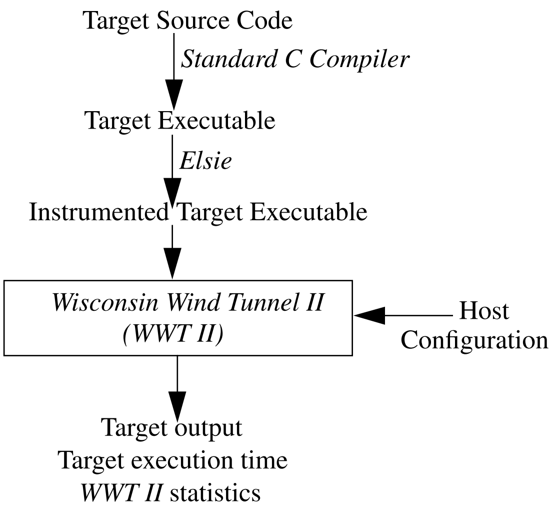

FIGURE 2. Relationship of Elsie to *WWT II*.

3 Elsie

Elsie modifies target executables that run on *WWT II* (Figure 2) to achieve the calculation of target execution time and simulate features of interest. Like other executable editors for direct-execution simulators, *Elsie* adds instrumentation to calculate the target's execution time and to simulate the target's memory system. Surprisingly, *Elsie* can be written in an almost machine-independent fashion for three reasons. First, *Elsie* uses the EEL executable editing library [8], which hides most details of modifying executables. EEL provides operations that *Elsie* uses to traverse a target executable's control-flow graph and to add *code snippets*. Snippets contain machine-specific instructions, which *Elsie* adds to edges in a control-flow graph to track the target's execution time. *Elsie* also replaces target memory instructions (e.g., loads and stores) with snippets that jump into the simulator, which simulates the target memory system. Second, there are few machine-dependent snippets and they are small. The eight mandatory snippets all contain four or fewer instructions each. Consequently, only small portions of machine-specific code must be rewritten to port *Elsie* to a different instruction set. The small number of machine-specific instructions needed make porting *Elsie* even easier. The current version of *Elsie* only runs on the SPARC V8 instruction set. Modification for other instruction sets involves describing the properties of the new processor and using a version of EEL aimed at this machine. For example, the detailed timings for the new instruction set are needed.

The introduction of instrumentation code to jump into the simulator to simulate every memory instruction increases *WWT II*'s overhead compared to *WWT* or *Tapeworm II*. *WWT* and *Tapeworm II* have low overhead because they directly execute memory instructions that hit in the target cache (see Section 2.2). *WWT II* reduces this overhead by providing a fast path for loads and stores that hit in the target cache [10]. Normally, on a load or store, the simulator translates the virtual address to the physical address using the target TLB, indexes into the cache, finds the appropriate cache block through a tag match, checks the state of the cache block, and, on a cache hit, loads or stores a value from or to the cache block. Instead, in the fast path, *WWT II* maintains pointers to all valid target cache blocks in each target TLB entry. Thus, if a load or store hits in the target cache, *WWT II* can directly find the block on a target TLB access.

4 Synchronized Active Messages (SAM)

Synchronized Active Messages (SAM) provides an architecture-neutral programming model that unifies a parallel host's communication and synchronization operations for a quantum-based, parallel, discrete-event simulation. This achieves the communication of target messages and synchronization of host processors in the simulator.

SAM, by design, is very simple so that it can be implemented easily across a wide range of parallel machines. *SAM* provides three main primitives: *SAM_Send_Msg*, *SAM_Bcast_Msg*, and *SAM_Sync*. Host processors communicate using SAM_Send_Msg, calculate the next quantum duration using SAM_Bcast_Msg (that is, via broadcast messages), and synchronize using SAM_Sync. Like Active Messages, a *SAM* message contains a virtual address of a handler that will be called at the receiving host processor. However, unlike active messages, *SAM* does not guarantee message reception until SAM_Sync completes. When SAM_Sync returns, *SAM* guarantees that all messages have been received and processed (so that messages have been scheduled for the next quantum) by calling the corresponding handlers. By supplying the appropriate handler, *SAM* can be utilized to calculate the next quantum duration via message broadcasts for simplicity, and thereby avoids a separate reduction interface, such as the one in the TMC CM-5.

Currently, *SAM* runs on three platforms: an SMP, a Cluster of Workstations (COW), and a Cluster of SMPs (COW/SMP). Each implementation is optimized to the platform's underlying communication substrate.

The *SAM* SMP implementation is straightforward because our SMP (SUN E6000) supports efficient low-latency communication over the memory bus. *SAM* allocates a shared-memory segment and for each process in the parallel program *SAM* sets up two sets of mailboxes in shared memory—destination mailboxes and source mailboxes. A process' destination mailbox is used by another process to send a point-to-point message to this process. Each message is explicitly copied into the destination mailbox because two process' only share the segment containing the mailboxes and not the entire address space. Mutual exclusion of destination mailbox is ensured through an atomic fetch-and-add operation. A process uses its own source mailbox to enqueue broadcast messages. We do not enqueue a broadcast message in the destination mailboxes because that would create multiple copies of the same message. Finally, when a process calls SAM_Sync, *SAM* drains a process' own destination mailboxes and checks all other process' source mailboxes for broadcast messages. Subsequently, *SAM* calls the handlers corresponding to each message and returns control to the simulator.

The COW implementation of *SAM* is more complex. Analysis of the COW's communication characteristics reveals that message overhead is high (26 µsecs under SunOS 5.5 with Myricom switches - see Table 1) so minimizing the number of messages is very important. *WWT II* sends few messages (two or less, per processor) that are small (80 or fewer bytes) in a quantum. Multiple messages occur on a host due to having multiple targets on a host and because protocol processing on a single target can involve multiple messages.

Taking these characteristics into account, we implement SAM_Sync through a software butterfly-style message exchange pattern. The number of stages is logarithmic in the number of processors, thereby reducing the number of messages on the critical path. We further reduce the number of messages by *piggybacking* the target messages from the current quantum and the data needed to determine the next quantum length on the butterfly synchronization. As *WWT II* sends very few short messages in each quantum, the total cost of the butterfly is not substantially increased over the synchronization cost, even though our piggybacking scheme sends all data to all host processors (Figure 3).

The COW/SMP implementation combines the COW and SMP implementations. The host processors within an SMP first exchange their messages. Then one pre-designated host processor in each SMP node exchanges messages with other host processors following the same piggybacked butterfly as shown in Figure 3. Finally, host processors within an SMP syn-

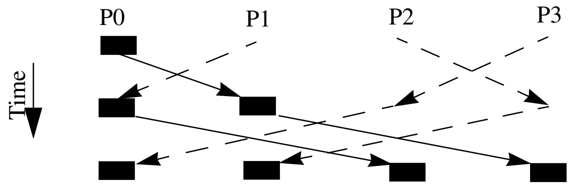

FIGURE 3. ***SAM*** **implementation for a COW. P0, P1, P2, and P3 denote host processors. Dark boxes represent data - here only P0 sends a message. Solid lines represent the flow of synchronization messages with data (piggybacking). Dotted lines represent flow of synchronization messages without data.**

Parallel Machine	Host Processor	Inter-Host Communication		N	P
		Memory Bus	Network		
SMP (16-processor SUN E6000)	250 MHz UltraSPARC	83.5 MHz, 256-bit wide split-transaction	N/A	1	16
COW (uniprocessor SPARC-server20)	66 MHz Hyper-SPARC	N/A	First generation version 2 Myricom Myrinet switches	16	16
COW/SMP (dual-processor SPARC-server20)	66 MHz Hyper-SPARC	50 MHz, 64-bits wide sequential	First generation version 2 Myricom Myrinet switches	8	16

TABLE 1. The host systems used. N is number of nodes and P is the total number of host processors.

chronize locally to ensure that the pre-designated processor has drained all messages from the network.

5 Methodology

This section describes our experimental framework, *WWT II*, and the target architecture and benchmarks we use for this study. Table 1 shows our three different parallel machine configurations. Figure 4 shows a graphical representation of the three types of machines used. The COW/SMP is the same as the COW, except that each node has two processors, instead of one. We use 16 COW nodes and 8 dual-processor COW/SMP nodes to equalize the number of host processors in the COW and COW/SMP configurations.

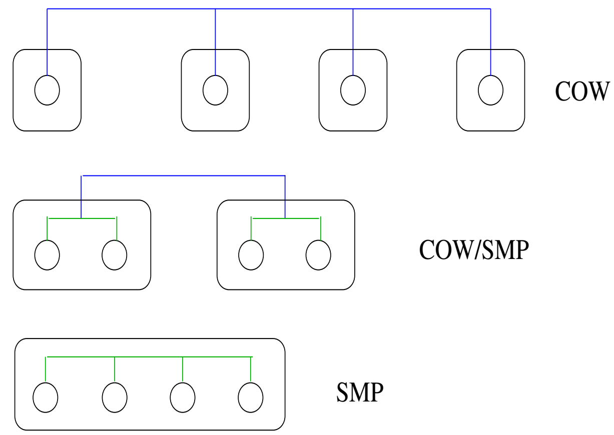

FIGURE 4. Graphical representation of the different machine configurations for 4 processors. Green represents a bus and blue represents a network.

Benchmark	Source	Description	Input Data Set
FFT	SPLASH-2	complex Fast Fourier Transform	2^{16} points
LU	SPLASH-2	LU factorization	order 512 matrix, order 16 blocks
radix	SPLASH-2	Integer sort	256K keys, 1K radix
tomcatv	*WWT* parallelization of SPEC	Mesh Generation with Thompson's solver	order 512 matrices, 4 iterations
water-sp	SPLASH-2	water molecule simulation	4K molecules, 3 steps

TABLE 2. Target benchmarks and the corresponding input data sets we used for our experiments.

For this study, we have chosen an S-COMA [11] shared-memory machine as our target architecture. Each target node has a single processor and a 256 kilobyte processor cache. Hardware coherence is implemented through a full-map directory protocol. Each host processor in *WWT II* simulates one or more target nodes. For example, for a 256-node target, an 8-processor *WWT II* configuration simulates 32-target nodes per host processor.

Table 2 shows the five target benchmarks and corresponding input data sets we used for our study.

In all our measurements we report the time it took *WWT II* to execute only the parallel portion of each

Bench-mark	Number of Host Proces-sors	Speedup		
		SMP	COW	COW/SMP
LU	1	1	1	1
	2	1.8	1.7	1.6
	4	3.1	2.6	2.5
	8	4.7	3.5	3.4
	16	5.4	3.6	3.5
tomcatv	1	1	1	1
	2	1.8	1.8	1.6
	4	3.3	2.9	2.7
	8	5.1	4.0	3.8
	16	5.8	4.3	4.1

TABLE 3. Parallel speedups across platforms for *WWT II* on a 32 node target system.

target benchmark. We assume SPARC V8 instruction set for our target benchmarks so all of our host processors are SPARC V8 compatible. Additionally, since *WWT II* takes the same path through the target executable, all our target executable runs report *exactly* the same target execution cycles, irrespective of which of our three platforms ran the experiments. *WWT II* takes the same path through the executable because we impose a strict ordering of events. This control over the experimental framework is essential to effectively characterize *WWT II*'s performance across our three platforms.

6 Performance Analysis

We now present results obtained from running *WWT II*. First we show its parallel performance and then we discuss its cost-effectiveness.

6.1 Parallel Performance

This section describes the performance of *WWT II* by looking at the host's parallel speedup (uniprocessor time / parallel time). This metric shows the effectiveness of utilizing the parallel simulation capability of *WWT II*.

We first look, in Table 3, at how the performance compares across our three parallel hosts. We only show selected benchmarks and a limited number of targets because they exemplify the results and are small enough to avoid virtual-memory thrashing on a single COW node. The data shows that *WWT II* achieves reasonable speedups for this modest number of targets across all three platforms. As will be shown below, the performance increases as larger simulations are performed. To give an idea of the absolute run times of *WWT II*, the 16 host processor run time for tomcatv is 1.8 and 9.4 minutes for the SMP and

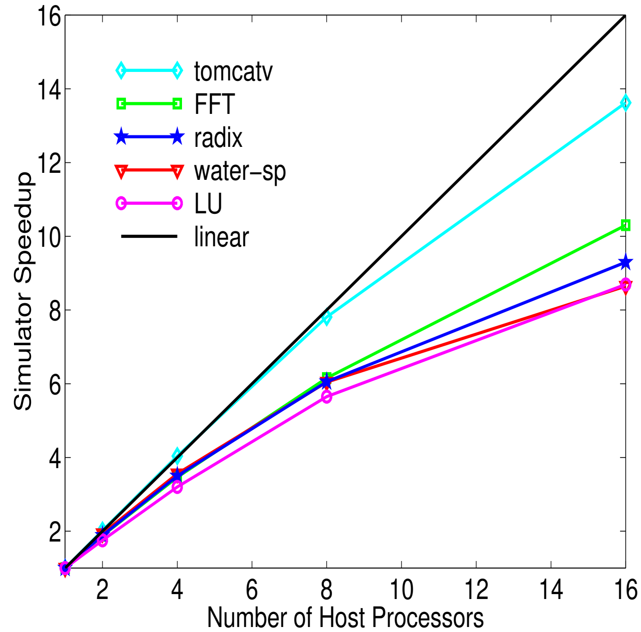

FIGURE 5. Simulator speedups on SMP across benchmarks for 256 targets.

COW, respectively. These show that parallel execution of simulations can perform in time frames which make their usage practical for many applications. When comparing between platforms, the speedups are better on the SMP as the number of host processors increases. This indicates the faster communication on the SMP yields better parallel performance.

We now turn to SMP results because the large memory available for any number of processors allows for running large memory targets across the full range of host processors. Without this ability we could not run the large parallel jobs on a single processor to determine speedups. Figure 5 shows the simulator achieves good speedups for up to 16 hosts across all benchmarks with 256 targets. At 16 hosts the speedups range from 8.6 to 13.6 for an efficiency of 54% to 85%. Also note that the speedup curves are monotonically increasing so that greater parallelism reduces the time for a given simulation. Figure 6 shows the effect of varying the number of targets. As can been seen, increasing the number of targets increases the simulator speedups. This effect is seen on all the benchmarks and tomcatv was shown because it has the largest effect. This trend is helpful since larger simulations, which require greater uniprocessor run times, will achieve better parallel performance. An important factor in the increased efficiency is the reduction in idle time due to improved load balancing as the number of targets per host is increased. Once a host has finished work for all of its targets in the current quantum, this host idles until the slowest host completes and enters the synchronization as shown in Figure 1. As the number of targets per host

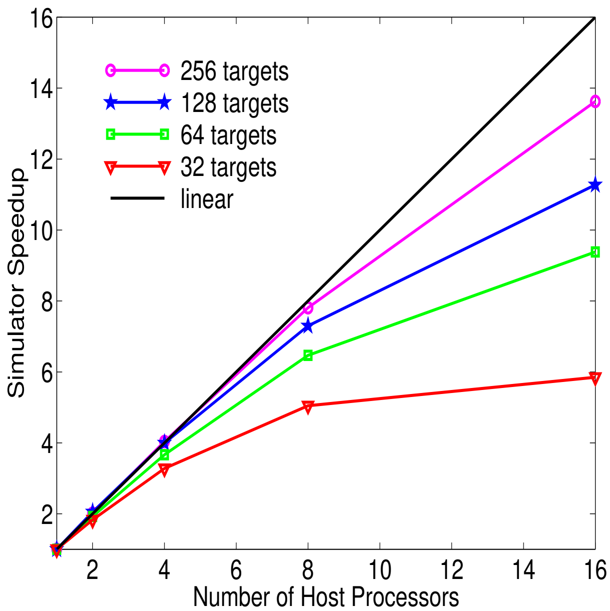

FIGURE 6. Simulator speedups on SMP for tomcatv for varying number of targets.

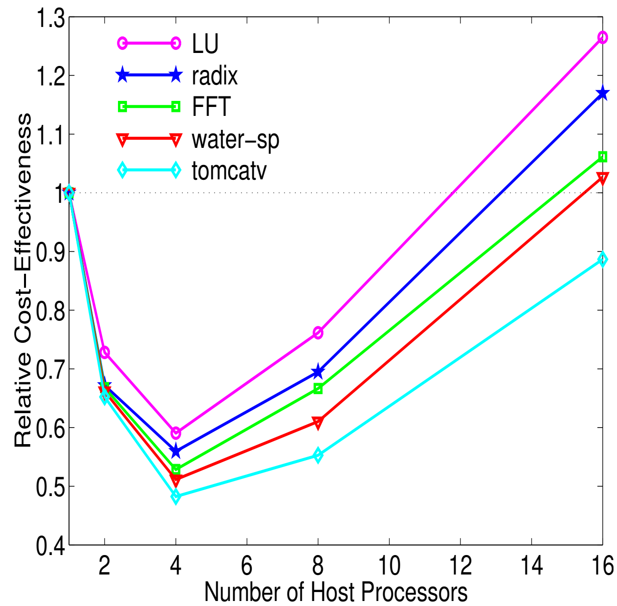

FIGURE 7. Relative cost-effectiveness across benchmarks for 64 targets and 64 Megabytes/ target

increases, the deviation from the average decreases so the idle time is decreased [12].

6.2 Cost-Effectiveness

The previous section shows that parallelism improves simulator run times for a given simulation. However, this does not demonstrate that the use of parallelism is cost-effective, i.e., it is cheaper to run a parallel simulation on N host nodes than N sequential simulations. To evaluate this question we need to specify the cost of the various host systems used. We define the cost to be the purchase price of the smallest system that could run the simulation in question. Thus, a simulation run on 4 hosts that needs 1 Gigabyte of memory would be the cost of the smallest box that has 4 processors and 1 Gigabyte of memory. A general discussion of cost-effectiveness can be found in [13].

An important component in the cost of a computer is the memory. As part of our analysis of *WWT II* we determined the memory usage (in Mbytes) of the simulator which is given by

$$M_{sim} = 1.26 * (\#\ hosts) + 1.97 * (\#\ targets)$$

$$M_{target} = target\ memory * (\#\ targets)$$

$$M = M_{sim} + M_{target}$$

where M_{sim} is the memory taken up by the simulator on all hosts without the target program, M_{target} is the memory for all targets, and M is the total memory used in all hosts. The cost of the SMP system in thousands of US dollars is given by

$$C = base + 9 * [(max(\lceil P/2 \rceil, \lceil M/512 \rceil)] + 16 * P + 0.0174 * M$$

where P is the number of host processors. base is 17.5 if $P \le 6$, 48.5 if $7 \le P \le 14$, and 181.5 if $15 \le P \le 30$. These cost figures were taken from a Sun price list dated 20 May 1997. From the cost and run time of the simulation we can define the cost-effectiveness to be

$$CE(P) = C(P) * time(P)$$

where a lower value of cost-effectiveness is better. To determine the cost-effectiveness of a parallel simulator it is useful to define the relative cost-effectiveness of running the simulation on P processors versus 1 processor. This is given by

$$RCE(P) = CE(P) / CE(1)$$

where values less than one mean it is cheaper to run on P processors than 1 processor.

Figure 7 shows the relative cost-effectiveness across the benchmarks. In these results it is assumed that each target uses 64 Mbytes of memory and the speedups are those achieved when the datasets in Table 2 were run. We chose these parameters because they clearly demonstrate the tradeoff involved. It is seen in Figure 7 that parallel simulation is cost-effective for these benchmarks, simulator, and cost parameters until 16 host CPUs. At this point all but one benchmark is no longer cost-effective. The minimum at 4 host processors shows the point of lowest cost. Thus, for these parameters, the cheapest simulation is on 4 hosts for all of the benchmarks. At this point the cost of parallel simulation is 48% to 59% of the cost

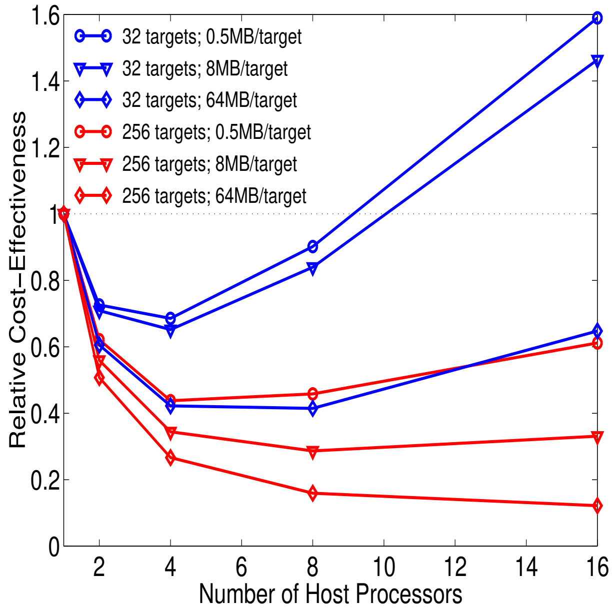

FIGURE 8. Relative cost-effectiveness where the number of targets and memory per target is varied. The memory is given as Megabytes per target.

of the uniprocessor simulation. This means that not only is parallel simulation faster but it is around half the cost.

Figure 8 shows the effect on cost-effectiveness of varying the number of targets and memory per target for the tomcatv application. Again we assume that the speedups measured from the actual benchmarks are unchanged as the amount of memory is varied. It is seen that as the number of targets is increased the relative cost-effectiveness is improved. This is consistent with the previous result that speedups are improved as number of targets increases. It is also seen that as the memory per target (and thus total memory) is increased, the relative cost-effectiveness is improved. Both of these trends, seen across the benchmarks, are consistent with previous results [12,13]. For the largest benchmark considered in Figure 8—256 targets and 64 Megabytes per target—the relative cost-effectiveness decreases as the number of host processors is increased. For this simulation, 16 host processors is the most cost-effective with a cost of 12% of the uniprocessor and it is an open question where the optimal number of host processors lies. At the other extreme of 32 targets and 0.5 Megabytes per target the graph looks similar to those seen in Figure 7. Here 4 host processors is most cost-effective and for 16 hosts the cost-effectiveness is worse than the uniprocessor case. These results clearly show that parallel simulation is cost-effective including sufficiently large simulations for large numbers of host processors.

7 Conclusions

This paper examined four key operations that underlie parallel, discrete-event, direct-execution simulation. These four operations are: calculation of target execution time, simulation of features of interest, communication of target messages, and synchronization of host processors.

We encapsulated portable implementations of these four operations in two tools called *Elsie* and *Synchronized Active Messages*. Using these tools, we easily and successfully ported the *Wisconsin Wind Tunnel II (WWT II)*—a parallel, discrete-event, direct-execution simulator—across a wide range of SPARC platforms, including desktop workstations, a SUN Enterprise server (SMP), a cluster of workstations (COW), and a cluster of symmetric multiprocessing nodes (COW/SMP). The speedups maintained across the SMP, COW, and COW/SMP demonstrate the effectiveness of our techniques for portability.

Analysis of *WWT II* shows it has good parallel performance and is cost-effective. Specifically, *WWT II* obtained speedups between 8.6 and 13.6 for 256 targets on 16 SMP host processors on the benchmarks studied. Furthermore, we showed that speedups improve as the number of targets per host is increased. In terms of cost-effectiveness, we saw large simulations using all 16 SMP host processors minimized the cost to 12% of the uniprocessor cost. For smaller simulations using 4 SMP host processors minimized the cost and reduced it to 48% to 59% of the uniprocessor cost.

In summary, WWT II demonstrates a technology for parallel simulation of target multiprocessors with up to hundreds of in-order processors executing user-level code. Other simulators eschew parallelism in favor of sequential simulation but can evaluate richer targets, such as multiprocessors with out-of-order processors executing user and system code. A future simulation challenge is use WWT II-like parallel simulation technology for accelerating the simulation of these richer targets. Information on obtaining *WWT II* is available at the URL http://www.cs.wisc.edu/~wwt/wwt2/.

Acknowledgments

This work was supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, NSF Grants CCR-9101035, MIP-9225097, and MIPS-9625558, NSF PYI/NYI Awards CCR-9157366, MIPS-8957278, and CCR-9357779, DOE Grant DE-FG02-93ER25176, University of Wisconsin Graduate School Grant, Wisconsin Alumni Research Foundation Fellowship and donations from Digital Equipment Corporation, IBM, Sun Microsystems, Thinking Machines Corporation, and Xerox Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the University of Wisconsin Graduate School. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

References

[1] [1] R. Covington, S. Madala, V. Mehta, J. Jump, and J. 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.

[2] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. 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*, pages 48–60, May 1993.

[3] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest. Tapeworm II: A new method for measuring os effects on memory architecture performance. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 132–144, October 1994.

[4] P. Ranganathan, V. S. Pai, and S. V. Adve. Rsim: An execution-driven simulator for ilp-based shared memory multiprocessors and uniprocessors. In *3rd Workshop on Computer Architecture Education*, 1997.

[5] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the simos machine simulator to study complex computer systems. *ACM Trans. on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[6] P. S. Magnusson et al. Simics/sun4m: A virtual workstation. In *Proceedings of Usenix Annual Technical Conference*, June 1998.

[7] H. Davis, S. R. Goldschmidt, and J. 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.

[8] J. R. Larus and E. Schnarr. Eel: Machine-independent executable editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[9] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[10] A. R. Lebeck and D. A. Wood. Active memory: A new abstraction for memory-system simulation. In *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 220–230, May 1995.

[11] E. Hagersten, A. Saulsbury, and A. Landin. Simple COMA node implementations. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, January 1994.

[12] B. Falsafi and D. A. Wood. Modeling cost/performance of a parallel computer simulator. *ACM Transactions on Modeling and Computer Simulation*, 7(1), January 1997.

[13] D. A. Wood and M. D. Hill. Cost-effective parallel computing. *IEEE Computer*, 28(2):69–72, February 1995.