Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator

Shubhendu S. Mukherjee*, Steven K. Reinhardt†, Babak Falsafi*, Mike Litzkow*, Steve Huss-Lederman*, Mark D. Hill*, James R. Larus*, and David A. Wood*

*Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, Wisconsin 53706-1685 USA
URL: http://www.cs.wisc.edu/~wwt
Email: wwt@cs.wisc.edu

†EECS Department
University of Michigan
1301 Beal Ave
Ann Arbor, MI 48109-2122 USA
URL: http://www.eecs.umich.edu/~stever
Email: stever@eecs.umich.edu

Abstract

The design of future parallel computers requires rapid simulation of target designs running realistic workloads. These simulations have been accelerated using two techniques: direct execution and the use of a parallel host. Historically, these techniques have been considered to have poor portability. This paper identifies and describes the implementation of four key operations necessary to make such simulation portable across a variety of parallel computers. These four operations are: calculation of target execution time, simulation of features of interest, communication of target messages, and synchronization of host processors.

Portable implementations of these four operations have allowed us to easily run the Wisconsin Wind Tunnel II (WWT II)—a parallel, discrete-event, direct-execution simulator—across a wide range of platforms, such as desktop workstations, a SUN Enterprise server, a cluster of workstations, and a cluster of symmetric multiprocessing nodes. We plan to release WWT II in August, 1997. We also plan to port WWT II to the IBM SP2.

We find that for two benchmarks, WWT II demonstrates both good performance and good scalability. Uniprocessor WWT II simulates one target cycle of a 32-node target machine in 114 and 166 host cycles respectively for the two benchmarks on a SUN UltraSPARC. Parallel WWT II achieves speedups between 4.1-5.4 on 8 host processors in our three parallel machine configurations.

1 Introduction

Software simulation is an important technique for studying computer architectures ranging from microprocessors [4, 5] to parallel computers [3, 17, 24]. Simulation speeds up design by enabling architects to evaluate computers without building hardware prototypes. However, simulating big problems—parallel machines with realistic workloads—requires large amounts of computation and memory. Two techniques, direct execution and parallel simulation, make this approach feasible.

In direct execution [6], 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 [6]. However, computing the execution time for dynamically scheduled processors with non-blocking caches is an open problem [15].

Parallel simulation of a parallel computer further speeds simulation by exploiting the parallelism inherent in the target parallel computer and the parallel host's large memory to hold the simulator's working

This work is 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.

set without paging. The advent of low-cost parallel computers, such as symmetric multiprocessors (SMPs) and clusters of workstations (COWs), make parallel simulation very attractive.

Unfortunately, parallel, discrete-event, direct-execution simulators are complex pieces of software that can be difficult to build and port. Portability is a desirable goal because of the wide range of processor architectures and parallel computers. In part, parallel, discrete-event, direct-execution simulators are not portable because they rely on machine-specific features. Direct-execution simulators 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 [17, 22] also modify the operating system to detect target cache misses. Similarly, 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.

The main contribution of this work is to identify and implement these four operations in a fashion that minimizes the dependence of a parallel simulator on host-specific features. Section 2 examines alternative implementations of these four operations. Section 3 and Section 4 describe two tools, called *Elsie* and *Synchronized Active Messages (SAM)* that encapsulate these operations in a portable way. *Elsie* 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 *Elsie* and *SAM*, we ported the *Wisconsin Wind Tunnel II (WWT II)*—the successor to the original *Wisconsin Wind Tunnel (WWT)* [17]—to a wide range of platforms, including desktop workstations, a SUN Enterprise server, and a Cluster of SPARCstations. We are also porting *WWT II* to the IBM SP2. We find that *WWT II* shows excellent to modest performance on our different platforms (Section 5 and Section 6). In Section 7 we present our conclusions.

We plan to release *WWT II* in August, 1997. The exact release date and additional information about *WWT II* will be available from the URL: http://www.cs.wisc.edu/~wwt/wwt2/.

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 (Section 2.1) and simulation of features of interest (Section 2.2)—relate to direct execution, while the last two—communication of target messages (Section 2.3) and synchronization of host processors (Section 2.4)—relate to quanta-based, parallel, discrete-event simulation.

2.1 Calculation of Target Execution Time

Simulation is generally uninteresting without a target's execution time. In pure software simulation, which interprets every target instruction, calculating a target's execution time is simple. The simulator updates a clock variable after simulating each target instruction. Unfortunately, returning control to the simulator after every instruction defeats the purpose of direct execution. This is because direct execution speeds simulation by directly executing blocks of target instructions on host hardware without any simulator intervention. Consequently, jumping into the simulator after every instruction to update the target clock can be expensive for direct execution.

The cost of updating a target clock variable can be reduced in two ways. First, instead of updating the target clock after every instruction, we can update it at edges of basic blocks in a routine's control flowgraph. Ball [1] showed how to optimize this by updating a counter, such as the target clock, only on a subset of edges. Second, instead of jumping into 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.

Target clock instrumentation can be done at four levels: source code [6], assembly code [3, 7], object code, and executable [17]. 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. 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. Consequently, calculating the target execution time via executable modification has been considered to have poor portability.

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. In Section 3, we show how we used one such tool, called EEL [13], to build an executable editor, called *Elsie,* to perform the target clock instrumentation on target executables.

2.2 Simulation of Features of Interest

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

In direct execution, to simulate features missing in a host, the target often needs the ability 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 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 simulate specific target features. For example, *WWT* and *Tapeworm II* [22] marked host memory blocks that are absent in the target cache or TLB 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. However, this method can pay 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* (Section 3) 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 native communication support differs radically across parallel computers. Massively Parallel Processors (MPPs) are programmed with explicit message-passing, COWs with sockets, and SMPs with shared memory. Consequently, the 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. This allows us to easily port *SAM* across different parallel computers.

2.4 Synchronization of Host Processors

Parallel, discrete-event simulation (PDES) that uses the conservative time bucket synchronization method [21] must rapidly synchronize host processors. In this method, target execution is broken up into lock-step intervals called quanta (Figure 1). Target messages sent during one quantum can only affect target state in subsequent quanta.

Quanta-based PDES imposes three synchronization requirements. First, host processors must be able to determine that a quantum has expired, and thus synchronize with the target node. 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). Third, host processors must ensure that all messages sent in a quantum are received and processed before the beginning of the next quantum. This allows a host processor to schedule reception of all target messages at the beginning of a quantum. The following three paragraphs discuss each of these three synchronization requirements.

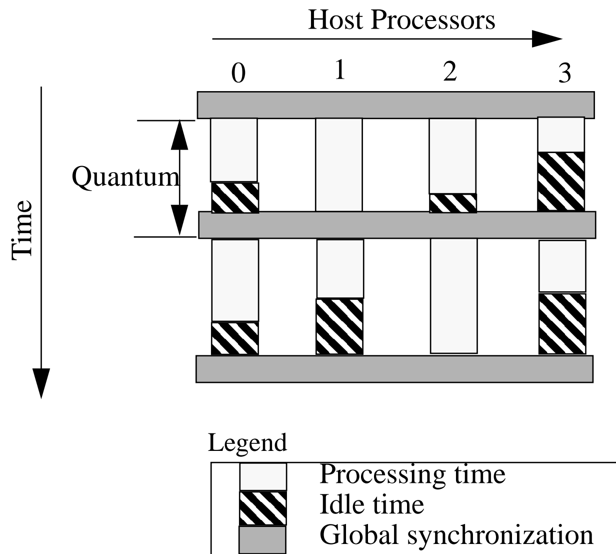


FIGURE 1. This figure shows how quanta-based PDES simulates a parallel target machine. Each host processor directly executes target instructions and simulates target events (processing time), enters an idle phase in which it spins for the global synchronization to begin, and then resumes execution after the synchronization.

There are two ways to determine if a quantum has expired. First, the simulator can check quantum expiration on specific entry points into the simulator. This approach is very efficient if the target frequently returns control to the simulator (e.g., on every load and store). *WWT II* uses this approach. However, this approach can hurt performance if simulated features do not recur frequently. This is because target nodes may not synchronize frequently enough, and consequently, target nodes waiting for messages from other target nodes may not make progress. Second, we can modify a target executable to check the progress of target execution time at specific points (e.g, on target clock updates) and jump into the simulator if a quantum has expired. This is a more robust method, but introduces additional overhead compared to the first method.

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 E5000 or a COW connected with Myricom Myrinet switches 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 quanta-based PDES, particularly when the quantum intervals are short.

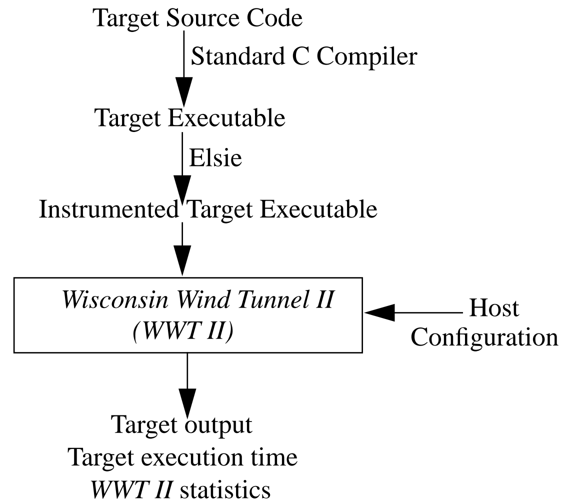


FIGURE 2. This figure shows how Elsie is related to *WWT II*.

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 *piggyback* all messages over a software barrier (e.g., our COW implementation of synchronization and communication described in Section 4), which would guarantee message reception before the beginning of a new quantum.

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 [24, 17], *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 [13], 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. Finally, EEL itself runs on different instruction-set architectures, such as the SPARC and the IBM RS/6000. Hence, porting *Elsie* from a SPARC to an IBM RS/6000 only requires rewriting machine-dependent snippets.

The introduction of code snippets to target executables incurs only a modest increase in the size of the target executable's text segment. The instrumentation overhead (measured in number of instructions added statically) for target execution time and memory instruction simulation are 68% and 70% respectively, averaged across our target benchmarks (Table 2). This is comparable to the instrumentation overhead introduced by MIT Proteus [3] or Stanford Embra [24].

In practice, *Elsie*'s instrumentation overhead in terms of actual execution time can be even lower for two reasons. First, the instrumentation overhead is perfectly parallelizable, because *Elsie* does not add extra instrumentation code for parallel simulation. Second, EEL can hide instrumentation overhead by scheduling instrumentation instructions in idle superscalar execution slots [19].

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. 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. The fast path optimization speeds up *WWT II* by 8% (averaged across our target benchmarks) on a SUN Enterprise E5000 machine running a 32-node target on a uniprocessor host. All our results (Section 6) assume this last optimization.

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 quanta-based, parallel, discrete-event simulation. This achieves the communication of target

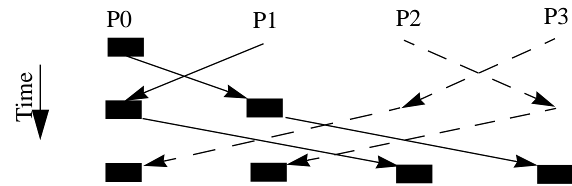


FIGURE 3. *SAM* implementation for a COW. P0, P1, P2, and P3 denote host processors. Dark boxes represent data - here only P0 sends a message. Solid lines represent the flow of synchronization messages with data (piggybacking). Dotted lines represent flow of synchronization messages without data.

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 [23], 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. *SAM* calculates 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 E5000) 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

Parallel Machine	Host Processor	Inter-Host Communication		N	P
		Memory Bus	Network		
SMP (8-processor SUN E5000)	167 MHz UltraSPARC	83.5 MHz, 256-bit wide	N/A	1	8
COW (uniprocessor SPARCserver20)	67 MHz HyperSPARC	50 MHz, 64-bits wide	Myricom Myrinet switches	8	8
COW/SMP (dual-processor SPARCserver20)	67 MHz HyperSPARC	50 MHz, 64-bits wide	Myricom Myrinet switches	4	8

TABLE 1. This table shows the experimental configuration for our parallel machines. N = number of nodes and P = total number of host processors in each machine. Each node of our COW is equipped with two processors. However, for the COW runs we only use one processor, whereas for the COW/SMP runs we use both the processors in a COW node.

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 minimizing latency and number of messages are very important. *WWT II* sends few messages (two or less, per processor) that are small (80 or fewer bytes) in a quantum.

The standard model of the latency to send a message from one process to another on the COW is:

end-to-end message latency = $\alpha + \beta * b$,

where α is the message latency for a zero-length message, β is the incremental cost per byte, and b is the number of bytes in a message. We use the Berkeley Active Messages as the native communication layer on the COW. For this messaging layer, α is 26 μsecs and β is 0.071 μsecs/byte, and the ratio of α to β is 366 bytes. This shows that communication latency of short messages is dominated by α, which means that we should minimize the number of messages. The high α to β ratio means that a modest increase in message size does not significantly affect message latency.

Target Benchmark	Description	Input Data Set
FFT	Performs Fast Fourier Transform	Points = 2^{10}
Radix	Performs integer radix sort	Keys = 256K Max Key = 512K Radix = 1K

TABLE 2. This table shows the target benchmarks and the corresponding input data sets we used for our experiments.

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 synchronize 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. Our SMP is a SUN E5000 machine with eight 167 MHz UltraSPARC processors connected with a split-transaction memory bus called the UltraGigaplane [20]. The COW nodes have 67 MHz HyperSPARC processors and are connected with Myricom Myrinet switches [2]. The COW/SMP is the same as the COW, except that each node has two processors, instead of one. Each COW node has a 50 MHz in-order memory bus called the MBus [12]. We use 8 COW nodes and 4 dual-processor COW/SMP nodes to equalize the number of host processors in the COW and COW/SMP configurations.

Differences	***WWT***	***WWT II***
Host Platforms	TMC CM-5	Workstation, SMP, COW, COW/SMP
Target Architectures	CC-NUMA	CC-NUMA, S-COMA, software DSM, SMP, and Tempest (active messages and shared memory)[11][a]
Memory Bus	Contention-free	Detailed simulation of a coherent memory bus
Network	Optional Network Simulation	Network is not modeled, but network contention is modeled at the network interfaces
Source Language	C	C++ (primarily)
Number of non-blank, non-comment lines of simulator source code	~16,000	~30,000

TABLE 3. This table shows the difference between *WWT* and *WWT II*. CC-NUMA = Cache-Coherent Non-Uniform Memory Architecture. COMA = Cache-Only Memory Architecture. DSM = Distributed Shared Memory. SMP = Symmetric Multiprocessor. The lines of source code reported above does not include the lines of source code for the executable editors, SAM, or the target benchmark.

a. A subset of these target architectures will be made available in the final distribution of *WWT II*.

WWT II is the successor to *WWT*, but is more detailed and flexible compared to *WWT*. Table 3 lists the differences between *WWT II* and *WWT*. We have already used *WWT II* for several research efforts [9, 14, 18, 16].

For this study, we have chosen a 32-node S-COMA [10] 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 32-node target, an 8-processor *WWT II* configuration simulates 4-target nodes per host processor.

Table 2 shows the two target benchmarks and corresponding input data sets we used for our study. Both FFT and Radix are SPLASH2 applications [25].

In all our measurements in Section 6 we report the time it took *WWT II* to execute only the parallel portion of each 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.

Benchmark	**Number of Host Processors**	**K**	**Slowdown**		
			SMP	**COW**	**COW/SMP**
FFT	1	32	166	241	241
	2	16	95	159	170
	4	8	54	65	117
	8	4	36	45	47
Radix	1	32	114	186	186
	2	16	66	103	139
	4	8	39	55	80
	8	4	25	37	46

TABLE 4. This table shows *WWT II*'s slowdown. K = number of target nodes simulated per host processor. For all the above measurements, the target size is constant, i.e. 32 nodes (second column * third column).

6 Performance Results

This section describes *WWT II*'s performance and scalability on our three parallel machines. We characterize *WWT II*'s performance using a metric called *slowdown*. We define slowdown as follows:

$$\text{slowdown} = \frac{\text{Max}_i\ [\text{host cycles}]}{\Sigma_j\ [\text{target cycles}]},$$

$$i = 1 \ldots \text{number of host processors},$$
$$j = 1 \ldots \text{number of target nodes}$$

Thus, slowdown is the rate at which a host simulates target node cycles (Table 4).

On our SMP, *WWT II*'s slowdown is modest (between 25-166), while on the COW and COW/SMP, *WWT II*'s slowdown is slightly worse (between 37-241) compared to the SMP. On the SMP, *WWT II*'s slowdown is comparable to MIT Proteus' slowdown (between 35-100 on a uniprocessor host) [3]. However, such comparison of slowdown between different simulators may not be insightful because slowdown

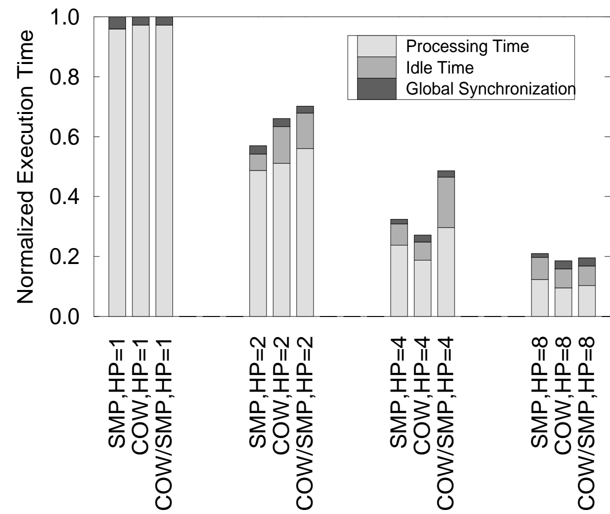
depends on the functionality and level of detail supported by a simulator.

Figure 4 provides further insight into *WWT II*'s scalability. It shows the breakdown of the average processing time, idle time, and global synchronization time, as was done qualitatively in Figure 1**.** The processing time is the sum of the execution time for all target processes. The idle time represents the fact that processing time is not uniformly distributed. Thus, it is the number of host processors multiplied by the critical path processing time minus the processing time. The critical path processing time is the sum of the maximum processing time across hosts for each quantum. Measuring the global synchronization time is more difficult. This occurs because the synchronization at the end of each quantum does not imply that all host processors leave SAM at the same time. An efficient software implementation of a barrier (synchronization) is unlikely to cause all processors to complete the barrier simultaneously. The only requirement is that no processes exit until all processors have entered the barrier. Consequently, an actual machine does not have the nice clean picture shown in Figure 1. The easiest way to overcome these issues is to define the synchronization time to be the difference between the total time and the critical path processing time. This works since the measured processing times are not effected by the variation in barrier exit times. This is how the results presented here were calculated.

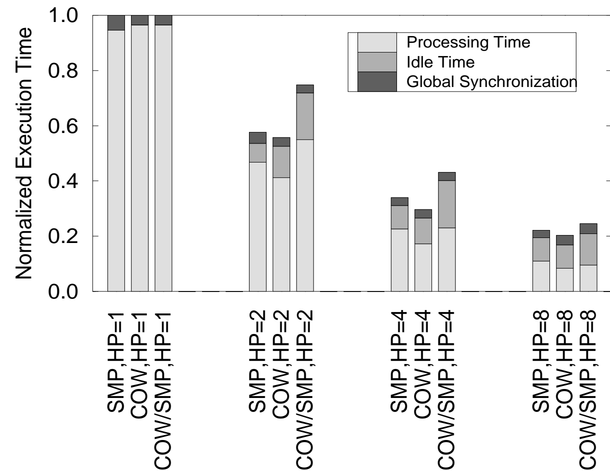
Figure 4 shows three interesting characteristics of *WWT II*. First, *WWT II* scales similarly for the SMP and the COW. However, COW/SMP scales (and performs) slightly worse than the COW for the same number of host processors. This is because the processing time on the COW/SMP is worse than that on a COW. We suspect this difference is caused by our COW/SMP node's memory system, which does not scale as well with increasing processors due to the particular memory bus used within one node.

Second, the global synchronization time is a small fraction of the total execution time (between 4-17%). This implies that our implementation of SAM and the synchronization layer is quite efficient. Further improvements in the synchronization layer can improve performance only marginally.

Third, with increasing host processors, the idle time introduced due to load imbalance within a quantum is a dominant factor of the total execution time. This idle time appears to be the key limiting factor to *WWT II*'s scalability. For example, on 8 host processors, the idle time accounts for 33-46% of the total execution time. Consequently, *WWT II* achieves a speedup of 4.1-5.4 on 8 host processors, which is good, but less than linear. We believe that the idle time increases with increasing host processors because we



(a) FFT



(b) Radix

FIGURE 4. This figure shows how *WWT II*'s scales for FFT (a) and Radix (b) on our three parallel machines. HP denotes the number of host processors in each configuration. The vertical axis shows the total execution time for a particular configuration divided by the total execution time on one host processor of the corresponding machine. On a single host processor, *WWT II* runs on average 3.8 times faster on the SMP compared to the COW. The COW/SMP execution time on a single host is identical to that of the COW. Each execution time bar is further divided up into average processing, idle, and global synchronization times. The global synchronization time on a single host denotes the overhead of switching to a different target node, while that on multiple host processors denote both the overhead of switching target nodes and synchronizing with other host processors.

have a fixed number of total target nodes. This increase occurs because the deviation of processing

time from the average increases as the number of targets per host decreases. This implies that increasing the number of target nodes per host, i.e. a larger simulation, with increasing host processors, will reduce the idle time and achieve better speedups [8].

Although *WWT II* does not achieve linear speedup, parallel simulation is still worthwhile. This is because such simulations are often memory-limited [26, 8]. Assume that (i) the goal is to maximize the rate at which one can do simulations of alternative design points, (ii) each simulation has a working set of M_{sim}, and (iii) the machine has a physical memory size of $M_{physical}$. To avoid thrashing, one can concurrently run at most $S = \lfloor M_{physical} / M_{sim} \rfloor$ simulations. If the system has more than S host processors, parallel simulation increases the rate of performing alternative simulations whenever speedup is greater than 1.0! This combined with our speedups of 4.1-5.4 on (8 processors) makes parallel simulation worthwhile.

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 platforms, including desktop workstations, a SUN Enterprise server, a cluster of workstations, and a cluster of symmetric multiprocessing nodes.

On two benchmarks, we found that the *WWT II* achieved both good performance and good scalability. Uniprocessor *WWT II* simulated one target cycle of a 32-node target machine in 114 and 166 host cycles respectively for the two benchmarks on an UltraSPARC. Parallel *WWT II* achieved speedups between 4.1-5.4 on 8 host processors on a SUN Enterprise E5000 server, a cluster of workstations, and a cluster of symmetric multiprocessing nodes.

References

[1] Thomas Ball. Efficiently Counting Program Events with Support for On-Line Queries. *ACM Transactions on Programming Languages and Systems*, 16(5):1399–1410, September 1994.

[2] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[3] 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.

[4] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set Version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin–Madison, May 1997.

[5] T. M. Conte and W. W. Hwu. Systematic Prototyping of Superscalar Computer Architectures. In *Proceedings of the 3rd IEEE International Workshop on Rapid System Prototyping*, June 1992.

[6] 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.

[7] 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.

[8] Babak Falsafi and David A. Wood. Cost/Performance of a Parallel Computer Simulator. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)*, July 1994.

[9] Babak Falsafi and David A. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[10] Erik Hagersten, Ashley Saulsbury, and Anders Landin. Simple COMA Node Implementations. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, page ?, January 1994.

[11] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *COMPCON '95*, pages 327–332, San Francisco, California, March 1995. IEEE Computer Society.

[12] Sun Microsystems Inc. *SPARC MBus Interface Specification*, April 1991.

[13] James R. Larus and Eric 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.

[14] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, May 1996.

[15] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, 1997.

[16] Steven K. Reinhardt. *Mechanisms for Distributed Shared*

Memory. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, December 1996.

[17] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[18] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[19] Eric Schnarr and James R. Larus. Instruction Scheduling and Executable Editing. In *29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 29)*, pages 288–297, December 1996.

[20] Ashok Singhal, David Broniarczyk, Fred Ceraukis, Jeff Price, Leo Yuan, Chris Cheng, Drew Doblar, Steve Fosth, Nalini Agarwal, Kenneth Harvey, Erik Hagersten, and Bjorn Liencres. Gigaplane (TM): A High Performance Bus for Large SMPs. In *Hot Interconnects IV*, pages 41–52, 1996.

[21] Jeff. S. Steinman. SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete-Event Simulation. *International Journal in Computer Simulation*, 2:251–286, 1992.

[22] Richard Uhlig, David Nagle, Trevor Mudge, and Stuart 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.

[23] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[24] Emmett Witchel and Mendel Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Proceedings of the 1996 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[25] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, July 1995.

[26] David A. Wood and Mark D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, 28(2):69–72, February 1995.