NOTE: This is a preliminary release of an article accepted by the ACM Transactions on Modeling and Computer Simulation. The definitive version is currently in production at ACM and, when released, will supersede this version.

Modeling Cost/Performance of a Parallel Computer Simulator

Babak Falsafi and David A. Wood

This paper examines the cost/performance of simulating a hypothetical *target* parallel computer using a commercial *host* parallel computer. We address the question of whether parallel simulation is simply faster than sequential simulation, or if it is also more cost-effective. To answer this, we develop a performance model of the Wisconsin Wind Tunnel (WWT), a system that simulates cache-coherent shared-memory machines on a message-passing Thinking Machines CM-5. The performance model uses Kruskal and Weiss's fork-join model to account for the effect of event processing time variability on WWT's conservative fixed-window simulation algorithm. A generalization of Thiébaut and Stone's footprint model accurately predicts the effect of cache interference on the CM-5. The model is calibrated using parameters extracted from a fully-parallel simulation ($p = N$), and validated by measuring the speedup as the number of processors (p) ranges from one to the number of target nodes (N). Together with simple cost models, the performance model indicates that for target system sizes of 32 nodes and larger, parallel simulation is more cost-effective than sequential simulation. The key intuition behind this result is that large simulations require large memories, which dominate the cost of a uniprocessor; parallel computers allow multiple processors to simultaneously access this large memory.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*modeling techniques*

General Terms: Performance, Experimentation

Additional Key Words and Phrases: Shared-memory multiprocessors, memory systems, conserva-

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 PYI Award CCR-9157366, NSF Grant MIP-9225097, an IBM Cooperative fellowship, and donations from A.T.&T. Bell Laboratories, 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.
An earlier version of this manuscript appears in [Falsafi and Wood 1994]. A generalized version of the cost/performance results also appears in [Wood and Hill 1995]. This manuscript presents a comprehensive description of the performance models used in the previous manuscripts.

Affiliation: Computer Sciences Department, University of Wisconsin–Madison
Address: 1210 West Dayton Street, Madison, WI 53706

tive discrete-event simulation, performance analysis

1. INTRODUCTION

The architecture of a parallel computer specifies an interface between software and hardware. Computer architects prefer to study the complex interactions across this interface by running and measuring real applications. Simulation allows evaluation of these interactions without building hardware prototypes, speeding the design process.

Simulation has long been used to evaluate proposed computer hardware for correctness and performance. However, most simulations have focussed on low-level implementation details: circuit level, switch level (ideal transistor), or logic (gate) level. These detailed simulations serve an important function, but are orders-of-magnitude too slow to evaluate system-level performance. Real applications on parallel machines run for billions, or even trillions of cycles; even register-transfer-level simulators are much too slow.

Over the last several years, direct execution has become widely used to accelerate architectural simulations [Covington et al. 1988; Brewer et al. 1991; Boothe 1992; Davis et al. 1991; Lin 1992; Uhlig et al. 1994]. Direct execution exploits the commonality between the instruction set of the simulated *target* machine and the underlying *host* system. For example, a floating-point multiply on the target is "simulated" by executing a floating-point multiply on the host. Such a system need only simulate the differences between the target system and the host, achieving impressive performance when the two systems are very similar.

Simulations of parallel computers have exploited direct execution in several ways [Boothe 1992; Davis et al. 1991; Cmelik and Keppel 1994; Rosenblum et al. 1995]. Most commonly, a parallel target system is simulated on a uniprocessor host. For example, the Tango system spawns an event generation process for each processor in a target shared-memory system. These processes directly execute all computation instructions, but must send most memory references to a central simulation process. Tango can be parallelized to a limited extent by running the event generation processes in parallel, but the central memory-system simulation process quickly becomes a bottleneck.

More recent simulators [Reinhardt et al. 1993; Brewer et al. 1991; Rosenblum et al. 1995; Fujimoto 1983] extend direct execution to simulate a parallel target machine on top of parallel host. The first of these—the Wisconsin Wind Tunnel (WWT)—runs on a a Thinking Machines CM-5 [Hillis and Tucker 1993]. WWT differs from the other simulators in two ways. First, it directly executes all load and store instructions that hit in the target system's cache. Second, it integrates direct-execution with a conservative fixed-window parallel discrete-event simulation algorithm to not only parallelize event generation, but also the memory system simulation [Lubachevsky 1989; Ayani 1989; Nicol 1992; Fujimoto 1990; Misra 1986].

Parallel simulators like WWT are much faster than comparable uniprocessor simulators, providing the quick turn-around-time that can be so important to the design cycle. However, parallel simulation is not necessarily cost-effective for evaluating alternative parallel machines. Computer architects frequently run many independent simulations—for different applications, memory systems, and system sizes—and compare the results. Because these simulations are independent and run as a batch, parallelism can be achieved much more simply by running them simultaneously on different workstations. And since workstations have better (i.e., lower) cost/performance ratios than parallel computers, this simpler "coarse-grain" parallelism appears more cost-effective than finer-grain parallel simulators like WWT. This cost/performance differential is only exacerbated by the reality that parallel simulators rarely achieve perfect speedups.

However, in the central result of this paper, we show that parallel computer simulations are, in fact, more cost-effective than uniprocessor simulations, for sufficiently large target systems. The key intuition behind this result is that large simulations require large memory sizes, which dominate the cost of a uniprocessor; parallel computers allow multiple processors to simultaneously access this large memory. Using cost models based on commercial products and general technology trends and a performance model based on WWT, we show that (1) for bus-based shared-memory multiprocessors, parallel simulation becomes more cost-effective when target systems reach 16 or 32 nodes, and (2) for massively parallel systems, with their large price premium, parallel simulation becomes more cost-effective when the target system size reaches 32.

This paper also develops an analytic model of WWT's performance which incorporates three major factors: event processing time, context switch overhead, and host cache and TLB interference. We show that the variability in event processing times can be accurately modeled using Kruskal and Weiss's model for fork-join parallel programs [Kruskal and Weiss 1985]. The frequency of context switches, incurred when switching between target nodes, is accurately modeled by the maximum of binomial random variables. We extend Thiébaut and Stone's *footprint* model to predict the interference of multiple targets in the host cache and TLB [Thiébaut and Stone 1987]. Our model improves upon Thiébaut and Stone's by allowing for (1) multiple (more than two) interfering processes and (2) sharing of address space among the processes. Finally, we show that the model accurately estimates the measured speedup of WWT, with maximum error of 8% in three applications and 16% for all five applications.

The next section reviews the design of the Wisconsin Wind Tunnel. Section 3 develops the analytic performance model, and Section 4 compares its predictions to the measured speedups. Section 5 describes how the performance model is extended to estimate cost/performance for many target and host systems. Section 6 presents and discusses the results from this cost/performance model, and Section 7 summarizes our contributions.

2. SIMULATION METHODOLOGY

2.1 The Wisconsin Wind Tunnel

The Wisconsin Wind Tunnel (WWT) is a simulator for evaluating parallel computer systems—specifically cache-coherent shared-memory computers [Reinhardt et al. 1993]. WWT uses the execution of shared-memory applications to drive a distributed discrete-event simulation of proposed hardware. Events generated by the simulation, such as cache misses and coherence messages, are used to schedule the application, permitting accurate calculation of the target system execution time.

WWT uses direct execution to exploit similarities between the target system (under study) and the host system (on which it executes). Because WWT executes on a message-passing machine (a Thinking Machines CM-5), it must simulate the shared memory abstraction using a fine-grain extension of Li's shared virtual memory [Li and Hudak 1989]. Shared virtual memory uses the standard address translation hardware to control memory access on each node. When a node first accesses a shared data page, it allocates a local copy and maps it into the shared address space on that node; subsequent accesses reference the copy. Multiple read-only copies are supported using the page protection facilities. Program accesses that require a data transfer to acquire a valid or exclusive copy are signaled as page faults.

WWT's fine-grain extension uses the CM-5's error-correcting code (ECC) bits to synthesize tag bits on each 32-byte block in physical memory [Reinhardt et al. 1993]. Using the three tag values—*invalid*, *read-only*, and *writable*—in combination with the address translation hardware, WWT implements a distributed shared memory that maintains coherence at a finer granularity than a virtual memory page.

WWT uses logical clocks to correctly calculate the logical execution time of a target system, modeling latencies, dependencies, and queuing. WWT manages interprocessor interactions by dividing program execution into lock-step quanta (also called fixed windows [Fujimoto 1990], bounded lag [Lubachevsky 1989] or time buckets [Steinman 1992]) to ensure all events originating on a remote node that affect a node in the current quantum are known at the quantum's beginning. WWT implements this using the CM-5's "network done" barrier, which guarantees that all messages are received before the quantum completes [Leiserson et al. 1993]. WWT combines this distributed simulation algorithm with direct execution by ordering all events on a node for the current quantum and directly executing the process up to its next event.

3. WWT PERFORMANCE MODEL

One approach to evaluating cost/performance of parallel simulation is to simply measure the performance of simulation runs on several different system sizes and directly compute the cost/performance ratio. Unfortunately, while this technique provides exact results for the measured systems, it is difficult to extrapolate them to larger or smaller systems. Furthermore, simple measurements provide little or no insight into *why* a system performs as it does, making it difficult to understand the generality of the results.

In this study, we construct a performance model of the Wisconsin Wind Tunnel that accurately predicts the simulation speedup. We do this by estimating the time to simulate N target nodes on p host nodes; as we vary p, the number of target nodes per host node, $K = N/p$, changes. We then compute speedup as the ratio of the time to simulate N target nodes on a single ($p = 1$) host node over the time to simulate N target nodes on p host nodes. We calibrate the model, for each application, by extracting parameters from a small number of fully-parallel ($K = 1$) simulation runs. Section 4 discusses the calibration and accuracy of the model, and

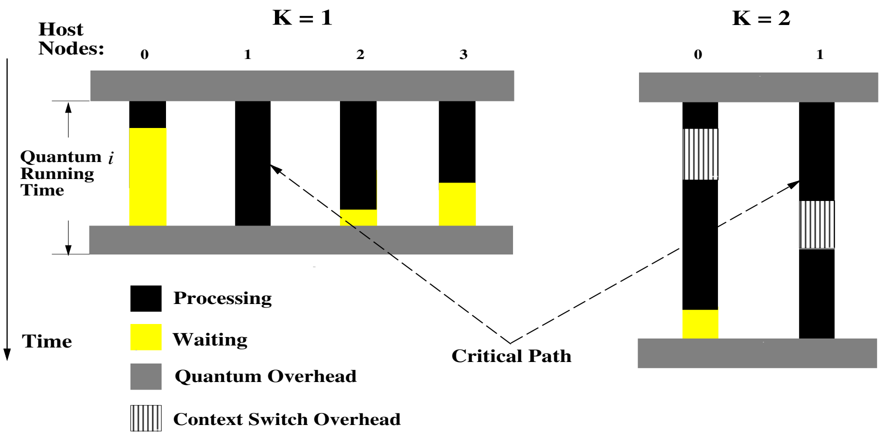

Fig. 1. Running Time of a Quantum

Section 5 describes how we extrapolate the model to estimate cost-performance for many different simulations.

Our model predicts the mean *running time* of a quantum. The running time of quantum i begins at the end of the $i-1$st quantum synchronization barrier and terminates at the end of ith quantum barrier, as illustrated in Figure 1. When simulating a single target node per host node ($K = 1$), two major factors dominate the quantum running time: *processing time*—the time to process simulation events including direct execution of the target program—and the *quantum overhead*—which consists of a barrier synchronization and the overhead of scheduling the target nodes. (The barrier synchronization uses the CM-5's "network done" operation to guarantee that all messages are received before the quantum completes, to ensure causality in the next quantum.) As illustrated in the left-hand side of Figure 1, the running time is simply the (fixed) quantum overhead plus the maximum processing time of any of the host nodes (e.g., host node 1 is on the critical path).

When there are multiple target nodes per host node ($K > 1$), the running time includes the sum of the processing times of each target node on the critical path. In addition, the simulation incurs both direct and indirect overheads from context switching. The direct overheads include the time to save and restore integer and floating-point registers. The indirect overheads occur because multiple target nodes compete for space in the host's cache memory and translation lookaside buffer (TLB), causing extra misses. The right-hand side of Figure 1 illustrates the case where two target nodes are simulated on each of two host nodes.

Finally, when all target nodes are simulated on a single host node ($K = N$), there can be no load imbalance so there is no waiting time. The quantum synchronization barrier becomes unnecessary and could be omitted; however, the overhead is insignificant compared to the $K = N$ processing times.

An ideal performance model for parallel simulation would accurately predict performance simply as a function of the number of host nodes, p, and the number of target nodes per host node, K. However, such a simple model cannot account for variations between target applications, target architectures, or host systems.

To accurately model WWT's performance, we found it necessary to break down the running time on the critical path into four components: the sum of the processing times (Section 3.1), the direct overheads (Section 3.2), the indirect overheads (Section 3.3), and the quantum synchronization time. Each of these submodels establishes a theoretical foundation to model the underlying physical behavior of these four primary performance factors. Our models for the components require a total of 14 parameters to be measured for each target application (and target architecture). Several of the parameters are simple event frequency measures, such as the mean number of active target nodes in a quantum. Others are timing measurements, such as the mean event processing time. Accurate, low-overhead timing measurements were made possible by the cycle counter in the CM-5's network interface unit. All parameters are simple to gather and can be extracted from four runs of the fully-parallel simulation, as described in Section 4.

We also experimented with simpler models that require fewer parameters, including an asymptotic approximation described in Section 6. However, these simpler models were unable to consistently capture the performance across the range of workloads, applications, and processors.

In the remainder of this section, we describe how we model each of the major contributors to simulation time: event processing time, direct context switch overhead, and host cache and TLB interference.

3.1 Modeling Processing Times

A potentially serious problem with conservative fixed-window simulation algorithms is that most host nodes will be idle while they wait for the slowest node to reach the barrier. In WWT, variations in event processing time are caused both by variation in the number of events that must be processed and in the time to process different types of events.

Within each quantum, a target node may process zero or more events. The (hopefully) common case is that a target node uses direct-execution to "simulate" local computation, including memory references that hit in its local cache. However, other events can occur, such as local cache misses and coherence messages from remote nodes. A target node may also have no events to process if, for example, the target program is waiting for a lock, barrier, or cache miss.

We have modeled this variability using a model that Kruskal and Weiss proposed for estimating the completion time of fork-join programs on MIMD parallel processors [Kruskal and Weiss 1985]. The model is asymptotically exact (as p and K go to infinity, with K growing faster than $\log_e p$) if the processing times are independent and identically distributed (i.i.d.) and the distribution function is increasing failure rate. However, they demonstrate that the model is remarkably robust even when these assumptions are violated.

In WWT, processing times are neither independent nor identically distributed. For example, when the target program uses barrier synchronization, target nodes that reach the barrier first will wait for the rest; since the barrier may span multiple quanta (due to target system load imbalances), the event processing times will be zero for all the waiting target nodes and hence are not independent. Moreover, a parallel program typically exhibits several distinct phases of execution, where the behavior of the program changes across phases, resulting in processing times that are not identically distributed over time. We have found that the lack of independence has little effect, perhaps since well-written parallel programs spend little time waiting for barriers, but that the phase behavior of programs is significant.

Kruskal and Weiss's model uses two parameters to characterize the workload: the mean μ and variance σ^2 of the processing times. We modify their model slightly, by using standard analysis-of-variance techniques to separate the variance within a quantum, σ^2_{intra}, from the variance of the entire population, σ^2 [Jain 1991]. This modification approximates the more technically correct, but computationally expensive, alternative of computing μ and σ^2 separately for each quantum. Our model for the mean processing time in a quantum, $T_{processing}$, is simply:

$$T_{processing}(K, p) = K\mu + \sigma_{intra}\sqrt{2K \log_e p} \tag{1}$$

The first term in the equation is simply the expected sum of the processing times on any host node.[1] The second term accounts for the quantum running time being determined by the slowest host node.

3.2 Modeling Direct Overhead

Because WWT uses a separate address space for each target node, it incurs a full context switch whenever it must simulate a different target node. The direct context switch overhead includes the time to save and restore both integer and floating-point registers. Since the CM-5's SPARC processor uses register windows, the time to restore integer registers must include factors for the mean number of underflow traps per context switch (N_{win}) and the mean underflow trap service time (T_{win}).

WWT includes several optimizations to eliminate unnecessary state saving on context switches. For example, because the simulator does not use any floating-point operations, WWT only restores the floating-point registers if the target CPU needs to execute. This reduces the overhead in the case that a target only needs to process non-target CPU events, e.g., directory messages.

WWT only schedules for simulation those target processes that are active—i.e., either have a running target CPU or a pending event—in a given quantum. Therefore, the number of context switches on the critical path is equal to the number of target nodes scheduled for simulation. To estimate how many of these fall on the critical path, we assume that the critical path has at least as many active target nodes as any other host node. Thus we can model the number of context switches as the expected maximum of p binomial random variables, each being the sum of K flips of a coin with probability equal to the probability that a given target process is active. Similarly, we predict the number of floating-point register saves and restores on the critical path using a "coin" whose probability is the fraction of target nodes in a quantum that directly execute CPU events.

Let N_{cxsw} denote the mean number target nodes that are active in a quantum. The probability of a target node being active would then be $\frac{N_{cxsw}}{N}$. Similarly, the probability of floating-point register saves and restores for a target process would

[1]Note that the mean μ includes the zero processing times that arise when a target node is blocked for some synchronization or protocol event

be $\frac{N_{fp}}{N}$, where N_{fp} denotes the mean number of target nodes with a running CPU in a quantum. Let $B(K, P)$ denote the binomial random variable equal to the sum of K flips of a coin with the probability P. The expected value of the maximum of p such i.i.d. binomial random variables is:

$$E[max_{i=1}^{p} B_i(K, P)] = \sum_{x=0}^{K}(1 - (P\{B(K, P) \leq x\})^p) \tag{2}$$

Note that since this expectation has no known closed-form solution, our overall performance model is computational, not closed form.

Denoting the mean service time for a context switch as T_{cxsw} and floating-point register saves and restores as T_{fp}, our model for the direct overhead is:

$$\begin{aligned} T_{direct}(K, p) \; = \; & E[max_{i=1}^{p} B_i(K, \frac{N_{cxsw}}{N})]\,(T_{cxsw} + N_{win}T_{win}) \\ & + E[max_{i=1}^{p} B_i(K, \frac{N_{fp}}{N})]T_{fp} \end{aligned} \tag{3}$$

This simple model proves to be extremely accurate.

3.3 Modeling Indirect Overhead

When multiple target nodes are simulated on the same host node, they compete for space in the host's cache memory and TLB. The interference that results has a first-order effect on simulation performance. Other researchers have seen similar effects in more generalized multiprogrammed, multithreaded and multiprocessor environments. Thiébaut and Stone [Thiébaut and Stone 1987] propose a model for predicting cache behavior for a multiprogrammed (multiple active processes) system. Agarwal et al. [Agarwal et al. 1989] enhance the model to estimate the performance of multithreaded processors, where context switches occur at (finer) cache miss intervals. Similarly, Mendelson et al. [Mendelson et al. 1990a] model interference due to coherence traffic among multiple processes in a multiprocessor system and the time-varying behavior of live and dead lines [Mendelson et al. 1990b].

We use a generalization of Thiébaut and Stone's *footprint* model to predict interference in the cache and TLB—i.e., a cache with a small number of sets—among multiple target processes on the same host node. Thiébaut and Stone's model estimates interference of two processes with disjoint address spaces in a cache. Because in WWT target processes share the text (instruction) segment of the simulator's address space, our model improves upon Thiébaut and Stone's by allowing sharing among address spaces of the interfering processes.[2] We further extend the model to allow for multiple (more than two) processes interfering in the cache.

In the following, we first describe our models for fully-associative and direct-mapped caches with two interfering processes. Then, we propose a technique for approximating interference among more than two processes. Because, our ultimate goal is to estimate the running time on the critical path of the simulation, we describe a technique for characterizing the behavior of processes on the critical

[2] To the best of our knowledge, our model is the first to allow for sharing of address spaces.

path. Finally, we introduce a technique for extracting the required parameters for the models from a fully-parallel simulation run.

3.3.1 *Model Assumptions and Terminology.* Thiébaut and Stone's model estimates the interference of two processes in the cache given their *footprints*. The footprint of a process is the set of unique (cache) blocks a process references in its address space. Assuming an infinite cache, the footprint corresponds to the set of blocks a process leaves in the cache upon a context switch. In a finite cache, some footprint blocks compete with others for placement. We define the *projection* of a process to be those footprint blocks which a process leaves in a finite cache.[3] We define *interference* to be the change in size of the projection of one process caused by another.[4]

In our models, process address spaces consist of two segments: a per process private segment, which is accessed exclusively by a given process, and a shared segment, which is accessed by all processes. Therefore, a process may interfere with another in two ways. As in Thiébaut and Stone's model, a process may reduce the projection of another by referencing blocks that replace blocks from the second process. In addition, in WWT a process may also increase the projection of another process by referencing blocks from the shared footprint, in effect prefetching blocks that may be used by the other process. Our models estimate the interference to be the difference of the above two effects; i.e., the number of blocks of one process replaced by a second process minus the number of shared blocks left in the cache by the second process.

Our models make the following assumptions:

—all sets are equally likely to be referenced,

—references within sets are independent and identically distributed,

—shared blocks in the projection of one process, are subsequently referenced by another,

—processes have similar shared footprints, and only differ in private footprints.

In the rest of this section, we first introduce models for estimating the interference of two processes in the TLB and cache; our host TLB is a fully-associative cache and our host cache is unified—i.e., it holds both instructions and data—and is direct-mapped. Second, we show how by using these models we can estimate the interference of multiple processes. Third, we show how to approximate the size of process footprints on the critical path. Finally, we present a technique for extracting the model parameters from a fully-parallel simulation run.

3.3.2 *Interference in a Fully-associative Cache.* This section describes a model for estimating the interference of two processes in a fully-associative cache. A fully-associative cache is one in which a single set contains all the block frames. Our model assumes that blocks from a process's footprint can occupy any of the frames. Given two processes accessing the cache consecutively, our model estimates how many blocks of the first process are replaced by the second process, and how many

[3] Thiébaut and Stone called this the "footprint in a finite cache".
[4] Thiébaut and Stone called this the "cache-reload transient".

shared blocks the second process leaves behind—to be subsequently accessed by the first process.

Much like Thiébaut and Stone, we first find the probability that the first process leaves a given number of blocks in the cache. We then find the joint probability that a second process replaces some of these blocks and leaves a given number of shared blocks behind. Finally, the interference is simply the expected value of the joint probability. Assume a fully-associative cache with N block frames. Given two processes A and B, let F_S denote the shared footprint and F_A and F_B denote the private footprints of the processes respectively. Assume A runs first and leaves a projection in the cache, and B subsequently runs and interferes with A's projection.

Let X denote the random variable representing the size of A's projection in the cache. We need to determine the probability of A's projection being of a given size. We know that A's footprint is of size $F_S + F_A$ and there are N block frames in the cache. Assuming that a footprint block can be placed in any frame, partitioning A's footprint among the frames is analogous to fitting $F_S + F_A$ customers in N queues [Lazowska et al. 1984]. Therefore, there are $\binom{F_S+F_A+N-1}{N-1}$ ways A can leave a projection in the cache. Similarly, in order for the projection to be of a given size, e.g., k, we will have to find the number of ways exactly k block frames in the cache are touched at least once. The latter is analogous to fitting $F_S + F_A$ customers in k non-empty queues, i.e., $\binom{F_S+F_A-1}{k-1}$. Moreover, there are $\binom{N}{k}$ ways to choose k block frames in the cache. It then follows that:

$$P\{X = k\} = \frac{\binom{N}{k}\binom{F_S + F_A - 1}{k - 1}}{\binom{F_S + F_A + N - 1}{N - 1}} \tag{4}$$

Assume A has left k blocks behind in the cache. When B runs, some of A's blocks will be replaced thereby reducing the size of A's projection. But, B will leave behind a number of shared blocks which are subsequently used by A, in effect increasing the size of A's projection. We will have to estimate how many blocks of A will be replaced by B and how many shared blocks B will leave behind in the cache. Let Z denote the random variable counting the number of A's blocks replaced by B, and S denote the random variable counting the shared blocks left behind by B. The quantity of interest would then be $P\{Z = i, S = j | X = k\}$.

Much like A, the number of ways B can leave a projection in the cache is $\binom{F_S+F_B+N-1}{N-1}$. Unlike A, we will have to distinguish the different orderings of shared and private blocks in B's footprint. Each projection may have $\binom{F_S+F_B}{F_S}$ possible orderings of shared and private blocks which will make the total number of possible projections $\binom{F_S+F_B}{F_S}\binom{F_S+F_B+N-1}{N-1}$.

Now, we will find the number of ways B can replace i blocks of A and leave j shared blocks behind. Let w be the size B's projection. There are $\binom{k}{i}$ ways to replace i blocks of A, $\binom{N-k}{w-i}$ ways to choose frames in the rest of the cache, and $\binom{w}{j}$ ways to choose j frames containing shared blocks. Therefore, the number of ways we can choose w block frames, i of which belong to A and j of which contain shared blocks is $\binom{k}{i}\binom{N-k}{w-i}\binom{w}{j}$. Given our selection of w block frames, the number of ways we can partition B's footprint in these frames—touching all of them at least once—is $\binom{F_S+F_B-1}{w-1}$. Since j of the frames contain shared blocks and $w-j$ of them contain private blocks, the rest of the footprint can be ordered in $\binom{F_S+F_B-w}{F_S-j}$ ways. Hence:

$$P\{Z=i,S=j|X=k\} = \frac{\sum_{w=max(i,j)}^{N} \binom{k}{i}\binom{N-k}{w-i}\binom{w}{j}\binom{F_S+F_B-w}{F_S-j}\binom{F_S+F_B-1}{w-1}}{\binom{F_S+F_B}{F_S}\binom{F_S+F_B+N-1}{N-1}} \tag{5}$$

and

$$P\{Z=i,S=j\} = \sum_{k=0}^{N} P\{X=k\}P\{Z=i,S=j|X=k\} \tag{6}$$

Let $I_{TLB}(F_S,F_A,F_B)$ denote the expected value of B's interference with A in the TLB, then:

$$I_{TLB}(F_S,F_A,F_B) = \sum_{i=0}^{N}\sum_{j=0}^{N}(i-j)P\{Z=i,S=j\} \tag{7}$$

3.3.3 *Interference in a Direct-mapped Cache.* In this section we describe a model for estimating the interference of two processes in a direct-mapped cache. A direct-mapped cache is one in which each set contains only a single block frame. Our model assumes that footprint blocks are equallly likely to land in any of the cache sets. Consequently, we estimate the interference in a single set, and multiply the result by the number of sets to find the interference in the cache. We can treat each set of a direct-mapped cache as a fully-associative cache and estimate the interference using the model in Section 3.3.2. Unlike a fully-associative cache, however, not all of a process's footprint land in one set of a direct-mapped cache. Therefore, we need to account for the fraction of a process's footprint that lands in a specific set of the cache. Moreover, because a direct-mapped cache contains only a single block frame in each set, we can develop somewhat simpler models than those described in Section 3.3.2.

Assume there are two processes A and B accessing the cache with a shared footprint of size F_S and private footprints of sizes F_A and F_B respectively. A first runs and leaves a projection in the cache, and B subsequently runs and interferes with A's projection. Assume a direct-mapped cache with N sets. Since references to sets are assumed independent and identically distributed, each reference is a Bernoulli experiment with probability $p=\frac{1}{N}$ of landing in a specific set (success), and probability $q=1-p$ of landing in any other set (failure). The size of A's projection in a set will then follow a tail binomial distribution:

$$P\{X=k\} = \begin{cases} q^{F_S+F_A} & \text{if k=0} \\ 1-q^{F_S+F_A} & \text{if k=1} \end{cases} \tag{8}$$

We now need to find whether B leaves a projection in the set and if so whether the projection contains a shared block. Since, some of B's footprint does not land in the set of interest, we first find the probability that a given portion of B's footprint lands in the set. Let Y denote the random variable representing the number of blocks in B's footprint landing in the set. Because B's references are also Bernoulli experiments, Y follows a binomial distribution:

$$P\{Y = l\} = \binom{F_S + F_B}{l} p^l q^{F_S+F_B-l} \tag{9}$$

Assume for now that B's footprint in the set is of size $l \geq 1$ blocks. We would like to find the probability that the last reference to the set is a shared block. Let m denote the number of shared blocks in B's footprint landing in the set. There are $\binom{F_S}{m}$ ways to choose shared blocks and $\binom{F_B}{l-m}$ ways to choose private blocks and $\binom{l}{m}$ ways to order the private and shared blocks in B's footprint. Therefore, the total number of orderings of B's footprint in the set is $\binom{F_S}{m}\binom{F_B}{l-m}\binom{l}{m}$. Similarly, if the last block is shared, the rest of the blocks in the footprint can be ordered in $\binom{F_S}{m}\binom{F_B}{l-m}\binom{l-1}{m-1}$ ways. Let S denote the random variable counting the number of shared blocks in B's projection. Therefore, for $l \geq 1$ the probability is:

$$\begin{aligned} P\{S = 1 | Y = l\} &= \frac{\sum_{m=1}^{min(l,F_S)} \binom{F_S}{m}\binom{F_B}{l-m}\binom{l-1}{m-1}}{\sum_{m=1}^{min(l,F_S)} \binom{F_S}{m}\binom{F_B}{l-m}\binom{l}{m}} \\ P\{S = 0 | Y = l\} &= 1 - P\{S = 1 | Y = l\} \end{aligned} \tag{10}$$

Summing over all possible values of l, we obtain:

$$P\{S = j, Y \geq 1\} = \sum_{l=1}^{F_S+F_B} P\{S = j | Y = l\} P\{Y = l\} \tag{11}$$

We now proceed to find the probability that B replaces $i \in \{0, 1\}$ blocks from A and leaves $j \in \{0, 1\}$ shared blocks in the set. Let Z denote the random variable counting the number of blocks in A's projection replaced by B. Then,

$$P\{Z = i, S = j\} = \begin{cases} P\{Y = 0\} + P\{X = 0\}P\{S = j, Y \geq 1\} & \text{if i=0} \\ P\{X = 1\}P\{S = j, Y \geq 1\} & \text{if i=1} \end{cases} \tag{12}$$

B's interference with A in the cache is then:

$$I_{cache}(F_S, F_A, F_B) = N(P\{Z = 1, S = 0\} - P\{Z = 0, S = 1\}) \tag{13}$$

3.3.4 *Interference among Multiple Processes.* The models we have developed in Sections 3.3.2 and 3.3.3 estimate interference of exactly two processes in the cache and TLB. In WWT, there may be multiple (more than two) target processes simulating on a single host node. In the following, we apply a simple extension to

Thiébaut and Stone's footprint model to allow for estimating interference among more than two processes.

We approximate the effect of multiple processes with given footprints by a single large process with a footprint equal to the sum of the footprints from the constituent processes. For instance, assume three processes A, B and C with footprints of size F_A, F_B and F_C respectively. Assume A, B and C run and access the cache in that order. In order to estimate how B and C interfere with A, we approximate their combined effect with a single process having a footprint of size $F_B + F_C$. Because our models assume that all processes have similar shared footprints, we only sum the private footprints in our approximation of multiple processes.

3.3.5 *Interference on the Critical Path.* Our goal is to predict the running time of a quantum. As discussed in Section 3, the latter includes the sum of the running times of each target process on the critical path. Although our measurements from a fully-parallel simulation run may reveal the size of all process footprints, they fail to indicate which target processes lie on the critical path of a given configuration of target nodes per host node. In the following, we describe a technique for characterizing the size of process footprints on the critical path.

One way to characterize the size of process footprints is to simply use the mean footprint size. Target processes on the critical path, however, tend to have longer running times and as such their footprints are typically larger than an average process. Assuming that the process with the largest footprint always lies on the critical path, we characterize critical path processes as one process with the largest footprint and the rest of the processes having average size footprints.

Let F_S denote the size of the shared footprint. Let F_{avg} and F_{max} denote the average and maximum size of the private footprints respectively. Let K denote the number of target processes per host node. We characterize critical path processes as $K-1$ processes with F_{avg} size private footprints and a single process with F_{max} size private footprint. We now need to estimate the interference among the processes. Let I denote the interference in either cache or TLB. The interference of $K-1$ processes with average footprints and the process with maximum footprint is simply $I(F_S, F_{max}, (K-1)F_{avg})$. The interference of all processes—i.e., $K-2$ processes with average footprints and the process with maximum footprint—with each of the average processes is $I(F_S, F_{avg}, (K-2)F_{avg} + F_{max})$. Let M denote the overall interference in terms of the number of block misses, then:

$$M(K) = I(F_S, F_{max}, (K-1)F_{avg}) + (K-1)I(F_S, F_{avg}, (K-2)F_{avg} + F_{max}) \quad (14)$$

Let $T_{cachemiss}$ and $T_{TLBmiss}$ denote the mean service time for a cache and TLB miss respectively. Let M_{cache} and M_{TLB} denote the number of interference misses in the cache and TLB as predicted by Equations 7, 13, 14. The model for the indirect overhead, $T_{cache\&TLB}$, will then be:

$$T_{cache\&TLB}(K) = M_{cache}(K)T_{cachemiss} + M_{TLB}(K)T_{TLBmiss} \quad (15)$$

3.3.6 *Parameter Extraction.* In this section we describe a technique for extracting the size of process footprints from a fully-parallel simulation run. There are a myriad of software [Larus and Schnarr 1995; Eggers et al. 1990] and hardware

[Agarwal et al. 1986] techniques for address trace generation. These techniques are typically very general and proved to be an overkill for our purposes; we need to find the number—and not the identity—of unique blocks in the reference stream of a process. Instead, we opted for a simpler technique that measures the projection rather than the footprint itself.

We first individually measure the running time of every target process. We then repeat the measurement while flushing the cache (TLB) before running a target process. The difference in measurements are solely due to the initial misses to each block frame in the cache (TLB). We use prior knowledge of the average cache (TLB) miss penalty to calculate the number of misses. The latter is the size of the projection of a process in the cache (TLB). Given the size of the projection, we use a reverse mapping of the projection probability functions (Equations 4 and 8) to estimate the size of the footprints. We then estimate the interference using these desired footprints. Note that using derived, rather than measured, footprints may reduce the model's error. In order to isolate private footprints from shared footprints, we repeat the above measurements, this time only flushing the simulator's instruction space.

3.4 Running Time of a Quantum

Putting the three submodels from Equations 1, 3 and 15 together, along with the fixed quantum overhead.
Finally, $T_{quantumoverhead}$, allows us to estimate the mean running time of a quantum:

$$\begin{aligned} T(K,p) &= T_{processing}(K,p) + T_{direct}(K,p) + T_{cache\&TLB}(K) \\ &\quad + T_{quantumoverhead} \end{aligned} \tag{16}$$

where p is the number of host processors, and $K = N/p$ is the number of target nodes per host node.

4. VALIDATING THE MODEL FOR $T(K,P)$

We validate the model by simulating a 32-node cache-coherent shared-memory multiprocessor with a 4-way set-associative 32-Kbyte cache kept coherent using the Dir_1SW coherence [Hill et al. 1993; Wood et al. 1993] and a 64-entry fully-associative TLB. The network latency (and hence quantum length) is 100 cycles. The target system executes in one of two phases. A serial phase in which shared memory is allocated and mapped on all nodes, and a parallel phase in which a single thread of execution is initiated on every node. Since we are interested in the behavior of the simulator when all target nodes have started executing threads, we only focus on the portion of the simulation corresponding to the parallel execution of code on the target nodes.

Table 1 depicts the model parameters we extract from the simulation. With a single run of a fully-parallel simulation, we extract the parameters associated with estimating the processing times and the direct overhead (first eight parameters in the table). As discussed in Section 3.3.6, we estimate mean footprint sizes by measuring the corresponding projection size. Measuring the projection of a process in the cache (TLB) requires a separate run, in which the cache (TLB) is flushed before simulating a target process in every quantum. Furthermore, we require

Table 1. Extracted Model Parameters

Name	Description
σ^2_{intra}	Variance in target processing time
	Mean Time in Cycles
μ	Processing
T_{cxsw}	Context switch
T_{win}	Register window underflow
T_{fp}	Floating-point registers save/restore
	Mean Frequency
N_{cxsw}	Context switch
N_{win}	Register window underflow
N_{fp}	Floating-point registers save/restore
	Mean Size in each of Cache & TLB
F_S	Shared footprint
F_{avg}	Average footprint
F_{max}	Maximum footprint

Table 2. Application Programs

Name	Input Data Set	Million Cycles
appbt	12 × 12 × 12, 15 iter	124
barnes	1024 bodies, 10 iter	95
sparse	256 × 256 dense	86
tomcatv	256 × 256, 10 iter	28
water	256 mols, 10 iter	49

separate runs in order to measure the projection of shared and private footprints. Therefore, with a total four additional runs of fully-parallel simulations, we can measure the parameters for the indirect overhead.

Table 2 depicts the benchmarks used to run on the simulated system. *Appbt* is a computational fluid dynamics program that solves systems of tridiagonal equations [Bailey et al. 1991]. *Sparse* solves $AX = B$ in parallel for a sparse matrix A. *Tomcatv* is a parallel version of the SPEC benchmark [SPEC 1990]. *Barnes* and *Water* are from the SPLASH benchmarks [Singh et al. 1992]. The data sets used in this study are much smaller than one would normally run. However, in order to measure speedup, the data set had to be small enough so that we could simulate all 32 nodes of the target system on a single CM-5 node (with 32 megabytes of memory). Because the small data sets limit the available parallelism in the target programs—resulting in poor target speedups—we expect the results in this paper to be conservative. Simulations of larger data sets achieve better speedups than we observe here.

We first compare our estimate of processing time, $T_{processing}(K,p)$, against the measured sum of processing times on the critical path. The left-hand side of Figure 2 illustrates these times as speedup: the sum of all $N = 32$ processing times divided by (a) $T_{processing}(K,p)$ and (b) the measured sum of K processing times on the critical path with p host nodes. We can make two observations from these graphs. First, the model is quite accurate for *appbt*, *barnes*, and *tomcatv*, but consistently underestimates the speedup for *sparse* and *water*, with a maximum observed error of -24% for *sparse* on a 4-node host system. The model is more accurate at the

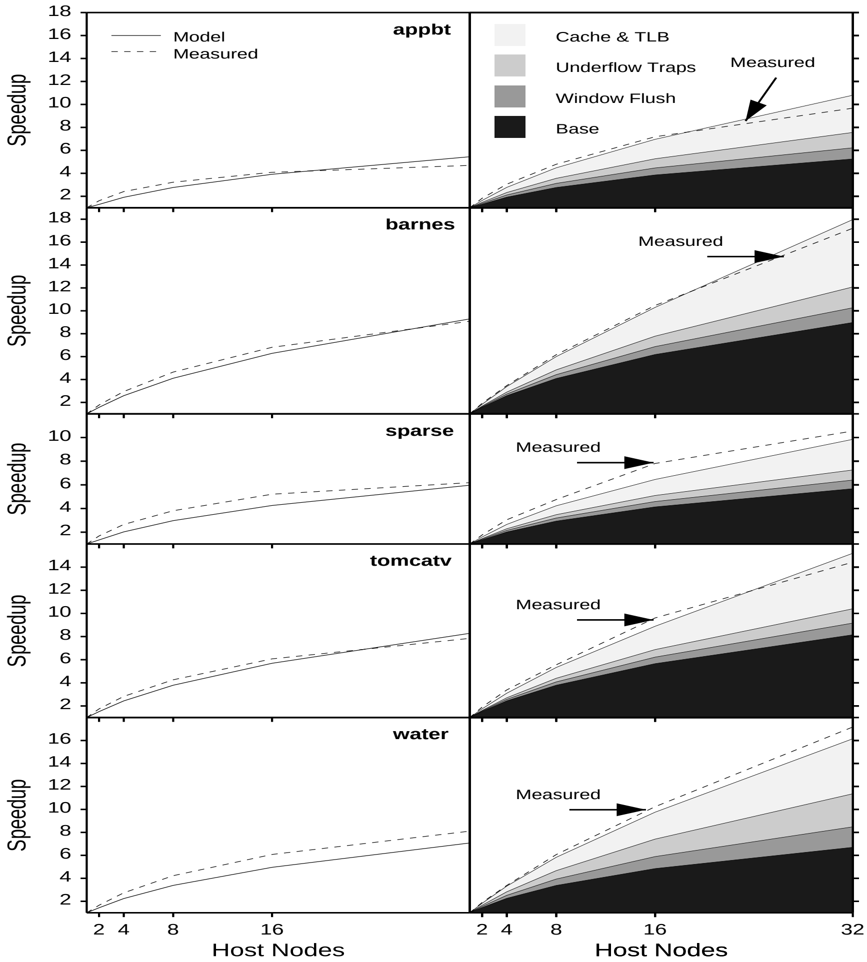

Fig. 2. Speedup of Parallel Simulation

extremes: it is exact, by definition, when $p = 1$, and the error is less than 16% for $p = 32$. The second, more fundamental observation, is that the inherent simulation parallelism is low, only providing speedups ranging from 4 to 9 on 32 host nodes. This is at least partially due to the low target system speedups these programs achieve for the small data sets used in this study.

Despite the relatively low "inherent" parallelism in event processing times, the Wisconsin Wind Tunnel actually achieves acceptable overall speedups, as illustrated in the right-hand side of Figure 2. The figure plots the overall speedups (a) as predicted by the model—i.e., $T(32, 1)/T(K, p)$—with a breakdown into the contribution of major components and (b) as measured. *Base* represents the sum of processing times in conjunction with floating point register save and restore, and quantum overheads—the contribution of the latter two is relatively small. The central observation is that overhead increases the simulation parallelism by up to a factor of two. This result is consistent with additional measurements which indicate that overhead accounts for 44% to 68% of the computation in a sequential WWT simulation. These overheads not only decrease as the host nodes increase, but, to the first order, they are perfectly parallelizable. Therefore, parallel simulation benefits both from processing simulation events in parallel and distributing the overhead across multiple host nodes.

The figure also illustrates that cache and TLB interference causes significant overhead; measurements indicate that it accounts for up to 30% of the running time when $K > 4$. When more than 4 target nodes compete for the same cache and TLB, our model and measurements show that almost none of a target process's private footprint remains in the cache (TLB) when it is rescheduled, and thus most private memory references miss. Parallelizing the simulation reduces the number of target nodes per host node, and hence reduces the number of cache misses on the critical path of the computation.

Although the absolute error in total speedup is roughly the same as the error due to $T_{processing}(K,p)$, the relative error is roughly half as big because overhead accounts for half the simulation time and we model overhead accurately. For four of the benchmarks the maximum error in the model is less than 12%. For *sparse* the maximum error is -16% for a 16-node host system.

5. MODELING COST/PERFORMANCE

The model introduced in Section 3 accurately predicts simulation performance for a target system with N nodes. Section 5.1 describes how we extend the model to estimate simulation performance for both larger and smaller values of N. Section 5.2 introduces our cost models for uniprocessor, bus-based shared-memory multiprocessor, and massively-parallel processor systems. Section 5.3 combines the cost and performance models to estimate cost/performance.

5.1 Scaling the Performance Model

The performance model developed in Section 3 extracts parameters from a fully-parallel ($p = N$) simulation of a specific target system, and uses them to predict the performance of that same simulation running on different numbers of host nodes. The model, however, says nothing about the simulation performance of larger or smaller target systems. To extend the model, we must make several assumptions about how the target and simulation systems scale.

We assume memory-constrained scaling [Jaswinder P. Singh and Gupta 1993] when we vary the size of the target system. In memory-constrained scaling, the data set size grows linearly with respect to the number of (target) nodes. This scaling model has two key properties. First, application parallelism generally increases at least linearly with data set size, so target system speedup should not limit simulation speedup. Second, this model tends to have only a minor effect on the computation/communication ratio, so that simulation processing times should have roughly the same distribution independent of N. Consequently, we can still use the mean and variance measured for a 32-node system to characterize this distribution. The Kruskal and Weiss model, used to compute $T_{processing}(K,p)$, will account for the increased (decreased) variability of larger (smaller) target system sizes.

We further assume that the overhead of multiplexing target nodes on a host node is independent of the number of host nodes, and use our earlier estimates to approximate the overhead for different target system sizes. For $K \leq 32$, we use our earlier estimates of the context switch frequency, and cache and TLB interference. We estimate the context switch frequency for $K > 32$, by linearly extrapolating the binomial model; since the tail of this curve is very close to linear, we do not expect this to introduce a significant error. We approximate the cache and TLB interference for $K > 32$, by simply using the estimated interference for $K = 32$; since both cache and TLB begin thrashing for more than 4 target nodes per host node, there will be essentially no reuse (i.e., hits) for large K.

5.2 Modeling the Cost of Host Systems

In this section, we introduce cost models for uniprocessors (Uni), small-scale bus-based shared-memory multiprocessors (Bus), and large-scale parallel supercomputers (MPP). The cost models are based on commercial products and allow us to vary the number of host processors, p, and the number of target nodes per host node, K. We assume that each host node requires 32 megabytes per target node. This is significantly more than needed for the small data sets used in this study; however, these data sets were chosen so that we could simulate 32 target nodes within 32 megabytes of memory (i.e., on one CM-5 node). Real data sets are much larger; for example, the official NAS input to *appbt* is 125 times larger than the data set presented here [Bailey et al. 1991].

Our uniprocessor cost model is based on the Silicon Graphics CHALLENGE M, a rack-mounted uniprocessor workstation server. We use a server configuration because desktop and deskside units do not provide the necessary memory expansion capability [Reidenbach 1993]. For a target system of size K, we model the cost of a uniprocessor simulation platform as:

$$C_{Uni}(K) = BaseC_{Uni} + C_{processor} + KC_{memory} \tag{17}$$

where $BaseC_{Uni}$ denotes the base cost of the frame (box, power supply, etc.), $C_{processor}$ denotes the cost of a processor board excluding memory, and C_{memory} denotes the cost of a 32-megabyte memory module.

Bus-based shared-memory multiprocessors consist of a frame containing a variable number of processor and memory boards connected by a backplane bus. This cost model is based on the Silicon Graphics CHALLENGE XL system.[5] The cost of a bus-based host system with p processors simulating a target system of size Kp (for all $2 \leq p \leq 40$) is:

$$C_{Bus}(K,p) = BaseC_{Bus} + pC_{processor} + KpC_{memory} \tag{18}$$

where $BaseC_{Bus}$ is the base cost for the frame.

Current implementations of massively parallel processors consist of a collection of workstation-like processing nodes connected together by a high-bandwidth interconnection network. Our cost model for these systems does not include a fixed base cost because they are generally expanded by adding entire cabinets, rather than individual processor boards. Rather than try and capture the complex step function of the actual cost, we simply approximate it as a linear function of p; this

[5]The same cost model, with different parameter values, also accurately predicts the Sun Ultra-Server systems [Roessler 1996].

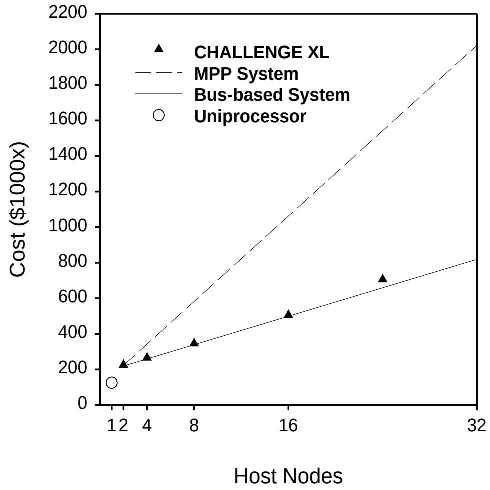

Fig. 3. Modeling the Cost of Host Systems

approximation should not introduce significant error since we only consider values of p that are powers of two. Modeling the network cost as a multiplier, $X_{network}$, of the processor cost, the overall cost (for all $p \geq 2$) is:

$$C_{MPP}(K,p) = p(1 + X_{network})C_{processor} + KpC_{memory} \tag{19}$$

For the purposes of this study, we use 1993 Silicon Graphics list prices for our uniprocessor and shared-memory multiprocessor cost estimates: $C_{processor}$ = \$20000, C_{memory} = \$3200 (32 megabytes), $BaseC_{Uni}$ = \$3200, and $BaseC_{Bus}$ = \$76800 [Reidenbach 1993]. We assume $X_{network} = 2$, which reflects the fact that current generation MPP nodes cost approximately three times a comparable uniprocessor. Ultimately, we expect competition to reduce $X_{network}$ to values of $0.1 \sim 0.5$.

Although these prices are valid for only a single point in the past, we believe that the general conclusions will hold for the future. This is because although memory cost per bit is dropping [Seminconductor Industry Association 1994], memory sizes continue to increase [Przybylski 1994] and memory cost appears to be an increasing fraction of total uniprocessor system cost.

Figure 3 plots the cost models as a function of the number of host nodes for a 32-node target system. The minimum cost of a parallel host is approximately two times the cost of a uniprocessor host system. The figure also depicts the prices of Silicon Graphics CHALLENGE XL bus-based multiprocessor servers [Reidenbach 1993]. The cost curve for the massively parallel processors has a much steeper slope as compared to the curve of the bus-based multiprocessors due to the high cost of the interconnection network per node.

5.3 Modeling Cost/Performance

Since speedup is a measure of parallel simulation performance, cost/performance is simply the cost of the host system divided by the simulation speedup it achieves. For a uniprocessor system, the cost/performance is simply C_{Uni}, because speedup is 1 by definition. Cost/performance for parallel simulation of a Kp-node target system is:

$$CP_{Machine}(K,p) = \frac{T(K,p)}{T(Kp,1)} C_{Machine}(K,p) \tag{20}$$

where $Machine$ is either Bus or MPP.

6. ANALYZING COST/PERFORMANCE

Given a model for parallel simulation cost/performance, there are two questions that we would like to address. First, is parallel simulation simply faster than sequential simulation, or is it also more cost-effective? Second, if we have parallel simulation, what value of K achieves the best cost/performance?

We address the second question first, by analyzing the asymptotic behavior of K_{min} using a simplified form of the cost/performance model. We know that such a minimum value exists, because the cost function is increasing linear in p and the speedup is a bounded convex function in p. Therefore, for a fixed target system size, the cost/performance function is concave with a minimum at $K = K_{min}$. We simplify the model slightly to clarify the asymptotic analysis. We approximate the running time of a quantum as $aK + b\sqrt{K \log_e p}$, where the first term accounts for factors contributing linearly to the running time such as the mean processing time of target nodes and the per-target node overhead on the critical path, and the second term accounts for the variation in the sum of processing times on the critical path. We also approximate the cost function as $p(C_{processor} + KC_{memory})$. The cost/performance function will then be:

$$CP_{asymptotic}(K,p) = C_{memory}(K + \frac{C_{processor}}{C_{memory}})(1 + \frac{b}{a}\sqrt{\log_e p}\frac{1}{\sqrt{K}}) \tag{21}$$

For a given target system size (i.e., fixed N), the above function has a minimum at K_{min} which is an increasing function of $C_{processor}/C_{memory}$, b/a and $\sqrt{\log_e p}$. Since the variation of the latter is negligible in the range of feasible values of p, a key contribution of this model is that K_{min} is, to the first order, independent of p.

The term b/a is reciprocally proportional to the amount of parallelism available in the simulation. Small values of b/a can result from either (a) processing times that have a small coefficient of variation, and thus cause little load imbalance, or (b) small mean processing times which cause the—perfectly parallelizable—overhead to dominate. In either case, high parallelism results in small values of K_{min}. This result is intuitive, since higher parallelism gives rise to larger speedups which in turn offset the cost of adding more host nodes.

The model also predicts that a decrease in the cost of memory with respect to the cost of a processor board results in a larger value of K_{min}. The intuition behind this result is that, for a given target system size, decreasing memory cost increases relative processor cost, and shifts the balance toward more memory intensive simulations.

Unfortunately, analyzing the simplified model does not help us answer the first question of when is parallel simulation better than sequential simulation. Instead, we graphically examine the full model.

Figure 4 plots $CP_{Bus}(K,p)$, $CP_{MPP}(K,p)$, and $C_{Uni}(Kp)$ for the simulation of *appbt* and *barnes*. These two applications are reasonably representative of the

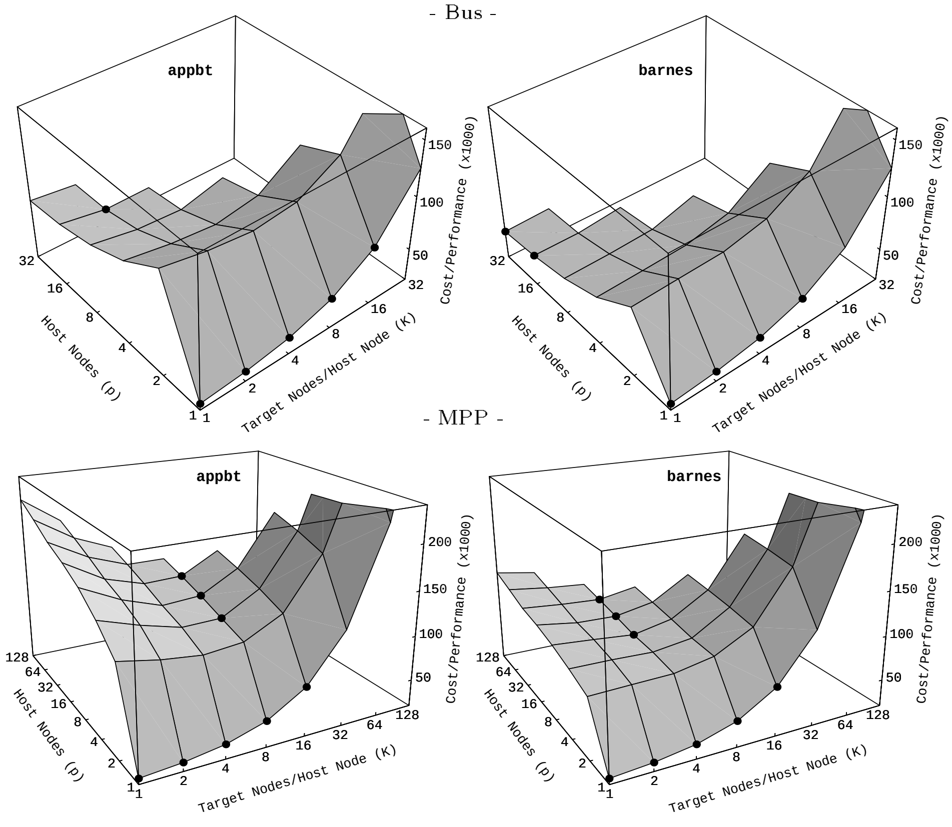

Fig. 4. Cost/Performance of Parallel Simulation

five benchmarks: *appbt* and *sparse* exhibit relative low speedups, while *barnes*, *tomcatv*, and *water* exhibit considerably higher speedups. The graphs use the same cost parameters as those in Figure 3. Moreover, a bullet at coordinates (K, p) indicates the minimum cost/performance for the simulation of a target system of size Kp.

The upper half of Figure 4 plots $CP_{Bus}(K, p)$ and $C_{Uni}(Kp)$ for $Kp \leq 32$. The graphs show that although uniprocessor simulation is more cost-effective for small target systems, up to 8 or 16 nodes, parallel simulation offers superior cost/performance as the target system grows beyond approximately 16 nodes. More important than the exact numbers, the trend clearly shows that parallel simulation becomes increasingly cost-effective as the target system grows.

The second interesting prediction of this model is the lack of continuity in p. That is, parallel simulation does not gradually become more effective, but rather once the speedup is sufficient to overcome the large base cost, the optimum cost/performance occurs when the simulation is either fully parallel (*barnes*) or nearly so (*appbt*).

The lower half of Figure 4 plots $CP_{MPP}(K, p)$ and $C_{Uni}(Kp)$ for $Kp \leq 128$. The figure illustrates that uniprocessor simulation is more cost-effective than parallel simulation for target systems of up to 16 nodes. The large jump in cost/performance as p increases from 1 to 2 nodes is due to the significant premium charged by MPP vendors. For host systems of up to 16 nodes, the simulation speedup is not large enough to offset this premium and therefore uniprocessor simulation offers better cost/performance. Minimum cost/performance for larger systems lies consistently at 4 and 8 target nodes per host node for the two benchmarks independent of the number of host nodes. Moreover, the larger parallelism available in the simulation of *barnes* results in optimum cost/performance at a smaller value of K. These results are in accord with the predictions of our model for the asymptotic behavior of K_{min}.

Decreasing memory cost not only shifts K_{min} towards larger values—confirming our analysis of the simplified model—but increases the target system size at which parallel simulation becomes more cost-effective than sequential simulation. For example, decreasing the memory cost (or, equivalently, the simulation's memory requirement) by a factor of four increases the break-even point for parallel simulation to 128 target nodes.

Decreasing the processor cost (and/or the cost of the network for MPP's) has a complementary effect, not only decreasing K_{min}, but reducing the break-even target system size. Similarly, increasing the parallel simulation speedups, as we expect for larger data sets, will also tend to make parallel simulation increasingly cost-effective.

7. SUMMARY AND CONCLUSIONS

This paper examines the cost/performance of simulating a hypothetical *target* parallel computer using a commercial *host* parallel computer. We address the fundamental question of whether parallel simulation is simply faster than sequential simulation, or whether it is also more cost-effective. We answer this by developing a performance model of the Wisconsin Wind Tunnel (WWT) that incorporates three major factors: event processing time, context switch overhead, and host cache and TLB interference. For the performance model, we show that:

—the variability in event processing times can be accurately modeled using Kruskal and Weiss's model for fork-join parallel programs;

—the frequency of context switches, incurred when switching between target nodes, is accurately modeled by the maximum of binomial random variables;

—an extension of Thiébaut and Stone's *footprint* model accurately predicts the interference of multiple targets in the host cache and TLB;

—the performance model's predictions of simulation speedup are within 10% on average and are always within 20% for these workloads.

We then combine the performance model with simple cost models and show—in the central result of this paper—that parallel computer simulations can be, in fact, more cost-effective than uniprocessor simulations. The key intuition behind this result is that large simulations require large memory sizes, which dominate the cost of a uniprocessor; parallel computers allow multiple processors to simultaneously access this large memory. Furthermore, we show that K_{min}, the number of target nodes simulated per host node for optimum cost/performance, is essentially independent of p, the number of host processors. For two specific cost models and the

WWT performance model, we show that (1) for bus-based shared-memory multiprocessors, parallel simulation becomes more cost-effective when target systems reach 16 or 32 nodes, and (2) for massively parallel systems, with their large price premium, parallel simulation becomes more cost-effective when the target system size reaches 32. More generally, since memory cost is an increasing fraction of overall uniprocessor system cost, we expect this result to qualitatively hold for future parallel systems.

ACKNOWLEDGMENTS

This work is part of the Wisconsin Wind Tunnel project, which is co-led by Profs. Mark Hill, James Larus, and David Wood. We would like to especially thank Steve Reinhardt for all his contributions to developing the Wisconsin Wind Tunnel. We would like to thank Alain Kägi, Rajesh Mansharamani and Vikram Adve for their valuable insights that helped us refine the performance models. We would also like to thank Jignesh Patel, and the anonymous reviewers for their helpful comments on earlier drafts of this paper.

REFERENCES

AGARWAL, A., HOROWITZ, M., AND HENNESSY, J. 1989. An analytical cache model. *ACM Transactions on Computer Systems 7*, 2 (May), 184–215.

AGARWAL, A., SITES, R. L., AND HORWITZ, M. 1986. ATUM: A new technique for capturing address traces using microcode. In *Proceedings of the 13th Annual International Symposium on Computer Architecture* (June 1986), pp. 119–127.

AYANI, R. 1989. A parallel simulation scheme based on the distance between objects. In *Proceedings of the SCS Multiconference on Distributed Simulation* (March 1989), pp. 113–118.

BAILEY, D., BARTON, J., LASINSKI, T., AND SIMON, H. 1991. The nas parallel benchmarks. Technical Report RNR-91-002 Revision 2 (Aug.), Ames Research Center.

BOOTHE, B. 1992. Fast accurate simulation of large shared memory multiprocessors. Technical Report CSD 92/682 (Jan.), Computer Science Division (EECS), University of California at Berkeley.

BREWER, E. A., DELLAROCAS, C. N., COLBROOK, A., AND WEIHL, W. 1991. Proteus: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516 (Sept.), MIT Laboratory for Computer Science.

CMELIK, R. F. AND KEPPEL, D. 1994. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (May 1994).

COVINGTON, R., MADALA, S., MEHTA, V., JUMP, J., AND SINCLAIR, J. 1988. The Rice parallel processing testbed. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1988), pp. 4–11.

DAVIS, H., GOLDSCHMIDT, S. R., AND HENNESSY, J. 1991. Multiprocessor simulation and tracing using tango. In *Proceedings of the 1991 International Conference on Parallel Processing (Vol. II Software)* (Aug. 1991), pp. II99–107.

EGGERS, S. J., KEPPEL, D. R., KOLDINGER, E. J., AND LEVY, H. M. 1990. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1990), pp. 37–47.

FALSAFI, B. AND WOOD, D. A. 1994. Cost/performance of a parallel computer simulator. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)* (July 1994).

FUJIMOTO, R. M. 1983. Simon: A simulator of multicomputer networks. Technical Report UCB/CSD 83/137, ERL, University of California, Berkeley.

FUJIMOTO, R. M. 1990. Parallel discrete event simulation. *Commun. ACM 33*, 10 (Oct.), 30–53.

HILL, M. D., LARUS, J. R., REINHARDT, S. K., AND WOOD, D. A. 1993. Cooperative shared memory: Software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems 11*, 4 (Nov.), 300–318. Earlier version appeared in *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*.

HILLIS, W. D. AND TUCKER, L. W. 1993. The cm-5 connection machine: A scalable supercomputer. *Commun. ACM 36*, 11 (Nov.), 31–40.

JAIN, R. 1991. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley & Sons.

JASWINDER P. SINGH, J. L. H. AND GUPTA, A. 1993. Scaling parallel programs for multiprocessors: Methodology and examples. *IEEE Computer 26*, 7 (July), 42–50.

KRUSKAL, C. P. AND WEISS, A. 1985. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering 11*, 10 (Oct.), 1001–1016.

LARUS, J. R. AND SCHNARR, E. 1995. Eel: Machine-independent executable editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)* (June 1995), pp. 291–300.

LAZOWSKA, E. D., ZAHORJAN, J., GRAHAM, G. S., AND SEVCIK, K. C. 1984. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice Hall.

LEISERSON, C. E., ABUHAMDEH, Z. S., DOUGLAS, D. C., FEYNMAN, C. R., GANMUKHI, M. N., HILL, J. V., HILLIS, W. D., KUSZMAUL, B. C., PIERRE, M. A. S., WELLS, D. S., WONG, M. C., YANG, S.-W., AND ZAK, R. 1993. The network architecture of the connection machine cm-5. In *Proceedings of the Fifth ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (July 1993).

LI, K. AND HUDAK, P. 1989. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems 7*, 4 (Nov.), 321–359.

LIN, J.-J. 1992. Efficient parallel simulation for designing multiprocessor system. Ph. D. thesis, University of Michigan, Ann Arbor.

LUBACHEVSKY, B. D. 1989. Efficient distributed event-driven simulations of multiple-loop networks. *Commun. ACM 32*, 2 (Jan.), 111–123.

MENDELSON, A., THIÉBAUT, D., AND PRADHAN, D. 1990b. Modeling live and dead lines in cache memory systems. Technical Report TR-90-CSE-14, Department of Electrical and Computer Engineering, University of Massachusetts.

MENDELSON, A., THIÉBAUT, D., AND PRADHAN, D. 1990a. Modeling of live and true sharing in multi-cache memory systems. In *Proceedings of the 1990 International Conference on Parallel Processing (Vol. I Architecture)* (1990).

MISRA, J. 1986. Distributed-discrete event simulation. *ACM Computing Surveys 18*, 1 (March), 39–65.

NICOL, D. 1992. Conservative parallel simulation of priority class queueing networks. *IEEE Transactions on Parallel and Distributed Systems 3*, 3 (May), 398–412.

PRZYBYLSKI, S. A. 1994. *New DRAM Technologies: A Comprehensive Analysis of the New Architectures.* MicroDesign Resources.

REIDENBACH, E. 1993. *CHALLENGE Server Perdiodic Table.* Silicon Graphics Computer Systems.

REINHARDT, S. K., FALSAFI, B., AND WOOD, D. A. 1993. Kernel support for the wisconsin wind tunnel. In *Proceedings of the Usenix Symposium on Microkernels and Other Kernel Architectures* (Sept. 1993).

REINHARDT, S. K., HILL, M. D., LARUS, J. R., LEBECK, A. R., LEWIS, J. C., AND WOOD, D. A. 1993. The wisconsin wind tunnel: Virtual prototyping of parallel computers. In

Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems (May 1993), pp. 48–60.

ROESSLER, B. 1996. Personal communication.

ROSENBLUM, M., HERROD, S. A., WITCHEL, E., AND GUPTA, A. 1995. Fast and accurate multiprocessor simulation: The simos approach. *IEEE Parallel and Distributed Technology 3*, 4 (Fall).

Seminconductor Industry Association. 1994. The national technology roadmap for semiconductors. http://www.sematech.org/public/roadmap/.

SINGH, J. P., WEBER, W.-D., AND GUPTA, A. 1992. Splash: Stanford parallel applications for shared memory. *Computer Architecture News 20*, 1 (March), 5–44.

SPEC. 1990. Spec benchmark suite release 1.0.

STEINMAN, J. S. 1992. Speedes: A multiple-synchronization environment for parallel discrete-event simulation. *International Journal in Computer Simulation 2*, 251–286.

THIÉBAUT, D. AND STONE, H. S. 1987. Footprints in the cache. *ACM Transactions on Computer Systems 5*, 4 (November), 305–329.

UHLIG, R., NAGLE, D., MUDGE, T., AND SECHREST, S. 1994. 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)* (Oct. 1994), pp. 132–144.

WOOD, D. A., CHANDRA, S., FALSAFI, B., HILL, M. D., LARUS, J. R., LEBECK, A. R., LEWIS, J. C., MUKHERJEE, S. S., PALACHARLA, S., AND REINHARDT, S. K. 1993. Mechanisms for cooperative shared memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (May 1993), pp. 156–168. Also appeared in *CMG Transactions,* Spring 1994.

WOOD, D. A. AND HILL, M. D. 1995. Cost-effective parallel computing. *IEEE Computer 28*, 2 (Feb.), 69–72.