Shubhendu S. Mukherjee
Compaq Computer

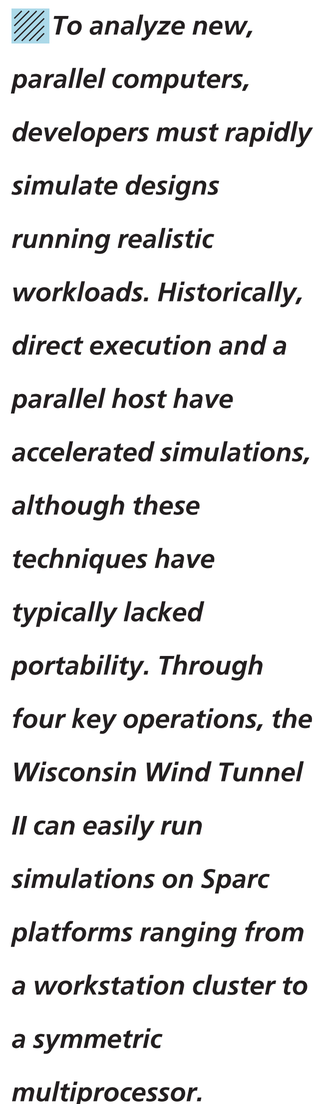
Steven K. Reinhardt
University of Michigan

Babak Falsafi
Carnegie Mellon University

Mike Litzkow, Mark D. Hill, and David A. Wood
University of Wisconsin-Madison

Steven Huss-Lederman
Beloit College

James R. Larus
Microsoft Research



To analyze new, parallel computers, developers must rapidly simulate designs running realistic workloads. Historically, direct execution and a parallel host have accelerated simulations, although these techniques have typically lacked portability. Through four key operations, the Wisconsin Wind Tunnel II can easily run simulations on Sparc platforms ranging from a workstation cluster to a symmetric multiprocessor.

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

Simulation techniques speed design by enabling architects to evaluate computers—from microprocessors to parallel machines—without building hardware prototypes. However, simulating parallel machines with realistic workloads requires significant computation and memory. Two techniques make such simulation feasible: *direct execution* and *parallel simulation*.

In direct execution, a program from the target system runs on an existing host system.[1] 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 simulates only those 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 exploits the target computer's inherent parallelism and the parallel host's large memory to hold the simulator's working 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. In contrast, other solutions to this problem—RSIM, SimOS, and SimICS, for example—run on uniprocessor hosts and are exceedingly slow in simulating large-target multiprocessors.[2–4]

Unfortunately, parallel discrete-event, direct-execution simulators are complex; building and porting them can be difficult. 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 either target executables or assembly code to calculate a target's execution time and simulate missing features. Some simulators[5,6] also modify the operating system to detect target cache misses. Also, parallel simulators often use machine-specific synchronization and communication features to achieve good parallel performance.

These low-level dependencies have been painfully obvious to us as developers of two generations of parallel direct-execution simulators. To minimize these dependencies, we have developed two tools, Elsie and Synchronized Active Messages (SAM), that encapsulate the operations underlying these simulations in a portable manner. With these tools, we've ported the Wisconsin Wind Tunnel II—the successor to the original Wisconsin Wind Tunnel[5]—to a range of platforms, including desktop workstations, a Sun

1092-3063/00/$10.00 © 2000 IEEE

enterprise server, and a cluster of Sparc workstations. The results we obtained on these platforms highlight WWT II's portability, good parallel efficiencies across a range of platforms, and cost-effectiveness.

The operations underlying simulations

Four operations help developers 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. Two others—*communication of target messages* and *synchronization of host processors*—relate to conservative-window, parallel, discrete-event simulation.

Calculation of target execution time

To evaluate a proposed architecture's performance, 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, elapsed-time calculation 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 variable after every instruction would nullify this performance advantage.

We can reduce the cost of updating the target clock variable in two ways. First, instead of invoking the simulator, the target itself can maintain and update its own target clock variable. This implies that we must augment the target code with extra code that updates the 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.[5] Unfortunately, the first three levels require source, assembly, or object code, which might be hard to obtain for vendor-provided libraries or commercial operating systems and databases. Executable modification removes this restriction because it adds target clock instrumentation directly to the executable; but it introduces two problems. First, its implementation is complex because the executable editor must handle machine-specific details (for example, fixing 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 let users 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. EEL (executable editing library)[8] is one such tool, which WWT II developers used to build an executable editor tool named Elsie (for "Edits Loads and Stores In Executables"), to perform the target clock instrumentation on target executables. We describe Elsie later.

Simulation of features of interest

For the study of proposed parallel architectures, simulators must let researchers simulate features that might be unavailable in a host. For example, the original WWT simulated a hardware, cache-coherent, shared-memory machine on Thinking Machines' CM-5, which is a message-passing parallel machine.

In direct execution, missing-feature simulation requires the target to execute a jump to 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 the host's missing features. The first uses host hardware and software mechanisms to transfer control. For example, WWT and Tapeworm II, a different simulator developed by Uhlig and colleagues,[6] marked host memory blocks, absent in the target cache or translation look-aside buffer (TLB), with bad error correction code. Accesses to memory blocks with bad ECC generated traps that were vectored to the simulator through the operating system. This let WWT and Tapeworm II simulate cache and TLB misses, respectively. This method is not, however, 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 errors. Without precise exceptions, a simulator will be unable to correctly simulate target cache misses.

The second approach replaces target instructions with code segments that transfer control to the simulator. Because this approach is more general, it can incur a performance penalty. For example, for this approach to simulate target cache misses, all loads and stores must check the target cache state, unlike the WWT approach in which the simulator checks the target cache block state only on cache misses.

Replacing instructions with new code segments introduces problems like those faced by target clock instrumentation; so, our solution is similar. We augment Elsie to replace target instructions to simulate features missing in the host—in our case, the target memory system.

Communication of target messages

Communication is inherent in parallel simulation because target nodes exchange messages. However, the most efficient communication method 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. To overcome this problem, we developed SAM, a simple messaging library. SAM abstracts the communication primitives away from the implementation mechanisms and techniques, and it handles processor synchronization. We describe it in more detail later.

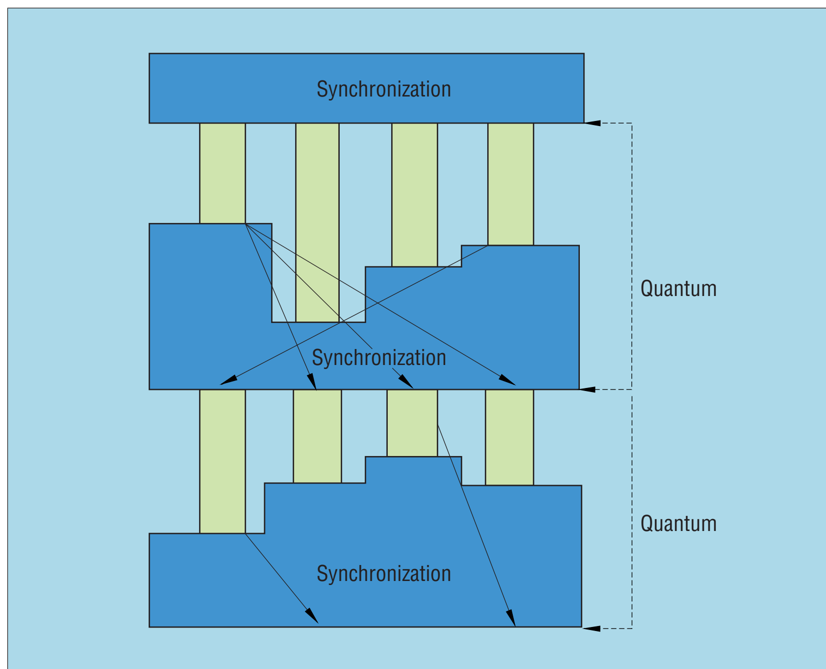


Figure 1. A graphical representation of quantum and messages sent for four processors. Blue regions are synchronization times; green areas are simulator processing times.

Synchronization of host processors

Parallel, discrete-event simulation that uses *conservative time bucket synchronization*[9] must rapidly synchronize host processors. This method breaks target execution into *quanta*, lock-step intervals (see Figure 1). Target messages sent during one quantum can affect target state only in subsequent quanta. We accomplish this effect by setting the quantum length on the basis of the delivery time necessary for a message to reach the target (this is a lower bound, so it is conservative). Because message delivery is guaranteed before the next quantum starts, the simulator ensures that the receiving target is aware of the message before it can have any effect on the target program's outcome.

Conservative-window, parallel, discrete-event simulation imposes three synchronization requirements. First, host processors must detect when target execution reaches a quantum's end.

Second, when a quantum expires, host processors must synchronize among themselves using a barrier and calculate the next quantum's duration. This duration is typically calculated as the sum of the minimum target execution time across all host processors (conventionally called a *reduction*) and a fixed quantum length (for example, 100 target processor cycles). This minimum execution time represents the simulator's "knowing" that all targets will not interact for some period of time so that it can extend the next quantum. The fixed quantum length 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 next quantum begins. Figure 1 shows that messages sent are received at the end of synchronization. A global reduction of the difference between the number of messages sent and received will be zero once delivery is complete. This lets a host processor complete reception of all target messages before beginning the next quantum.

Detecting a quantum's end

There are two ways to detect a quantum's end. First, the simulator can check for quantum expiration on each entry into the simulator. This 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, which might delay other target nodes. In this case, we can modify the target executable to check the target execution time more frequently (for instance, on target clock updates) and invoke the simulator if a quantum has expired. This method is more robust but adds overhead.

Synchronization

Different parallel computers provide different degrees of hardware support for barrier synchronization and reductions. For example, the 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 cluster of workstations connected with an off-the-shelf network lack hardware support for either. So, these machines must implement both in software. The lack of hardware support for barriers and reductions can degrade the performance of conservative-window, parallel, discrete-event simulation, particularly when quantum intervals are short.

Message processing

Most parallel computers lack the hardware support to determine whether all messages injected into a host network have been drained (received), although the CM-5 is a notable exception. However, there are various ways to do this in software. For example, we can collect acknowledgments for every message injected into the network. Alternatively, we can confirm message delivery at a quantum's end, combined with barrier synchronization. SAM implements the necessary functionality for the second alternative, while permitting portability.

Elsie

Elsie modifies target executables that run on WWT II (see Figure 2) to calculate target execution time and simulate features of interest. Like other executable editors for direct-execution simulators, Elsie adds instrumentation for this calculation and to simulate the target's memory system.

Surprisingly, contrary to earlier expectations, Elsie can be written in an

almost machine-independent fashion, for two reasons. First, Elsie uses EEL, which hides most details of modifying executables, 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 (loads and stores, for example) with snippets that jump into the simulator, which simulates the target memory system.

Second, machine-dependent snippets are few and small; the eight mandatory snippets each contain four or fewer instructions. Consequently, we would need to rewrite only small portions of machine-specific code to port Elsie to a different instruction set, and the few machine-specific instructions make porting even easier.

Elsie currently runs only on the Sparc V8 instruction set. Modifying Elsie for other instruction sets will involve describing the new processor's properties and using a version of EEL for that processor. For example, we will need detailed timings for the new instruction set.

Though the above techniques make WWT II portable, augmenting the target with instrumentation code for simulating 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. WWT II reduces its overhead by providing a fast path for loads and stores that hit in the target cache.[10] Typically, on a load or store, the simulator translates the virtual address to the physical address using the target TLB, indexes into the cache, and finds the appropriate cache block through a tag match. Next, the simulator checks the cache block's state and, on a cache hit, loads or stores a value from or to the cache block. But in the fast path, WWT II maintains pointers to all valid target cache blocks in each target TLB entry. So, if a load or store hits in the target cache, WWT II can directly find the block on a target TLB access.

SAM

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. The model achieves target message communication and host processor synchronization in the simulator.

SAM is simple by design so that it can be implemented easily across a range of parallel machines. It has three main primitives. Host processors communicate using SAM_Send_Msg, calculate the next quantum duration using SAM_Bcast_Msg (via broadcast messages), and synchronize using SAM_Sync.

Like active messages, a SAM message contains the virtual address of a handler that SAM will call at the receiving host processor. Unlike active messages, however, SAM messages do not guarantee message reception until SAM_Sync completes. When SAM_Sync completes, SAM guarantees that all messages have been received and processed (and messages scheduled for the next quantum) by calling the corresponding handlers. By supplying the appropriate handler, SAM can calculate the next quantum duration through message broadcasts for simplicity, thereby avoiding a separate reduction interface, such as the one in the CM-5.

Currently, SAM runs on three platforms: an SMP, a COW, and a cluster of SMPs (COW–SMP). We have optimized each implementation to the platform's underlying communication substrate.

The SAM SMP implementation is straightforward because our Sun E6000 SMP supports efficient low-latency communication over the memory bus. SAM allocates a shared-memory segment and, for each process in the parallel program, sets up two sets of mailboxes in shared memory—destination and source mailboxes. One process uses another's destination mailbox to send a point-to-point message to that process. Each message is explicitly copied into the destination mailbox because two processes share only the segment containing the mailboxes, not the entire address space. An atomic fetch-and-add operation ensures mutual exclusion within a destination mailbox. A process uses its own source mailbox to queue broadcast messages. To avoid multiple copies of the same message, we do not queue a broadcast message in the destination mailboxes.

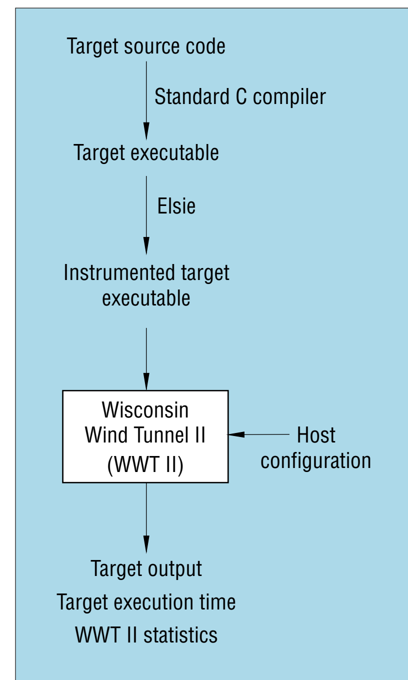


Figure 2. Elsie's relationship to the Wisconsin Wind Tunnel II.

Finally, when a process calls SAM_Sync, SAM drains a process's own destination mailboxes and checks all other processes' 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 high message overhead (26 microseconds under SunOS 5.5 with Myricom switches), so minimizing the number of messages is important. WWT II sends two or fewer messages, per processor, of up to 80 bytes in a quantum. Multiple messages occur on a host if it has multiple targets and because a single target's protocol processing can involve multiple messages.

Considering these characteristics, 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 minimize messages by piggybacking the target messages from the

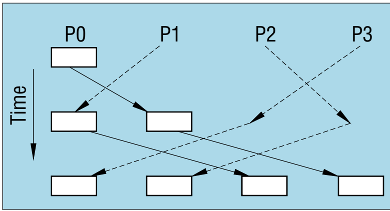


Figure 3. SAM implementation for a cluster of workstations. P0, P1, P2, and P3 denote host processors. Boxes represent data—here, only P0 sends a message. Solid lines represent the flow of synchronization messages with data (piggybacking). Dotted lines represent the flow of synchronization messages without data.

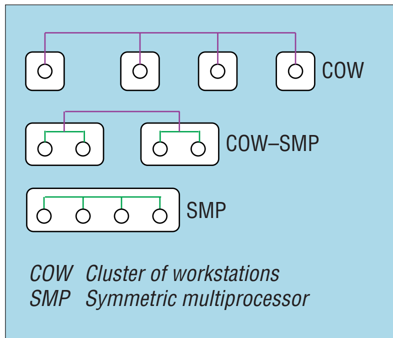


Figure 4. The different machine configurations for four processors. Green represents a bus; blue, a network.

current quantum and the data needed to determine the next quantum length on the butterfly synchronization. Because WWT II sends very few short messages in each quantum, the butterfly's total cost is not substantially increased over the synchronization cost, even though our piggybacking scheme sends all data to all host processors, as Figure 3 shows.

Table 1. The three host systems for WWT II. N *is the number of nodes and* P *is the total number of host processors.*

Parallel machine	Host processor	Interhost communication: Memory bus	Interhost communication: Network	*N*	*P*
SMP 16-processor Sun E6000	250-MHz UltraSparc	83.5 MHz; 256-bit-wide split transaction	N/A	1	16
COW Uniprocessor Sparcserver20	66-MHz HyperSparc	N/A	First-generation, version 2, Myricom Myrinet switches	16	16
COW–SMP Dual-processor Sparcserver 20	66-MHz HyperSparc	50 MHz; 64-bit-wide sequential transaction	First-generation, version 2, Myricom Myrinet switches	8	16

In the COW–SMP implementation, the SMP host processors first exchange messages. Next, one predesignated host processor in each SMP node exchanges messages with other host processors, as Figure 3 shows. Finally, SMP host processors synchronize locally to ensure that the predesignated processor has drained all messages from the network.

Test configurations

Table 1 shows the three parallel machine configurations we used with the WWT II experimental framework. Figure 4 depicts the three machines. The COW–SMP in Figure 4 is the same as the COW, except that each node has two processors, not one. We use 16 COW nodes and eight dual-processor COW–SMP nodes to equalize the number of host processors in the COW and COW–SMP configurations.

An S-COMA (simple cache-only memory architecture) shared-memory machine was our target architecture.[11] Each target node has a single processor and a 256-Kbyte processor cache. Hardware coherence is implemented through a full-map directory protocol. Each WWT II host processor simulates one or more target nodes. For example, for a 256-node target, an eight-processor WWT II configuration simulates 32 target nodes per host processor.

Table 2 shows our five target benchmarks and corresponding input data sets. Our measurements report the time WWT II took to execute only the parallel portion of each target benchmark. We assume our target benchmarks use the Sparc V8 instruction set, so we ensured that all of our host processors are Sparc V8 compatible.

Additionally, because WWT II always takes the same path through the target executable, all our target executable runs report the same execution cycles no mat-

Table 2. Target benchmarks and the corresponding input data sets for our experiments.

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, four iterations
water-sp	Splash-2	Water molecule simulation	4K molecules, three steps

ter which platform ran the experiments. WWT II always takes the same path through the executable because we strictly order the events. Such control over the experimental framework is essential to effectively characterize WWT II's performance across our three platforms.

Performance analysis

We analyzed WWT II's overall performance on the basis of two main factors: parallel performance and cost-effectiveness.

Parallel performance

We assessed WWT II's parallel performance in terms of the host's parallel speedup (uniprocessor time divided by parallel time). Table 3 compares WWT II's performance across our three parallel hosts; the simulator achieved reasonable speedups for a modest number of targets. We show only selected benchmarks and limited targets because they exemplify the results and are small enough to avoid virtual-memory thrashing on a single COW node. As shown below, WWT II's performance increases with larger simulations.

As an example of the WWT II's absolute runtimes, the 16-host processor runtimes for the tomcatv benchmark are 1.8 and 9.4 minutes for the SMP and COW, respectively. These runtimes show that parallel simulations are practical for many applications. The speedups we achieve are better on the SMP as the number of host processors increases, indicating that the SMP's faster communication yields better parallel performance than does the COW.

The SMP results are important because the large memory available for any number of processors lets us run 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 5a shows that 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%. The speedup curves

Table 3. Parallel speedups across platforms for WWT II on a 32-node target system.

Benchmark	No. of host processors	Speedup SMP	Speedup COW	Speedup COW–SMP
LU	1	1.0	1.0	1.0
	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	1.0	1.0	1.0
	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

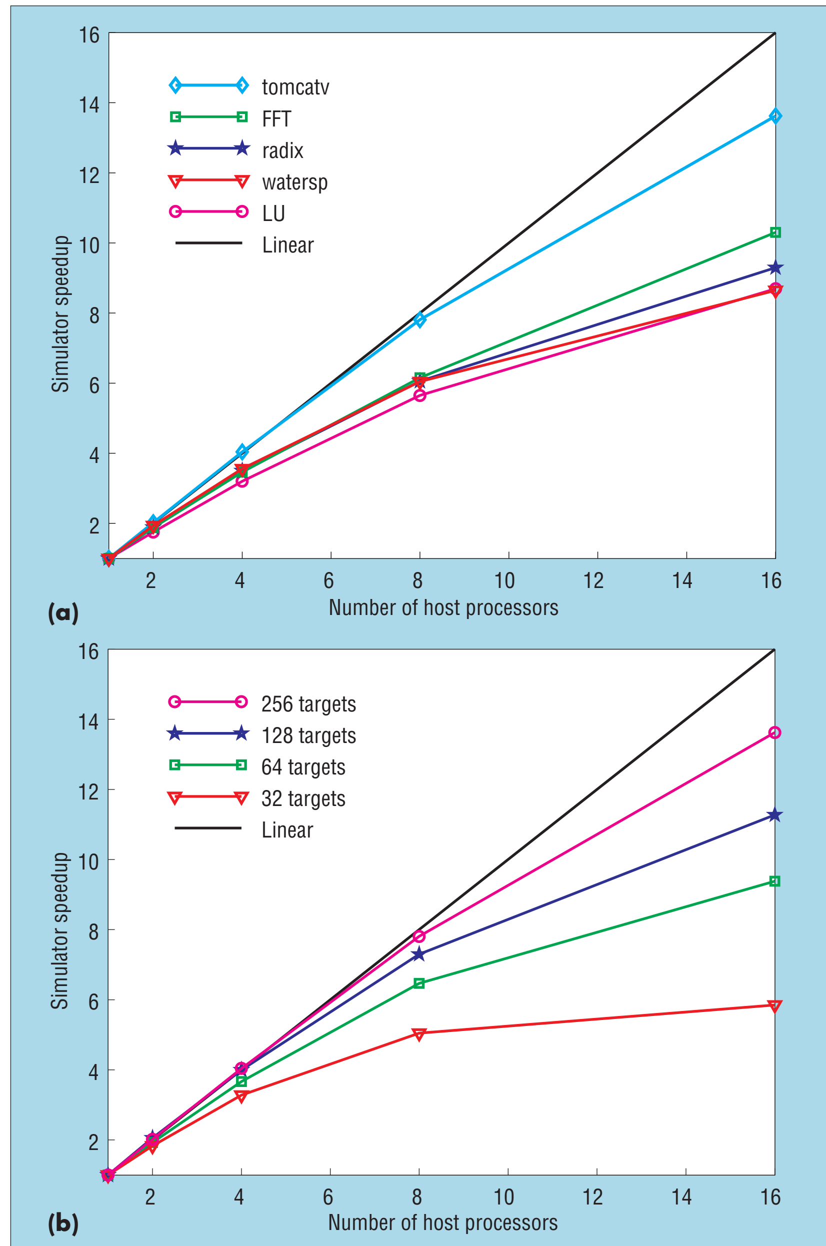


Figure 5. (a) Simulator speedups on a symmetric multiprocessor across benchmarks for 256 targets. (b) Simulator speedups on an SMP for tomcatv with a varying number of targets.

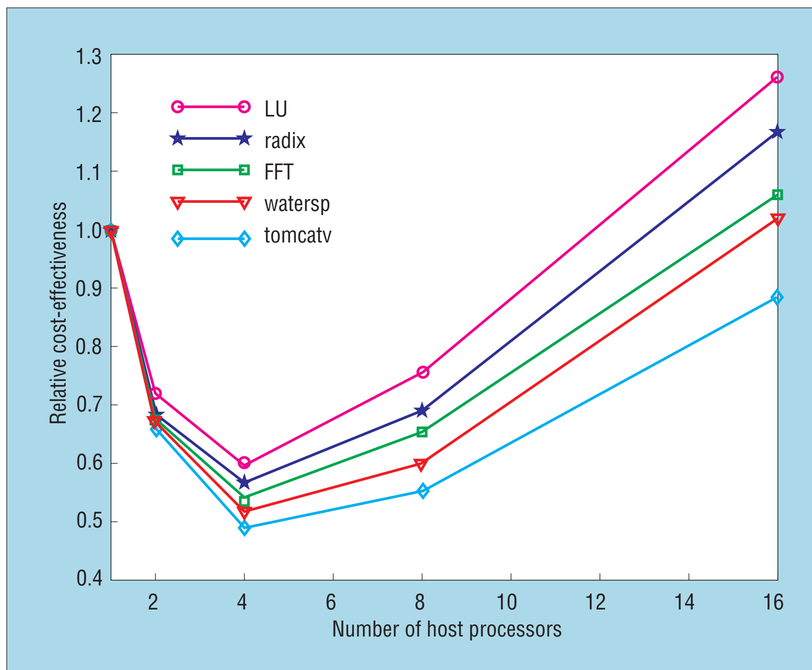


Figure 6. The relative cost-effectiveness across benchmarks for 64 targets and 64 Mbytes per target.

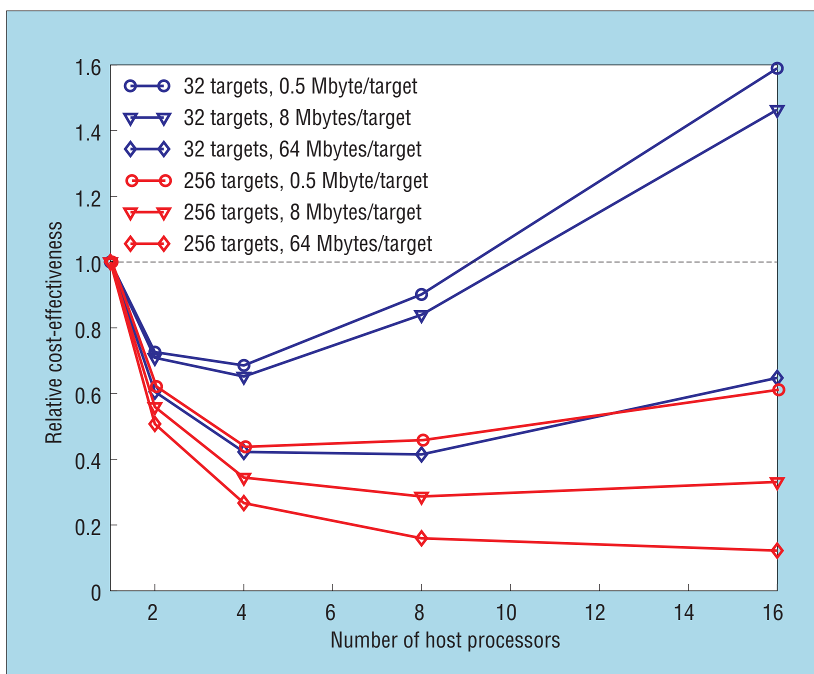


Figure 7. Relative cost-effectiveness when the number of targets and memory per target vary.

increase monotonically, so that greater parallelism reduces a given simulation's runtime.

Figure 5b shows the effect of varying the number of targets—increasing the targets increases the simulator speedups. This effect, seen on all benchmarks, is most pronounced with tomcatv. This trend indicates that larger simulations, which require greater uniprocessor runtimes, will achieve better parallel performance.

Key to the increased efficiency is the reduction in idle time, resulting from improved load balancing as the number of targets per host increases. Once a host completes work for its targets in the current quantum, it idles until the slowest host completes and enters the synchronization as shown in Figure 1. As the number of targets per host increases, the deviation from the average decreases, as does the idle time.[12]

Cost-effectiveness

Although parallelism clearly improves runtimes for a given simulation, this does not demonstrate that parallelism is cost-effective.[13] To evaluate whether running a parallel simulation on N host nodes is cheaper than running N sequential simulations, we must specify the cost of the host systems. The cost is the purchase price of the smallest system that could run the proposed simulation. So, a simulation run on four hosts, needing 1 Gbyte of memory, would be the cost of the smallest box with four processors and 1 Gbyte of memory.

Memory is a key cost component. In our WWT II analysis, we determined the simulator's memory usage (in Mbytes), 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 used 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

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

where P is the number of host processors, and *base* (base cost of an "empty" system) is 17.5 if $P \leq 6$, 48.5 if $7 \leq P \leq 14$, and 181.5 if $15 \leq P \leq 30$. (These cost figures came from a 1997 Sun price list.) From the simulation's cost and runtime, we determine cost-effectiveness by

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

where a lower value is better. We determine a parallel simulator's cost-effectiveness from the relative cost-effectiveness

of running the simulation on P processors versus one processor:

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

Values less than one show that running on P processors is cheaper than running on one processor.

Figure 6 shows the relative cost-effectiveness across the benchmarks. These results assume that each target uses 64 Mbytes of memory, and the speedups are those achieved when running the data sets in Table 2. Parallel simulation is cost-effective for these benchmarks, simulator, and cost parameters until we try to run on 16 host CPUs. At that point, only tomcatv is cost-effective. For all benchmarks, the point of lowest cost is on four host processors. Parallel simulation is thus not only faster but costs roughly half (48% to 59%) of the uniprocessor simulation.

Figure 7 shows how varying the number of targets and memory per target for tomcatv affects cost-effectiveness. Again, we assume that the speedups measured from the actual benchmarks are unchanged as the amount of memory varies. The relative cost-effectiveness improves as the number of targets increases, which is consistent with the previous result—speedups improve as the number of targets increases. Also, as the memory per target (and thus total memory) increases, the relative cost-effectiveness improves. Both trends, observed across the benchmarks, are consistent with previous results.[12,13]

For the largest benchmark considered in Figure 7—256 targets, 64 Mbytes per target—the relative cost-effectiveness drops as the number of host processors rises. For this simulation, 16 host processors is most cost-effective, costing 12% of the uniprocessor simulation. The optimal number of host processors, however, remains an open question.

At the other extreme—32 targets, 0.5 Mbytes per target—the graph results resemble those in Figure 6. Here, four host processors are 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.

WWT II DEMONSTRATES THE technologies that support parallel simulation of target multiprocessors with up to hundreds of in-order processors executing user-level code. More information on WWT II can be found at www.cs.wisc.edu/~wwt/wwt2.

Researchers at Wisconsin continue multiprocessor research but with an emphasis on commercial workloads. Simulation work continues, using the Virtutech SimICS infrastructure[4] that supports the operating system and device simulation necessary for these workloads. Future work will apply ideas from WWT II so that workload sizes can more accurately reflect what the Internet economy demands of its computing infrastructure.

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 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; a University of Wisconsin Graduate School Grant; a Wisconsin Alumni Research Foundation Fellowship; and donations from Digital Equipment, IBM, Sun Microsystems, Thinking Machines, and Xerox. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant CDA-9024618 with matching funding from the University of Wisconsin Graduate School.

The US 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 US government.

References

1. R. Covington et al., "The Rice Parallel Processing Testbed," *Proc. ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems*, ACM Press, New York, 1988, pp. 4–11.
2. P. Ranganathan, V.S. Pai, and S.V. Adve, "RSIM: An Execution-Driven Simulator for ILP-Based Shared Memory Multiprocessors and Uniprocessors," *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter,* Oct. 1997.
3. M. Rosenblum et al., "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Trans. Modeling and Computer Simulation*, Vol. 7, No. 1, Jan. 1997, pp. 78–103.
4. P.S. Magnusson et al., "SimICS/Sun4m: A Virtual Workstation," *Proc. Usenix Ann. Technical Conf.*, Usenix Assoc., Berkeley, Calif., 1998.
5. S.K. Reinhardt et al., "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers," *Proc. ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems*, ACM Press, New York, 1993, pp. 48–60.
6. R. Uhlig et al., "Trap-Driven Simulation with Tapeworm II," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS VI), ACM Press, New York, 1994, pp. 132–144.
7. H. Davis, S.R. Goldschmidt, and J. Hennessy, "Multiprocessor Simulation and Tracing Using Tango," *Proc. Int'l Conf. Parallel Processing, Vol. II: Software*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1991, pp. II99–II107.

8. J.R. Larus and E. Schnarr, “EEL: Machine-Independent Executable Editing,” *Proc. SIGPLAN 1995 Conf. Programming Language Design and Implementation* (PLDI ’95), ACM Press, New York, 1995, pp. 291–300.
9. R.M. Fujimoto, “Parallel Discrete Event Simulation,” *Comm. ACM*, Vol. 33, No. 10, Oct. 1990, pp. 30–53.
10. A.R. Lebeck and D.A. Wood, “Active Memory: A New Abstraction for Memory-System Simulation,” *Proc. Sigmetrics Conf. Measurement and Modeling of Computer Systems*, ACM Press, New York, 1995, pp. 220–230.
11. E. Hagersten, A. Saulsbury, and A. Landin, “Simple COMA Node Implementations,” *Proc. 27th Hawaii Int’l Conf. System Sciences* (HICSS 27), IEEE Computer Soc. Press, Los Alamitos, Calif., 1994.
12. B. Falsafi and D.A. Wood, “Modeling Cost/Performance of a Parallel Computer Simulator,” *ACM Trans. Modeling and Computer Simulation*, Vol. 7, No. 1, Jan. 1997, pp. 104–130.
13. D.A. Wood and M.D. Hill, “Cost-Effective Parallel Computing,” *Computer*, Vol. 28, No. 2, Feb. 1995, pp. 69–72.

Shubhendu S. Mukherjee is a principal hardware engineer at Compaq Computer in Shrewsbury, Massachusetts, involved with advanced research and development of Alpha microprocessors. Mukherjee has worked extensively on microarchitectures, hardware fault detection, cache coherence protocols, network interfaces, multiprocessor architectures, and performance models. Mukherjee received his B.Tech. in computer science and engineering from the Indian Institute of Technology, Kanpur, India, and his MS and PhD in computer sciences from the University of Wisconsin-Madison. Contact him at Compaq Computer Corp., 334 South St., Mail Stop SHR3-2E/R28, Shrewsbury, MA 01545; shubhendu.mukherjee@compaq.com.

Steven K. Reinhardt is an assistant professor of electrical engineering and computer science at the University of Michigan in Ann Arbor. His primary research interest is in computer system architecture, focusing on uniprocessor and multiprocessor memory systems, operating system and architecture interactions, and system simulation techniques. He received his BS from Case Western Reserve University and his MS from Stanford University, both in electrical engineering; and his PhD in computer sciences from the University of Wisconsin-Madison. Contact him at EECS Dept., 1301 Beal Ave., Ann Arbor, MI 48109-2122; stever@eecs.umich.edu; www.eecs.umich.edu/~stever.

Babak Falsafi is an assistant professor in the Electrical and Computer Engineering Department at Carnegie Mellon University. Previously, he was an assistant professor in the School of Electrical and Computer Engineering at Purdue University. His research interests include prediction and speculation in high-performance memory systems, power-aware processor and memory architectures, and analytic and simulation tools for computer system performance evaluation. He received his BS in computer science and BS in electrical and computer engineering from the State University of New York at Buffalo, and his PhD in computer sciences from the University of Wisconsin. He is a recipient of the NSF Career award in 2000 and is a member of the IEEE and the ACM. Contact him at Carnegie Mellon Univ., ECE Dept., Pittsburgh, PA 15213; babak@ece.cmu.edu; www.ece.cmu.edu/~babak.

Mike Litzkow is a researcher in the Computer Sciences Department at the University of Wisconsin-Madison. His technical interests include multimedia educational software and parallel and distributed computing systems. He earned his BS in computer sciences from the University of Wisconsin. Projects he has worked on include the CSNet Nameserver, the Charlotte Distributed Operating System, Condor, the Wisconsin Wind Tunnel, and Learning on Demand. Contact him at the Computer Sciences Dept., Univ. of Wisconsin-Madison, 1210 West Dayton St., Madison, WI 53706; mike@cs.wisc.edu; http://eteach.cs.wisc.edu/Mike.

Steven Huss-Lederman is an associate professor in the Computer Science Department at Beloit College. Previously, he was an assistant faculty associate in the Computer Sciences Department at the University of Wisconsin-Madison. His research has included computational chemistry, parallel algorithms, and linear algebra. He earned his BS from the University of Maryland and his PhD from the California Institute of Technology in chemistry. He is a member of the IEEE. Contact him at Beloit College, 700 College St., Beloit, WI 53511-5595; huss@beloit.edu; http://cs.beloit.edu/~huss.

Mark D. Hill is a professor and Romnes Fellow in both the Computer Sciences Department and the Electrical and Computer Engineering Department at the University of Wisconsin-Madison. He is a director of ACM SigArch, is coinventor on 16 US patents, has published more than 50 technical papers, and recently coedited *Readings in Computer Architecture* (Morgan Kaufmann). He earned a BSE in computer engineering from the University of Michigan and an MS and PhD in computer science from the University of California, Berkeley, won an NSF Presidential Young Investigator award, and is an IEEE Fellow. Contact him at the Depts. of Computer Sciences and Electrical and Computer Eng., Univ. of Wisconsin-Madison, 1210 West Dayton St., Madison, WI 53706-1685; markhill@cs.wisc.edu; www.cs.wisc.edu/~markhill.

James R. Larus is a senior researcher at Microsoft Research, where he runs the Software Productivity Tools group. Before joining Microsoft, he was an associate professor in the Computer Sciences Department at the University of Wisconsin-Madison, where he ran the Wisconsin Wind Tunnel project with Mark Hill and David Wood. He has an AB in applied math from Harvard College and an MS and a PhD in computer science from the University of California, Berkeley. He won a NSF Young Investigator award in 1993. Contact him at larus@microsoft.com.

David A. Wood is an associate professor and Romnes Fellow in the Computer Sciences and Electrical and Computer Engineering Departments at the University of Wisconsin-Madison. He coleads the Wisconsin Multifacet project (www.cs.wisc.edu/multifacet), exploring techniques for exploiting prediction and speculation in memory systems. He received his BS in electrical engineering and computer science and PhD in computer sciences from UC Berkeley. He received the NSF’s Presidential Young Investigator Award; is area editor of *ACM Transactions on Modeling and Computer Simulation*, and is a member of the ACM, the IEEE, and the IEEE Computer Society. Contact him at david@cs.wisc.edu.