Accuracy vs. Performance in Parallel Simulation of Interconnection Networks

Douglas C. Burger and David A. Wood

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, Wisconsin 53706 USA
{dburger,david}@cs.wisc.edu

Abstract

Parallel simulation is emerging as the dominant technique for studying parallel computers. However, the interconnection networks of these machines can be modeled at many different levels of abstraction, allowing researchers to trade off accuracy and performance. In this paper, we use the Wisconsin Wind Tunnel, a parallel simulator for cache-coherent shared-memory machines, to study the trade-offs of accuracy versus performance for six different network simulation models. We evaluate these models for a variety of parallel applications, cache-coherence protocols, and topologies. We show that only the two most expensive models—which model contention at individual links—are robust in the presence of high network loads or non-uniform traffic patterns.

1 Introduction

Simulation has long been the dominant technique used to study the hardware, software, and hardware/software interactions of both current and proposed parallel machines [4, 5, 10]. Hardware designers use it to evaluate correctness and performance of new designs; software developers use it to develop codes for non-existent and non-accessible machines; computer architects use it to evaluate and refine the hardware/software interface.

This work is supported in part by NSF PYI Award CCR-9157366, NSF Grant MIP-9225097, and donations from Thinking Machines Corporation, Xerox Corporation, and Digital Equipment Corporation. Our Thinking Machines CM-5 was purchased, through NSF Institutional Infrastructure Grant No. CDA-9024618, with matching funding from the Univ. of Wisconsin Graduate School.

© ISSN 0-8186-7074-6. Copyright(c) 1995 IEEE. All rights reserved.

Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE. For information on obtaining permission, send a blank email message to info.pub.permissions@ieee.org.

By choosing to view this document, you agree to all provisions of the copyright law protecting it.

Parallel simulations of parallel computers have recently become widely used to accelerate these studies [6, 10, 18]. These systems exploit the inherent parallelism of the system under study (the *target*) by simulating it on an existing parallel machine (the *host*). They can be orders of magnitude faster than sequential simulators [11, 18], permitting researchers to evaluate system-level performance by running real applications on real data sets.

Unfortunately, all simulations must balance the need for simulation accuracy versus the desire for good simulation performance. Closer modeling of target system details by a simulator will produce more accurate results but will result in a slower simulation. This trade-off is particularly distinct when modeling the interconnection network of a parallel computer. At one extreme, a software engineer doing initial program development for a hypothetical machine cares only for a functionally correct simulation, not accurate performance estimates. At the other extreme, hardware designers implementing the interconnection network router require cycle-by-cycle simulation to understand the detailed interactions within their design.

Between these two extremes lies a wide range of studies where the trade-off between simulation speed and accuracy is less clear: application performance tuning, memory system design, cache-coherence protocol design, etc. If given a choice of network models, a researcher would presumably choose the fastest simulation model that gives "sufficient" accuracy for the given study. Unfortunately, without sufficient data on these trade-offs, researchers must either be conservative, and select a slower algorithm than necessary, or risk incurring unacceptable error.

To illustrate the potential pitfalls, consider our recent study of cache-coherence protocols using a parallel computer simulator [21]. The simulation assumed a simple point-to-point network with constant message delivery time (the C100 model described below). The results using this simulation indicate little performance difference between the simpler $\mathtt{dir}_1\mathtt{SW+}$ protocol and the more

Network Model	Appbt	Barnes	Ocean
C100	1.03	1.06	1.26
Baseline	1.10	1.12	1.63

Table 1. Relative protocol performance for two network simulations

This table compares the relative performance of two cache coherence protocols, $\mathtt{dir_1SW+}$ and $\mathtt{dir_NNB}$, for each of two network simulations, `C100` and `Baseline`. The values are the logical execution time of $\mathtt{dir_1SW+}$ divided by the run time of $\mathtt{dir_NNB}$.

complicated $\mathtt{dir_NNB}$ protocol, for most applications. As Table 1 shows, the more complex protocol is only 3-6% faster than $\mathtt{dir_1SW+}$ for two applications, and 26% faster for the poorly behaved Ocean application[1]. However, using a more accurate network model (the `Baseline` model described below) for a 2-dimensional mesh indicates a significantly greater difference. The $\mathtt{dir_NNB}$ protocol is actually 10-12% faster than $\mathtt{dir_1SW+}$ for Barnes and Appbt, and 63% faster for Ocean. While these results do not invalidate the conclusions of that paper, they suggest that the simpler protocol will not scale to larger systems with 2-dimensional meshes.

In this paper, we explore the simulation trade-off of *accuracy* versus *performance*, with the intent of understanding when simple network models are sufficient and when slower, more detailed models must be used. We use the Wisconsin Wind Tunnel (WWT)—a system for simulating cache-coherent, shared-memory multiprocessors using a parallel message machine [18]—to evaluate and compare six network models, for a variety of parallel applications, network topologies, and cache-coherence protocols. The six network models range from very fast to very accurate:

- **`C100`**: the original WWT model which assigns a constant 100 cycle delay to every message, independent of network topology, system size, and other network traffic.
- **`CMean`**: assigns a constant delay equal to the mean message time for a particular application, protocol, topology, and system size (as measured by a previous run of the `Baseline` simulator, below).
- **`Free`**: assigns a variable delay to account for network topology (i.e., number of hops) and message length, but ignores possible network contention.
- **`Random`**: adds a random delay, representing contention, to the variable delay of the `Free` model.
- **`Approximate`**: a distributed approximate simulator that accurately models channel utilization for each network node and uses past global information to estimate local contention and message ordering.
- **`Baseline`**: a detailed centralized event-driven simulation that accurately models network contention.

1. We do not use the `checkin` and `checkout` annotations [21], which significantly improve the performance of Ocean.

Our results quantify the intuitive trade-off between accuracy and performance. The “exact” `Baseline` simulation runs an average of 10 times slower than the original `C100` model. Conversely, the `C100` model has an average error (with respect to the `Baseline` model) of 12%, and over 20% error in several cases. The other models provide a continuum between these two extremes; for example, the `Random` model achieves mean error less than 5% while running an average of 2.5 times slower than `C100`.

However, our results also indicate that non-uniform traffic patterns (e.g., broadcasts), can introduce significant error if a model does not accurately account for contention. For example, despite its low mean error, we observed that `Random` has over 20% error for several combinations of application and cache coherence protocol that cause many broadcasts.

We show that the `Approximate` model—which estimates contention for each network link—is much more robust than these simpler models: 33 of 36 cases have error less than 5% and the remaining three—all of which exhibit heavy broadcast traffic—incur error less than 10%. The price for this increased accuracy is, of course, performance: the `Approximate` model runs five times slower than the `C100` model, on average. For the 32-node systems studied, `Approximate` is only a factor of 2 faster than the centralized `Baseline` simulator. However, because `Approximate` estimates the contention in parallel, it will scale to larger systems while `Baseline` will not.

The next section describes the parallel simulation environment, target system assumptions, and workloads. Section 3 qualitatively discusses the trade-off between simulation accuracy and performance, and describes the six simulation schemes in detail. Section 4 presents the simulation results and discusses the quantitative trade-offs. Finally, Section 5 summarizes the results.

2 Background

2.1 Parallel simulation environment

The *Wisconsin Wind Tunnel* (WWT) provides an abstraction of a user-specified parallel machine which directly executes application binaries [18]. A software layer implements this abstraction on a commercial hardware platform (a Thinking Machines CM-5). Given an

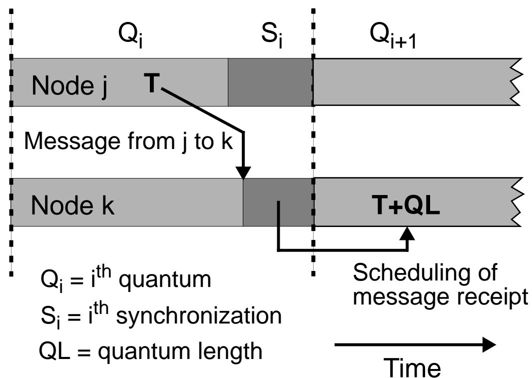

Figure 1. Quantum-based synchronization

Target messages sent at logical time T in quantum Q_i are scheduled as simulation events at logical time T + QL.

application, topology, protocol, and network model, WWT calculates the logical execution time (in cycles).

WWT achieves good simulation performance for two reasons. First, direct execution, in which a target instruction is "simulated" by executing the identical host instruction [7], allows most target instructions to execute at the speed of the underlying host hardware. WWT uses a fine-grain extension of shared virtual memory [17] to directly execute all load and store instructions (excluding instruction fetches), rather than just computation instructions.

Second, WWT exploits parallelism by simulating each target processor node on a separate host node. WWT uses a conservative synchronous discrete-event simulation algorithm [15, 16, 20] to insure causal event ordering between nodes [12]. This algorithm allows each node to directly execute instructions within a fixed-length quantum (also called a fixed-time window or time bucket [20]). At the end of a quantum, the host nodes must synchronize to ensure that all target messages have arrived, so that events that must be processed in the next quantum are properly ordered. This is illustrated in Figure 1: messages sent during quantum Q_i are guaranteed to arrive before the barrier synchronization S_i completes [14].

The performance of WWT is very sensitive to the quantum length, *QL*, the number of target system cycles simulated between barriers. As the quanta shrink, the overhead of initiating direct execution—essentially a context switch—dominates the simulation time. This is graphically illustrated in Figure 2, which shows that simulation performance decreases by a factor of 5–6 as the quantum length decreases from 100 cycles to 5 cycles. This slowdown is caused both by increased synchronization overhead and by an increase in load imbalance between the host nodes.

Clearly, we would like to simulate using large quanta; unfortunately, the maximum quantum length is limited by

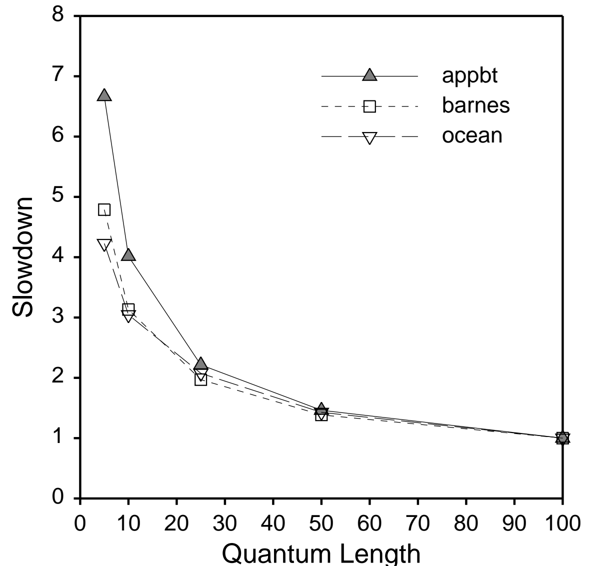

Figure 2. Effect of quantum length on simulation time

the speed and topology of the target network. To insure causality, the quantum length must be less than or equal to the minimum (logical) time that one target node may affect the simulation state of another.

The original version of WWT achieves a relatively large quantum (100 target cycles) by using a very simple topology-independent network model. This model ignores all network details and assigns messages a fixed end-to-end latency from any node to any other. However, because this simple model ignores network topology, contention, and other important factors, we have no guarantees that it accurately represents any real network.

2.2 Network assumptions

In the central focus of this paper, we compare a detailed "baseline" simulation against a set of increasingly approximate simulations. The baseline model takes into account not only system size and topology, but also details of the router implementation such as channel width, buffer sizes, and transmission delay.

The topology-dependent simulators used in this study are general enough to model *k*-ary *n*-cubes of arbitrary dimensionality—which include meshes, cubes, tori, and hypercubes—and fat-trees of arbitrary degree [14]. A fat-tree is a *v*-ary tree in which the aggregate bandwidth remains constant at each level; each internal (non-leaf) node has *v* children and a logical link to its parent with *v* times the bandwidth of each child link. In practice, multiple physical nodes (with constant bandwidth on each link) are grouped together to form logical internal nodes.

The implementation-dependent simulations assume network routers loosely based on the Torus routing chip

[8]. The Torus chip implements wormhole routing, where a message is broken into flow control digits, or *flits*, which are forwarded to an output channel as soon as it becomes available. A channel is *acquired* by the first flit in a message (the header) and is *relinquished* only after the last flit (the tail) passes through. Since a message can simultaneously occupy multiple channels on different routers, other messages may block waiting for a message to relinquish a channel.

In the k-ary n-cube topologies, messages use simple dimensional routing: each message is routed as far as necessary along each successive dimension, from the lowest dimension to the highest. In fat-tree topologies, messages are routed pseudo-randomly up the tree to the lowest common ancestor of the source and destination nodes, and then deterministically down the unique path to the destination. Deadlocks are prevented by providing a total ordering of physical channels, preventing cyclic blockage in the network [9].

Other differences with the Torus chip are: synchronous (rather than asynchronous) handshaking, unidirectional (rather than bidirectional) channels, and the network interface chip, which connects the CPU to the network. The network interface chip is modeled as a physical channel that experiences contention in the same manner as router channels. While these differences were introduced to simplify the simulation, the resulting target network represents a point in the design space which is both feasible and reasonable. We assume 8-bit flits, each channel buffers up to four flits, and can transmit one flit per cycle. There is a one cycle delay to resume the sending of a buffered message after it has been blocked. Message headers require 12 bytes.

2.3 Target system parameters

To evaluate the accuracy and performance of the various network simulators, we use the Wisconsin Wind Tunnel to model a cache-coherent shared-memory multiprocessor. All simulations assume that each processing node consists of a CPU, a 256-kilobyte 4-way set-associative data cache, a network interface, and a portion of the logically shared memory. A directory-based coherence protocol is used to maintain a sequentially consistent view of shared memory.

We vary three system parameters: network topology, cache coherence protocol, and workload. The interconnection network is one of a *k*-ary 2-cube, *k*-ary 3-cube, 2-ary fat tree, or 4-ary fat tree. The directory-based coherence protocols are:

- **dir_NNB** — an all-hardware protocol which maintains a directory pointer for each cache in the system (e.g., a bit vector). dir_NNB sends exactly the number of invalidation messages needed when a cache requests an exclusive copy [1].
- **dir_1SW** — a hybrid hardware/software protocol which maintains one hardware directory pointer which doubles as a counter for more than one sharer. dir_1SW traps to a software handler when it must send invalidation messages and sends broadcast invalidations when there is more than one sharer [13].
- **dir_1SW+** — an improved version of dir_1SW that optimizes the cases when there is only one copy of a cache block [21], eliminating a large fraction of the traps incurred by dir_1SW.

Benchmark	Shared misses	Cycles/ shared miss	Sharers/ line
Appbt	907551	3840	1.04
Barnes	396008	4704	6.48
Ocean	3612652	832	1.01

Table 2: Benchmark communication patterns

2.4 Benchmarks

The three benchmarks used for this study were Appbt, Barnes, and Ocean. Appbt is a locally parallelized version of a NAS Parallel Benchmark [3], and Barnes and Ocean are members of the SPLASH Benchmark suite [19]. We chose these three benchmarks because they exhibit very different communication patterns.

Appbt is a computational fluid dynamics program that solves multiple independent systems of non-diagonally dominant block tridiagonal equations. The block sizes are 5x5, since the solution requires 5 independent linear equations. The data in the cubic domain are partitioned into subcubes, each of which is assigned to a different processor. Communication occurs only on the boundaries of the domain subcubes. Two processors share points on the faces of the subcubes, four share points on the edges of the subcubes, and eight share points on the corners of the subcubes.

Barnes is a gravitational N-body simulation that uses the Barnes-Hut algorithm. The bodies are stored in an (8-ary) oct-tree, the nodes of which are partitioned dynamically across all available processors. Spatial proximity provides no guarantee of processor proximity. The sharing patterns are very irregular due to their dynamic nature.

Ocean is a hydrodynamic simulation that models a two-dimensional cross-section of a cuboidal ocean basin. The main data structures are two-dimensional arrays, which are divided up into columns and assigned in sequences to

individual processors. All sharing occurs between 2 processors which hold adjacent columns (boundary columns).

Table 2 shows communication statistics for the three applications: the number of misses to shared data, the frequency of misses to shared data (per processor), and the mean number of sharers when a cache line is upgraded from shared to exclusive state. To obtain these numbers, the applications were run with the $\texttt{dir}_1\texttt{SW+}$ protocol, the `C100` network model, and 32 target processors.

3 Network simulation models

Parallel computer simulators like the Wisconsin Wind Tunnel exploit direct execution and parallelism to both efficiently and accurately model the CPU and memory system. Ideally, we would like to apply these techniques to simulate interconnection networks with the same levels of accuracy and performance.

Direct execution improves simulation performance by using an underlying host operation to "simulate" a target machine operation. Thus, since WWT uses the CM-5's network to send target messages, it is tempting to claim it directly executes them. However, direct execution has two distinct steps: (i) simulate the functionality of the target operation and (ii) calculate the target execution time. For CPU instructions, WWT achieves the second by instrumenting the executable file to advance the logical clock by an appropriate number of ticks. This static analysis is sufficient to accurately model most current processors, even superscalar processors[1]. Unfortunately, the delivery time for a message depends upon what other messages are in the network, and hence cannot be calculated statically.

Efficient parallel simulation of interconnection networks is also difficult. Conservative discrete-event simulation algorithms achieve parallelism by exploiting lookahead, the minimum (logical) time before one node can affect the state of another. By never simulating past the current lookahead time, conservative algorithms ensure that simulation events are simulated in the correct order, and the simulation obtains "exact" results. Unfortunately, for the general class of wormhole-routed networks, this lookahead time is no more than one or two cycles (the hop-to-hop flit transmission time). As discussed in Section 2.1, shrinking the quantum size to this level obviates most of the advantages of directly executing CPU instructions.

An optimistic simulation algorithm, on the other hand, can simulate beyond the lookahead time, but must rollback the simulation state whenever it detects that simulation events are processed out of their logical time sequence (i.e., a causal violation). Optimistic algorithms offer superior performance when rollbacks are infrequent and the overhead of saving and restoring target state is sufficiently small. Unfortunately, preliminary analysis indicates that rollbacks would be frequent in network simulation: for some applications, over half of all messages see some contention, and hence might require rollback. Furthermore, rolling back a direct-execution simulation requires frequent, expensive checkpoint operations to save the processor state. Consequently, we do not consider optimistic algorithms in the remainder of this paper.

Instead of focusing on making exact simulation fast, this paper examines a set of approximate network models. A central tenet of this paper is that most users of parallel computer simulators do not require exact interconnection network simulation. They are either ambivalent about performance, or they care mostly about relative (rather than absolute) performance. For example, consider the development of a new cache coherence protocol. During the initial debugging stages, the designer does not care about accurate performance measurements. Even during evaluation, accurate relative performance (i.e., protocol A is 10% faster than protocol B) is much more important than accurate absolute performance (i.e., protocol A takes 9 seconds and protocol B takes 10). A fast simulator that provides accurate relative performance is much more valuable than a slow, exact simulator.

In this paper we examine the accuracy versus performance trade-offs for six different network simulators. There is no single "best" network model, but rather a range of alternatives, from which users can chose a particular accuracy/speed ratio. Of course, some models will be clearly superior to others for a given speed or accuracy, but no one model supersedes all others. We study three general categories of network models, listed in order of increasing accuracy and decreasing performance:

- Constant latencies
- Variable, analytically-generated latencies
- Detailed simulations

The remainder of this section describes these schemes in more detail.

3.1 Constant latency models

The simplest approximation, used in the original version of the Wisconsin Wind Tunnel, assigns a constant end-to-end latency to each message, independent of the distance between source and destination nodes. Since no computation is needed to determine the latency, this model incurs the minimum possible overhead. In addition, the lookahead (and hence the maximum quantum length) is equal to the constant network latency. Since the constant approximates the mean message latency, the quanta are generally much larger than for more accurate schemes,

1. As dynamic scheduling techniques—such as dynamic branch prediction—become more popular, static analysis may prove insufficient.

which have lookahead no greater than the *minimum* network latency.

While constant models obviously ignore network contention, they may still provide reasonable results if we select a good constant. The right constant depends upon the target's architectural parameters and the application's communication behavior. No single constant will provide accurate results for both a lightly-loaded network and a heavily-loaded network.

We consider two constant latency models: `C100` and `CMean`. `C100` is the original network model of WWT, where the constant is 100 target cycles, independent of application, topology, protocol, and system size. `CMean` uses "perfect knowledge" (obtained from the `Baseline` simulator, described below) to use the mean message latency for a given application, topology, protocol, and system size. This model is impractical because it performs a slower, more detailed simulation simply to obtain its parameter, however, it provides an indication of the best we can do with a constant model.

3.2 Variable (analytic) latency models

Variable latency models analytically compute latencies for each message, taking into account the distance from source to destination and possibly using local or past information to estimate contention. Variable latency models will generally be more accurate than constant models, because they explicitly account for network topology and correctly predict that messages to distant nodes take longer than those to neighboring nodes. These models will still not account for non-uniform access patterns, and the potential resulting contention.

The variable models will not run as fast as the constant models for two reasons. First, they must calculate the message latency for each message. Second, and more importantly, the quantum length must be less than or equal to the minimum message latency (15 cycles on our target systems). The increase in synchronization overhead and load imbalance will significantly degrade performance.

We examine two variable latency models in this study. `Free` assumes contention-*free* communication, where the message latency is the sum of the message length (in flits) plus the distance in network hops times the number of cycles to transmit a flit[1].

The `Random` model simply adds a random variable—to estimate contention—to the contention-free latency. Contention is modeled using a 2-stage hyperexponential distribution, given the squared coefficient of variation and the mean contention [2]. These parameters are obtained by performing a `Baseline` simulation for the given application, topology, and protocol. Thus `Random` incorporates "perfect knowledge" in the same way as `CMean`. More detail is provided in Appendix A.

3.3 Detailed simulation models

Detailed simulations, which model the utilization of individual channels in the interconnection network, are potentially much more accurate than the simpler constant- and variable-latency models. The simpler models do not account for the contention caused by non-uniform traffic patterns, e.g., hot-spots, localized communication, and distinct communication and computation phases.

However, because of the vastly greater bookkeeping and communication requirements of these models, they also incur much higher overheads. More importantly, because they model internal details of the network routers, they require a short quantum length to run in parallel. General wormhole routers, which may multiplex two or more virtual channels per physical channel, require a quantum length of one cycle. This worst case occurs because at any given cycle, a message may arrive for a higher priority virtual channel, preempting another message. This not only affects the delivery time of the preempted message, but also any messages that are waiting for channels held by the preempted message.

Simulating with single cycle quanta is prohibitively slow on the Wisconsin Wind Tunnel[2]. To provide greater lookahead—as described in Section 2.2—we do not multiplex virtual channels onto physical channels. This restriction allows the simulator to calculate the time at which a channel is relinquished as soon as there is sufficient buffer space between the channel and the header flit to hold the entire message. Since the message does not affect the destination's state until the last flit arrives, this property provides lookahead—at the destination node—proportional to the minimum message size.

We exploit this lookahead property in two detailed simulators. `Approximate` is a distributed simulation that accurately models the utilization of individual channels, but sacrifices exact message ordering for improved performance. `Baseline` uses a centralized simulator to accurately model all details of the network routers and contention.

3.3.1 A distributed approximation

Disallowing multiplexed virtual channels provides increased lookahead for the calculation of the message completion but it does not provide adequate lookahead to simulate network nodes in parallel. Early calculation of

1. We assume that messages must completely arrive at the destination node before they affect the state of the system.

2. WWT could be extended to support short quanta for network simulation and long quanta for processor simulations, but such an extensive modification is beyond the scope of this study.

message completion improves end-to-end lookahead, because a message does not affect a processing node's state until the tail flit arrives. However, network router nodes are affected by the header flit, and hence still have only single cycle lookahead.

We have developed a distributed approximate scheme (described more fully in Appendix B) that decouples the behavior of the network routers along a message's path. Each node estimates the local contention seen by the message, using *no information* from previous hops along the path. By eliminating the dependencies between intermediate hops, the lookahead increases to the minimum end-to-end message traversal time. This introduces error, as the header arrival time at each intermediate time becomes an estimate. The original message ordering along intermediate nodes is also lost. We expect that this model will perform accurately for networks that are not too heavily loaded. This model is also scalable to simulation of larger systems, as the network simulation messages and computation are completely distributed.

3.3.2 Baseline simulation

The `Approximate` scheme sacrifices exact contention calculations to simulate network nodes in parallel. The `Baseline` simulator, conversely, sacrifices parallelism to compute contention exactly. By centralizing the network simulation, `Baseline` can exploit the message length lookahead provided by the restriction on multiplexed channels in the target network. Target nodes send all messages to the centralized network simulator, which exactly models all network traffic and contention before forwarding the message and the correctly calculated arrival time to the destination node. As with all simulation, assumptions are made in `Baseline` that introduce discrepancies with the results that would be obtained from an actual network. However, we believe that the error resulting from the assumptions is negligible. Appendix C describes the simulation algorithm in more detail.

The fundamental disadvantage of `Baseline` is clearly the centralization of the network simulation. In addition to reducing overall simulation performance, this scheme becomes a more severe bottleneck as the simulated target system grows larger, making long simulations infeasible.

4 Results

In this section, we compare the accuracy and performance of the six simulation schemes for the applications, topologies, and protocols described in Section 2. As illustrated in Figure 3(a), our results indicate that there is a clear trade-off between accuracy and performance. This figure plots mean *slowdown*—a measure of performance—on the x-axis against mean percent error—a measure of accuracy—on the y-axis. The slowdown for simulation model *X* is defined as the CM-5 run-time for a simulation using model *X*, divided by the CM-5 run-time for a simulation using the Wisconsin Wind Tunnel's original `C100` model. Similarly, percent error compares the logical time for model *X* to the logical time for `Baseline`. The horizontal dotted line marks 5% relative error, an estimated level of accuracy that we believe is necessary for many system performance studies (e.g., cache coherence protocols).

Figure 3(a) illustrates that the six models provide a wide range of accuracy and performance. The Baseline simulation runs an average of 10 times slower than the original `C100` model. At the other extreme, the `C100` model has an average error of 12%. The other models represent additional points in the continuum, allowing researchers to balance their need for accuracy against their desire for performance.

Both `CMean` and `Random` have average accuracy at or below the 5% threshold and achieve reasonable performance. `CMean` incurs a slowdown of 1.5, while Random, which is somewhat more accurate, has a slowdown of 2.5. Unfortunately, to achieve this accuracy, both models require "perfect knowledge" of the mean latency or mean contention, respectively. In practice, additional error will be introduced when researchers estimate these values incorrectly. The `Free` model does not require any external parameters, but incurs an average error of 7%. Since it exactly computes the minimum delivery time for each message, the error is introduced by the failure to model contention.

The problem with these simple models is illustrated in Figure 3(b). Even though the mean error is low, several combinations of application, topology, and protocol incur significantly greater error: for example, even with perfect knowledge of the mean contention, `Random` underpredicts the logical time by over 20% in two cases. The factors that contribute to this error can be seen—by exclusion—in Figure 3(b and c). The figures show that none of the extreme error cases occur with the applications Barnes and Appbt or the $\mathtt{dir}_N\mathtt{NB}$ cache coherence protocol. Consequently, the large errors are caused by the interaction between the application Ocean and the broadcast-based protocols ($\mathtt{dir}_1\mathtt{SW}$ and $\mathtt{dir}_1\mathtt{SW+}$). The simpler models all have a fundamental problem with broadcasts, because they implicitly assume uniform traffic loads and broadcast is inherently non-uniform.

Figure 3(e) plots error against offered load, the average number of flit-hops (message length times number of network hops) each processor injects per cycle. The data clearly show that for low offered load, all simulation models attain good accuracy. This follows for the obvious reason that the lower the offered load, the less effect network

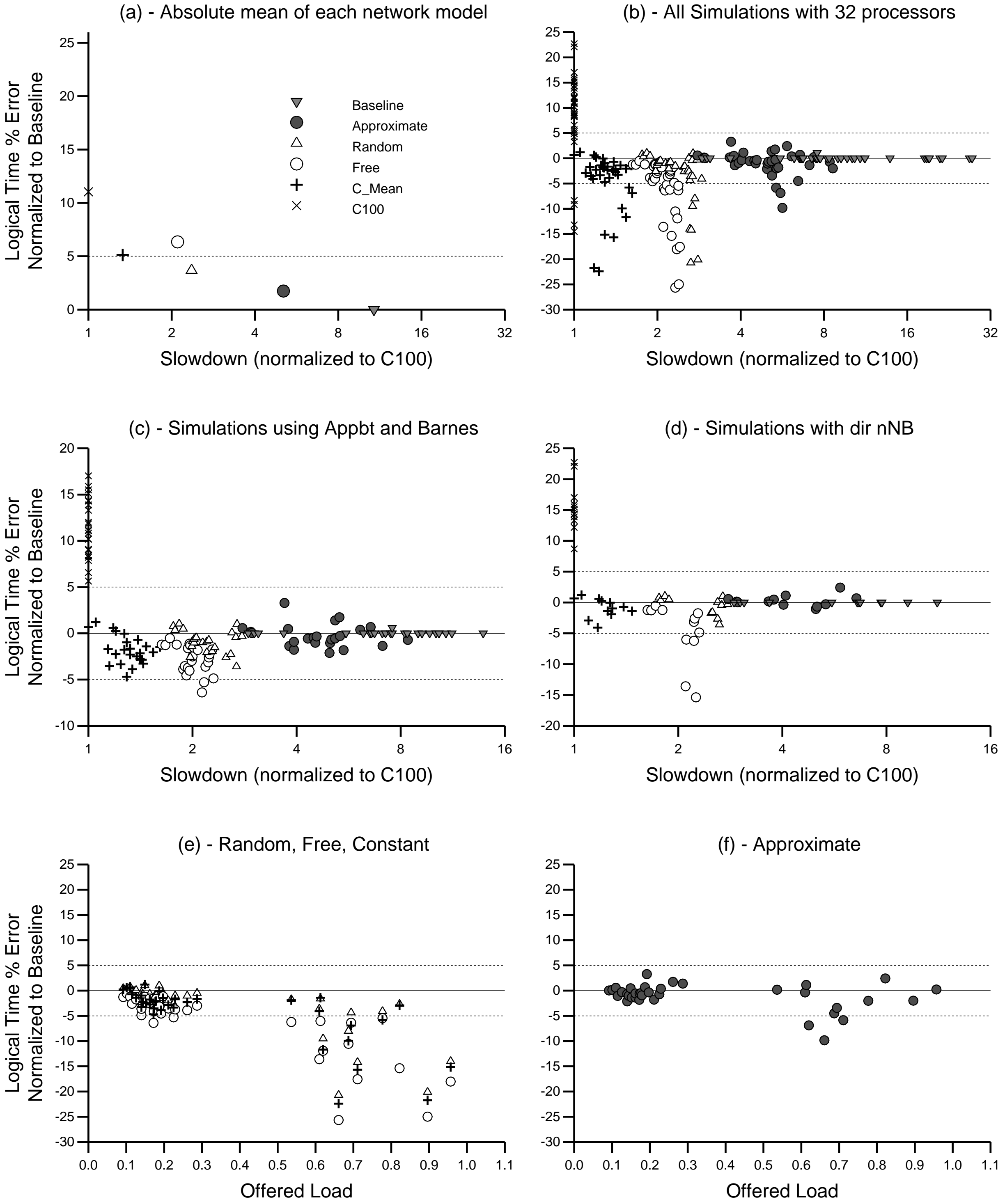

Figure 3. Comparison of error vs. slowdown and load

Graphs (a)-(d) show the error in logical time produced by different network simulation models. The errors are relative to the `Baseline` result, run with identical parameters. Each point corresponds to a combination of application, topology, network model, and cache-coherence protocol. Note that these graphs have differing scales on the y-axes. Graphs (e)-(f) compare the error of different network models as a function of offered load. The offered loads are calculated by multiplying the number of messages, the flits per message, and the distance each message travelled and dividing by the total logical run time and the number of target processors being simulated.

simulation error will have on the total logical time. Also, low offered load implies low contention, for which the simpler models are reasonably accurate. For high offered load—primarily caused by broadcasts—the error is much greater for the less robust topologies (i.e., a 2-D mesh and a 2-ary fat-tree).

Figure 3(f) shows that the `Approximate` model is much more robust than the simpler schemes. Because `Approximate` estimates the contention at each network link, it captures the congestion caused by excessive broadcast traffic. 33 of 36 cases have error less than 5% and the remaining three—all of which exhibit heavy broadcast traffic—incur error less than 10%. The price for this increased accuracy is, of course, performance: the `Approximate` model runs five times slower than the `C100` model, on average. Half of this slowdown is caused by decreasing the quantum length from 100 cycles to 15; the remainder is due to the extra host messages that must be sent to the intermediate nodes and the overhead of modeling the contention at each link.

At first glance, the `Approximate` model may seem little faster than the `Baseline` simulator: for the 32-node systems presented here, the difference is only a factor of two. However, the centralized `Baseline` simulator is fundamentally unscalable, while the `Approximate` model is scalable because it estimates the contention in parallel.

5 Conclusions

In this paper we use the Wisconsin Wind Tunnel to examine the accuracy and performance trade-offs of six different network simulation models. The models range from the original constant model of the Wisconsin Wind Tunnel, `C100`, to a centralized `Baseline` simulator that exactly models a restricted wormhole-routed design. The accuracy versus performance trade-off is striking: there is a factor of 10 performance difference between the fastest and slowest of the models. The fastest model—the original `C100` model—incurs an average error of 12%. The other models provide a continuum between these two extremes; for example, the `Random` model achieves a mean error of less than 5% while running an average of 2.5 times slower than `C100`.

However, only the `Approximate` model generates results with less than 10% maximum error for the 36 combinations of protocol, topology, and application considered in this study. Our results indicate that non-uniform traffic patterns, specifically broadcasts, can introduce significant error if a model does not accurately account for contention. For example, despite its low mean error, the `Random` model produces errors over 20% for several combinations of application and cache-coherence protocols that broadcast frequently.

In summary, these results indicate that simple models, while adequate for some simulation applications, are much too inaccurate for others. The inaccuracies of the simple models will only be exacerbated by studying larger target systems. More detailed models—such as the `Approximate` model described in this paper—are needed to accurately account for the contention caused by non-uniform communication patterns.

Acknowledgments

The authors thank Babak Falsafi and Steve Reinhardt for helping us to develop the network simulators, and the WWT group for providing a stimulating environment and critical feedback. We also thank David Chaiken, Babak Falsafi, and Jim Goodman for their insightful comments on earlier drafts of this paper.

References

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

[2] Arnold O. Allen. *Probability, Statistics, and Queueing Theory with Computer Science Applications*. Academic Press, Inc., 1990.

[3] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002 Revision 2, Ames Research Center, August 1991.

[4] Bob Boothe. Fast Accurate Simulation of Large Shared Memory Multiprocessors. Technical Report CSD 92/682, Department of Computer Science, University of California-Berkeley, January 1992.

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

[6] David L. Chaiken and Anant Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 314–324, April 1994.

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

[8] William J. Dally. *A VLSI Architecture for Concurrent Data Structures*. PhD thesis, California Institute of Technology, 1986.

[9] William J. Dally and Charles L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(10):547–553, May 1987.

[10] Helen Davis, Stephen R. Goldschmidt, and John Hennessy.

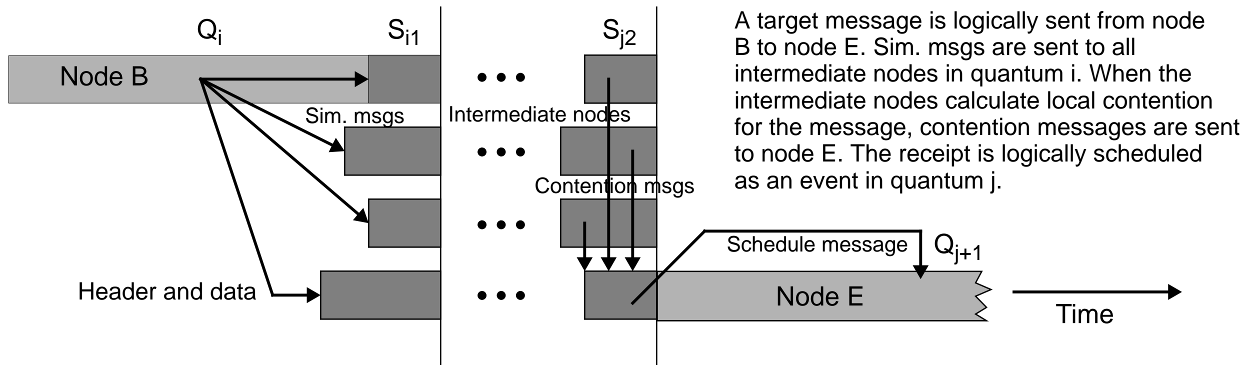

Figure 4. Synchronization for the distributed approximate model

Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages II99–107, August 1991.

[11] Babak Falsafi and David A. Wood. Cost/Performance of a Parallel Computer Simulator. In *Proceedings of PADS '94*, July 1994.

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

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

[14] Charles E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fifth Symposium on Parallel Algorithms and Architectures*, July 1992.

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

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

[17] Steven K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel Support for the Wisconsin Wind Tunnel. In *Proceedings of the Usenix Symposium on Microkernels and Other Kernel Architectures*, September 1993.

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

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

[20] Jeff S. Steinman. Breathing Time Warp. In *Proceedings of Parallel and Distributed Simulation*, pages 109–118, 1993.

[21] David A. Wood et al. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993. Also appeared in it CMG Transactions,/ Spring 1994.

A Variable latency model details

The variable latency model `Free` assumes contention-*free* communication. Thus the message latency (NL_{free}) is the sum of the message length (in flits) plus the distance (in hops) across the network[1]:

$$NL_{free} = T_f(N_k + L_k) \quad (1)$$

where T_f is the time to transmit a flit, N_k is the number of hops, and L_k is the number of flits in the message.

The `Random` model extends `Free` by adding a random variable X to account for contention:

$$NL_{random} = T_f(N_k + L_k) + X \quad (2)$$

The random variable X is computed as a 2-stage hyper-exponential random variable [2]. The squared coefficient of variation C_X^2 and the mean contention $E[X] = \mu^{-1}$ are input as simulator parameters. The appropriate values of $E[X]$ and C_X^2 are obtained by performing a `Baseline` simulation for the given application, topology, protocol, and system size.

B Implementation of a distributed approximation

`Approximate` removes the serialization of the message travelling through intermediate nodes. When a message is injected into the network, simulator messages are immediately sent to every intermediate node along the message's path. Each of these nodes estimates the local arrival time of the message, and the resulting local conten-

1. We assume that messages must completely arrive at the destination node before they affect the state of the system.

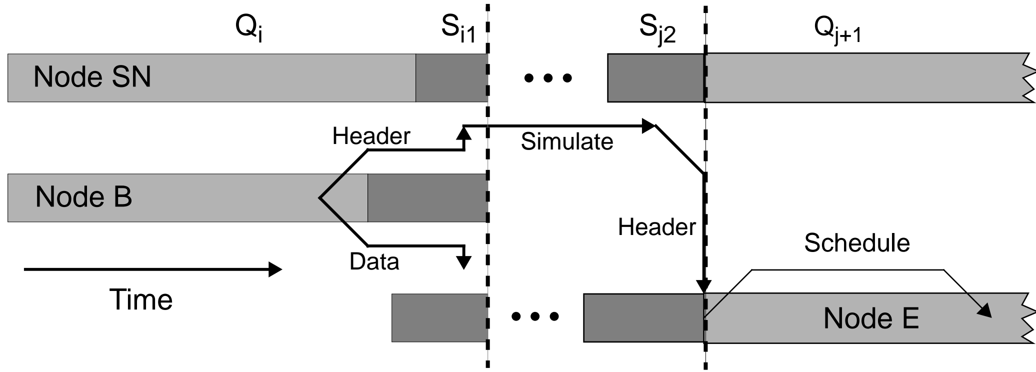

Figure 5. Synchronization for Baseline simulation

tion. The estimated local contention is then forwarded by each node to the destination. This scheme is illustrated in Figure 4. The key advantage of this scheme is that it allows the intermediate hops to simulate each message in parallel. The price of this parallelism is the loss of the exact time at which a message's header flit arrives at each intermediate network node. By relaxing the arrival times of messages, we also relax their arrival order and therefore the channel acquisition order. We estimate the arrival time at a hop i by adding the message's injection time to the product of the distance from the source and the mean global contention per hop in the network. This global estimate is obtained by computing the mean of all queueing delays seen by an arriving message at every channel in the network:

$$\mathrm{GML} = \frac{\sum_{i=1}^{NC} R^i_{T_G} - T_G}{NC} \qquad (3)$$

NC is the total number of channels in the target network, T_G is the current global logical time, and $R^i_{T_G}$ is the time at which the ith channel is released. The arrival time at a hop i is simply:

$$H_{k^i} = H_{k^o} + i\,(T_f + \mathrm{GML}) \qquad (4)$$

We calculate contention at an intermediate hop by scheduling a target message at its estimated arrival time. The contention seen at this hop—i.e., the time between the estimated arrival time and the eventual acquisition of the channel—is sent to the target message's destination. The destination node collects the local contention estimates from each hop along the target message's path, then schedules the arrival. The network latency is a simple function of the message length, distance travelled, and local contention estimates (c^i_k is the contention seen at hop i of message k):

$$\mathrm{NL}_k = T_f L_k + N_k + \sum_{i=1}^{N_k} c^i_k \qquad (5)$$

Unfortunately, if there is high global contention, but little or no local contention, this simple approximation can introduce a causal violation. The destination node may not receive the local contention estimate until *after* it should have scheduled the target message arrival. We solve this by simply scheduling the message as early as possible if it arrives late, effectively assuming that it encountered some extra contention during transit. This problem can arise frequently when applications exhibit distinct communication and computation phases, in which the network is subjected to bursts of heavy traffic immediately followed by periods of light traffic. In these cases `Approximate` will tend to overestimate the mean message latency.

C Implementation of baseline simulation

In `Baseline`, target nodes send all messages to the centralized network simulation node. As an optimization on the CM-5, only the message header packet is sent to the centralized node; all data packets (which contain pieces of cache blocks) are sent directly to the target destination nodes. The central node performs a detailed simulation of the network at the end of each quantum, and forwards message completions to their original destinations.

This centralized simulation structure requires an additional global synchronization at the end of each quantum[1].

1. `Approximate` also requires an extra synchronization per quantum to drain the network of local contention estimates.

The first synchronization ensures that all messages have been received by the central network simulation node; the second guarantees that messages sent by the central node have been received at their destinations. This structure is depicted in Figure 5.

The central `Baseline` simulation is event-driven and only needs to model (i) when the header of a message tries to acquire a channel, (ii) when the tail of a message relinquishes a channel, and (iii) when a blocked message is awakened, acquiring the channel for which it was queued.

The non-trivial part of this simulation involves the calculation of the time at which a channel may be relinquished. This is performed by calculating the buffer space between the channel and the header flit; when the space is sufficient to contain the entire message, the flits will pass through the channel in question regardless of whether or not the header flit is eventually blocked.

We first compute the hop number that the header flit must reach to enable channel i's release time to be deterministically calculated.

$$F^i_k = \min\left(i + \left\lceil \frac{L_k}{b} \right\rceil, N_k + 1\right) \quad (6)$$

Any channel in between F^i_k and i that had sufficient contention to delay the tail flit leaving node i must be taken into account. Nearer contention is subsumed by contention at nodes farther ahead, so only the farthest node which affects the tail flit must be factored into the channel's release time. The following equation calculates the farthest node which affects the tail:

$$G^i_k = \max\left\{ G^i_k \middle| \left(\sum_{j=i+1}^{G^i_k} c^j_k \right) \geq (B(G^i_k - i)) \wedge (i \leq G^i_k < F^i_k) \right\} \quad (7)$$

Equation 8 represents the actual time at which the channel in question is made available. That time is obtained by summing the time at which the header flit acquired the channel, the time spent transmitting flits through the channel, and the contention experienced by the tail flit (the last three terms in Equation 8).

$$T^i_k = H^i_k + L_k T_f + \left(\sum_{j=i}^{G^i_k} c^j_k \right) - B(G^i_k - i) + G^i_k - i \quad (8)$$