

A Comparison of Two Approaches to Parallel Simulation of Multiprocessors

Andrew Over, Bill Clarke and Peter Strazdins
Department of Computer Science
Australian National University
Canberra, ACT, Australia
Email: sim-devel@cnuma.anu.edu.au

Abstract—

The design trend towards CMPs has made the simulation of multiprocessor systems a necessity and has also made multiprocessor systems widely available. While a serial multiprocessor simulation necessarily imposes a linear slowdown, running such a simulation in parallel may help mitigate this effect.

In this paper we document our experiences with two different methods of parallelizing Sparc Sulima, a simulator of UltraSPARC III-based multiprocessor systems. In the first approach, a simple interconnect model within the simulator is parallelized non-deterministically using careful locking. In the second, a detailed interconnect model is parallelized while preserving determinism using parallel discrete event simulation (PDES) techniques. While both approaches demonstrate a threefold speedup using 4 threads on workloads from the NAS parallel benchmarks, speedup proved constrained by load-balancing between simulated processors. A theoretical model is developed to help understand why observed speedup is less than ideal.

An analysis of the related speed-accuracy tradeoff in the first approach with respect to the simulation time quantum is also given; the results show that, for both serial and parallel simulation, a quantum in the order of a few hundreds of cycles represents a ‘sweet-spot’, but parallel simulation is significantly more accurate for a given quantum size. As with the speedup analysis, these effects are workload dependent.

I. INTRODUCTION

Simulation frameworks have long been able to model multiprocessor systems, however until recently processor architecture designers were largely able to rely solely on uniprocessor performance models. The current trend towards multicore designs from IBM, AMD, Intel and Sun renders multiprocessor simulation a necessity for most architectural modelling.

Processors such as IBM’s Power5, Intel’s Core Duo and AMD’s Athlon64x2 already support two cores, with four core designs scheduled for release. Sun’s UltraSPARC T1 (Niagara) already supports 32 threads of execution across eight cores [1]. These multicore designs typically feature shared structures (such as level two caches) and it is essential to simulate the design in full to understand potential performance issues. Uniprocessor simulation already imposes a significant slowdown and traditional (serial) multiprocessor simulation exacerbates the situation by imposing an additional linear slowdown. Simulating a small benchmark on a uniprocessor may be time consuming, however, simulating the same benchmark on a 32-core system may prove prohibitive. Techniques such as using

fast functional simulation for warming or sampling may help reduce the overhead, but the basic linear slowdown remains.

A logical starting point for addressing the slowdown imposed by multiprocessor simulation is to examine the *parallel* simulation of multiprocessors. The main obstacle to effective parallel simulation is correctly ordering communication between processors. In traditional designs, this primarily concerns the cache coherence protocol. In an MP system, the cache coherence protocol and interconnect are the source of interactions, while many CMPs interact through a shared cache (multi-CMP designs must consider both interactions). While the cache is subject to state changes caused by external events, the majority of the core state remains private.

This paper examines two different methods of parallelizing the simulation of an UltraSPARC III-based multiprocessor system, along with their relative strengths and weaknesses. The first method ensures functional correctness, is easy to understand, but compromises determinism. The degree of variability the method introduces to the simulation results (in terms of execution time and hardware events observed) is also examined, the crucial factor being the simulation time quantum, which determines how closely the simulated threads are synchronized in terms of simulated time. The second method employs parallel discrete event simulation (PDES) techniques and preserves determinism by employing lookahead derived from the interconnect and coherence protocol. It also has greater potential in simulation accuracy. A simple theoretical model examining speedup limitations is presented along with an analysis of the workload dependency of the observed speedups. For the first method, speed-accuracy tradeoffs from varying the size of the simulation time quantum is also analyzed, and similarly shows strong workload dependency.

This work makes several contributions. A simple non-deterministic parallelization scheme is presented which differs from prior work by protecting against concurrent access in caches rather than in memory. The parallel implementation of the coherency protocol is non-trivial, and, to the authors’ knowledge, a description of it has not appeared previously in the literature. The extent to which round-robin timeslicing of multiprocessors in simulation influences accuracy is examined in conjunction with the non-determinism introduced by this parallelization technique. While this effect has been briefly noted, it has to the authors’ knowledge not been examined in

detail in the literature. A second mode of the multiprocessor simulator is parallelized using PDES techniques while preserving absolute determinism and obtaining a speedup. While detailed parallel CMP simulators exist, the authors are unaware of any simulator which has been parallelized in this manner on a shared-memory host.

The remainder of this paper is structured as follows. Section II presents some background material, examines previous work, and explains the initial design of the simulator, while Section III explores the implementation of the parallelized simulator. Results are presented in Section IV, and discussed along with the theoretical model in Section V. This is followed up in Section VI with an analysis of speed-accuracy tradeoffs. Conclusions are presented in Section VII.

II. BACKGROUND

A substantial body of research exists on parallel simulation, and the area has recently attracted renewed interest, owing in part to the reasons explained in the introduction.

A. Prior Work

Direct-execution simulators capable of modelling multiple processors have long supported execution in parallel. Typically each simulated processor is mapped to an independent host thread, with all host threads running in parallel aside from periodic synchronization. Tango Lite [2] is an assembly-language annotation framework which modifies compiled applications to invoke the simulator prior to each memory reference. Each simulated processor was modelled by its own thread of execution, with synchronization taking place upon simulator invocation. DirectRSIM [3] operated in a similar fashion, invoking a more detailed processor and memory timing simulator on each memory reference.

Embra [4] (a component of SimOS [5]) was a dynamic binary translation framework capable of full machine simulation. As part of the translation process, the translated code could be augmented to perform limited memory hierarchy simulation. Embra was capable of modelling multiple processors using one host thread per simulated processor and a shared memory region. Each thread would run in an unconstrained fashion until the simulated application reached the desired region. It is noted that this mode was highly non-deterministic (relying entirely on host scheduling), and unsuitable for performance evaluation. Its intent was to allow rapid positioning of workloads.

The Wisconsin Wind Tunnel [6] employed PDES techniques to model multiprocessor shared memory systems on a CM-5 distributed-memory host. The key technique employed was the division of simulated execution time into quanta, based on the assumption that it would take some minimum amount of time for any coherence event to become visible to a foreign processor. Provided no event arising within a given quanta can have any impact on another processor within the same quanta, all nodes may be run without communication within each quanta. At the end of a given quantum, coherence events may be exchanged before resuming simulation for the next quanta.

In this fashion, nodes runs in lockstep between communication windows. These techniques were later extended to work on SMP systems and clusters in the WWT-II [7].

Chidester and George [8] explored the development of CMP simulators capable of running on clusters, and extended SimpleScalar [9] to run in such a configuration. In this configuration each simulated processor was assigned to a host. All L1 traffic was handled locally, while L2 traffic was routed to an independent cache model which managed requests. Simulation was divided into quanta along similar lines to the WWT. This is an exhaustive work which examines numerous possible implementations, along with the discussed solution.

Manjikian [10] presented the parallelization of SimpleScalar by cloning all processor resources and spawning multiple threads. Multiple processors could be handled within each thread by interleaving, with all threads synchronizing on a barrier periodically. Cache coherency was not modelled.

A simulation time quantum of 1024 was chosen, but there was no discussion on why this value was chosen. However, it has been earlier noted that in serial multiprocessor simulation, where each CPU is simulated for a time quantum similarly but in a round-robin fashion, that there is a speed-accuracy tradeoff in the size of the quantum [5]. That is, a long quantum may give performance advantages, in terms of better cache utilisation on a host processor, but may give unrealistic interleavings of memory events, and therefore less accurate simulation [5]. It is evident that this tradeoff applies similarly to parallel simulation; however, to our knowledge, this tradeoff has not been systematically evaluated or analyzed, either for parallel or serial simulation.

SimpleScalar was also adapted to CMPs and parallelized by Donald and Martonosi [11] (along with IBM's Turandot). A method of using thread local storage to simplify modifying serial simulators to run in parallel is presented, along with two example implementations. This implementation differs from Manjikian's earlier work by synchronizing as necessary upon access to the L2 cache. No processor accessing the shared L2 may proceed until all simulated processors have reached the same timestamp, providing effective serialisation.

B. Simulator Design

Sulima is an execution-driven full-system simulator aimed at modelling an UltraSPARC IIIcu-based system [12]. Processor simulation is performed using a fetch/decode/execute loop which processes one instruction at a time. While the initial implementation assumed a single cycle latency for each instruction and fixed latencies for all memory transactions, the timing model has been augmented with a fast pipeline model [13], and a detailed memory system model.

The UltraSPARC IIIcu [14] is an in-order superscalar processor. While loads are blocking, memory latency may still be hidden through the use of software prefetch. The L1 caches consist of a 32 KiB 4-way pseudo-random I-cache and a 64 KiB 4-way write-through, pseudo-random D-cache. A 2 KiB prefetch cache (P-cache), which may be filled only through hardware prefetch or via prefetch instructions, is

employed simultaneously with the D-cache on floating point loads. The L2 (E-cache) is up to 8 MiB 2-way LRU with the tags residing on-chip with data residing in off-chip SRAM. Stores issue to an eight entry store buffer which coalesces them into a 2 KiB 4-way LRU write cache (W-cache). The W-cache is used to minimize bandwidth requirements between the processor and the off-chip E-cache data. Each line in the W-cache maintains per-byte valid bits, allowing the data to be merged with incoming data from the E-cache on loads. The D-cache is not included in the E-cache.

Coherency takes the form of a MOESI protocol implemented over a snooping domain of up to 24 processors, with a directory-based scheme operating between multiple snooping domains, as described by Charlesworth [15].

Over et al [16] describes the changes to Sulima required to support both a low-overhead memory model and a more detailed and accurate timing model. The two different timing models are named in accordance with their interconnect simulation methodology: the *passive* backplane assumes simple fixed latencies, while the *active* backplane incorporates a detailed simulation of the interconnect itself.

Multiprocessor support is handled via a simple round-robin scheme. The passive backplane simulates each processor for a user-specified simulation time quantum (timeslice), while the active backplane employs a fixed quantum for reasons described below. In this manner, all processors are kept in relatively close synchronization.

Sulima employs two different interconnect simulation models which are discussed further below (and in detail in prior work [12], [16]). The cache access process is identical for both implementations, with differences manifesting when the interconnect is contacted. The cache simulation is strictly procedural. Each level accesses lower level caches (and eventually the interconnect) via function calls.

Loads begin by examining the D-cache. On a hit, the replacement policy is updated and data is returned, while on a miss the request is forwarded to the E-cache. The E-cache will contact the interconnect if necessary to either obtain data or perform an upgrade, before updating the replacement policy and returning data. The miss is then allocated into the D-cache.

A store miss proceeds similarly, but the D-cache is updated only on a hit as there is no allocation on a store miss.

III. IMPLEMENTATION

The difficulty in performing parallel simulation effectively reduces to managing communication between simulated processors in a manner which firstly preserves correctness and secondly preserves simulation accuracy. In the context of shared-memory multiprocessors, this becomes an issue of managing cache coherency and memory consistency.

For this reason, the two different parallelization approaches discussed are closely tied to the memory model (and corresponding implementation of coherency). Each approach relies on behavioural assumptions and design goals rendering it specific to the given model. These approaches will be discussed in turn, with the relative advantages and disadvantages noted.

The two different approaches differ in their simulation of the interconnect. The first (passive) approach corresponds to a simple fixed latency model of an interconnect. The second (active) approach models the interconnect using event driven simulation. The event driven model is coupled with the procedural processor simulation polling to determine the final interconnect latency.

A. *Passive Backplane*

The passive backplane is a simple, fast, memory model corresponding to the original implementation of Sulima [12] which aims to achieve some degree of timing fidelity without excessive complexity. This memory model services requests immediately by returning the requested data (or performing a store) and indicating the appropriate latency to stall the processor model. While a trivial store buffer is implemented (which simply acts as a means of calculating store latency), stores are performed upon request. Prefetch is implemented, but is viewed as succeeding immediately; there is no support for performing asynchronous memory operations in this model.

Coherency is managed by allowing the interconnect to directly inspect the cache of every processor in the system. When it is determined that a miss has occurred or an upgrade is required in the E-cache, the backplane is notified and the coherency protocol is invoked.

The backplane performs two loops over every processor in the system. The first instance queries every cache directly to determine whether the line in question is present and its state. The responses are merged to obtain an overall snoop response, before a second pass is performed, informing each cache of the result of the request, and allowing them to take the necessary action (such as invalidation). The second pass may also copy data out of a foreign cache directly if a cache-to-cache transfer is required.

As the D-cache is not included in the E-cache, a D-cache invalidation may arise from snooping, even if the line in question is not present in the E-cache.

While this arrangement is satisfactory for serial simulation, when running in parallel, there are several immediate concerns:

- A D-cache hit may update data used by the replacement policy.
- A foreign snoop may invalidate a D-cache line and update data used by the replacement policy.
- An E-cache hit may update data used by the replacement policy.
- A foreign E-cache request may downgrade E-cache state and copy data.
- A foreign E-cache request may invalidate a line.

These concerns reduce to ensuring that no two simulated processors may simultaneously access E-cache lines with the same cache index, even if on separate simulated processors. The E-cache access process checks cache tags and state prior to accessing data, providing a window in which cache state may be inconsistent. Consider two racing simulated writes to the same memory line: the first processor could upgrade its

cacheline state before being preempted by a second processor which obtains exclusive access, performs a cache-to-cache transfer and completes a write to the same cacheline. When the first simulated processor resumes, it will not realise that its line state has been downgraded, and its write will be lost. This is one example of a potential coherency problem arising because tag checks and data access are not atomic within the simulator. These errors are likely to crash the simulated application (due to corrupt memory), or crash the simulator itself (due to corrupt state).

Index-based locking provides the necessary degree of protection. By allowing only one simulated processor at a time to access a given index within the cache, state is protected against conflicting, concurrent updates. Providing index-based locking also allows non-conflicting accesses to proceed in parallel, while protecting against potential corruption. Previous approaches have locked memory to ensure atomic updates to memory, however, Sulima models data in caches, making this approach unsuitable.

If locking is required for the D-cache on a hit, care must be taken to ensure the same lock is chosen for a given address in both the D-cache and the E-cache. Lock selection is based upon the intersection of the bits determining the cache index in the D-cache and the bits determining index in the E-cache. Only physical bits may be used for locking, as the address used for indexing must be consistent across caches (the D-cache is virtually indexed). In our implementation this allows 128 different locks to be used.

To perform parallel simulation, multiple host threads are spawned, and each is assigned one or more simulated processors. Within each thread, processors are simulated in a round-robin fashion using an “inner timeslice”. The host threads synchronize on a barrier upon the completion of each “outer timeslice” (simulation time quantum). This ensures that processors remain approximately synchronized, and is an analogous approach to earlier work (see e. g. Manjikian [10]). A prototype of the authors’ design was implemented by Thorn [17].

While this approach exposes some parallelism within the simulation, employing host-based locking compromises determinism. Event ordering between transactions touching the same cache line is determined by the order in which the host threads claim the appropriate lock (and thus to some extent on host scheduler behaviour). The magnitude of this effect may be reduced by shrinking the outer timeslice, however, even an outer timeslice of one will not ensure determinism.

Although the non-determinism introduced may have been expected to complicate simulator maintenance and debugging, in practice the relative simplicity of the locking scheme was easy to debug in conjunction with stress testing. Failure modes were not always reproducible, but were usually simple to find.

B. Active Backplane

The active backplane is intended to provide a detailed model of cache and interconnect behaviour. The system store buffer, write cache and prefetch cache are fully modelled,

and asynchronous memory transactions (arising from both the store buffer and prefetch instructions) are supported. The interconnect address bus is modelled including contention and arbitration, however, the system data bus does not model contention (merely fixed latencies). Mild NUMA effects (comparable to those observed on a V1280) are modelled. The design, implementation and validation of the timing model are documented by Over et. al. [16].

The most notable difference between the active and passive backplane is that the active is implemented as an event-driven timing model. This is interfaced with the processor model by using polling to determine when a request has completed.

The interconnect model is substantially more complex and operates in a fundamentally different manner to the passive backplane. Rather than allowing a cache miss to force direct interaction with foreign caches, all coherency interactions are handled using queued events. Once a request is received, snoop events are queued on all processors. Once responses are received by the interconnect, the results are merged and all processors are notified. The coherency protocol runs in parallel on all caches, and the combination of the original transaction and the snoop response is sufficient to allow each cache to take the correct action.

A key observation in this design is that in accordance with the interconnect protocol [15], a request takes a number of bus cycles to become visible to foreign processors. This latency provides sufficient lookahead to introduce parallel discrete event simulation (PDES) techniques (Fujimoto [18] provides a good survey). A conservative, synchronous technique analogous to that used in the Wisconsin Wind Tunnels [6], [7] is employed. Based on the observed “window” between an event being initiated on one processor and having any observable impact on another, we observe that provided a processor has a list of all events occurring within its window, it may safely simulate until the end of its window without any external communication. Rather than dynamically adapting the window size, a fixed window is chosen based upon properties of the coherency protocol. At the end of each simulation quantum, the interconnect simulator is invoked to determine and enqueue all events for the following timeslice. Execution alternates between parallel simulation of processors and serial simulation of the interconnect. This is similar to the `Baseline` simulator employed on the WWT by Burger and Wood [19].

Although many similar techniques from the WWT are employed, parallelization is performed by taking advantage of the timing of the coherency protocol, allowing a parallel simulation of a system containing a detailed interconnect model. A more detailed processor model is employed along with simulation of a more closely-coupled snooping (rather than directory based) protocol.

The races described in Section III-A cannot arise in the active backplane owing to the serialisation imposed by the interconnect, and the use of event driven techniques. The interconnect model determines a global event ordering on all new requests, which in turn determines the cycles on which cache coherency interactions occur, guaranteeing that only one

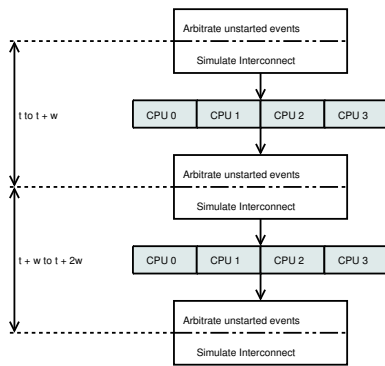


Fig. 1. Active Backplane Runtime Behaviour

snoop may occur per cycle. Furthermore a cache may only be modified by the owning processor model, either via a request for data or an enqueued snoop event. This guarantees that modification occurs only within a single thread, and neatly avoids concurrency issues within the simulator.

With this approach, the processors may be simulated in parallel before invoking the interconnect model to run (serially). Simulation therefore alternates between a parallel simulation phase, and a serial phase in which events are scheduled and enqueued for the following cycle as illustrated in Figure 1. As event ordering is determined within a serial region of the simulation, it remains deterministic¹.

Although deterministic, debugging this implementation proved much more complex than the non-deterministic approach owing largely to its event driven nature. In spite of the clean separation of simulated processors, a number of races were discovered in the shared data structures of the interconnect, and in the implementation of the coherence protocol. Finding these problems proved difficult as the symptoms and the underlying fault were often widely separated, while attempts to log events often serialised the simulation and hid the problem.

IV. RESULTS

The scalability of both parallelization approaches was examined using a variety of benchmarks. This approach aimed to expose underlying limits of the approaches in addition to revealing insight into workload dependencies.

The stability of the initial approach subject to varying outer timeslices was also examined, with results compared against those of a serial simulation.

A. Methodology

The majority of testing was performed on a Sun V1280 containing 12×900 MHz UltraSPARC IIIc processors, 24 GiB of RAM and running Solaris 10. Testing was confined to 8 of the 12 processors comprising two of the three processor boards using processor sets. While the system is mildly NUMA, this

¹A host lock is used to serialise access to the system call emulation facility of the simulator. This does introduce some trivial non-determinism, however, this remains confined to benchmark setup and teardown.

is not viewed as a significant effect. Processor assignment was not controlled manually (beyond the processor set).

A combination of synthetic and application derived benchmarks were used for simulation performance evaluation. In all cases, these benchmarks were much smaller than would be considered appropriate for architectural evaluation. This is not viewed as a serious problem as the benchmarks act as a workload for evaluation of the simulator, not as an evaluation of the simulated target.

A synthetic benchmark *sharetest* was employed to simulate high cache contention. This benchmark is effectively composed of walking a series of addresses contained within an array. 70% of the references are reads to a private area, while the remaining 30% are writes to a small shared area common to all processors. The intent of the benchmark is to force a substantial degree of conflict between processors in the memory subsystem.

A second synthetic benchmark *spintest* was employed. This benchmark performs no memory references and simply performs addition within a loop. The intent of this benchmark is to examine scalability in the absence of memory traffic.

The OpenMP implementation [20] of the NAS Parallel Benchmarks [21] was employed to examine scalability under more realistic workloads. These benchmarks are composed of simplified kernels taken from common scientific workloads (primarily computational fluid dynamics).

The synthetic benchmarks were compiled using `gcc` version 3.4.4, while the NAS benchmarks were compiled using Sun Studio 10. All benchmarks were compiled to 64-bit and were optimized for the UltraSPARC IIIc.

Owing to changes in the underlying workload as the processor count is increased, scalability was determined by comparing the run time of a parallel simulation of the workload with the timing of the same workload configuration under the serial simulator. Allowing $T_{ser}(p)$ to denote the serial simulation time for p processors, and $T_{par}(p, t)$ to denote the parallel simulation time for p processors employing t host threads, speedup is calculated as shown in Equation 1.

$$S(p, t) = \frac{T_{ser}(p)}{T_{par}(p, t)} \quad (1)$$

Owing to setup and verification overhead, only the parallel region of each benchmark was timed for the purposes of calculating speedup. To allow comparison of speedups with varying numbers of simulated processors assigned to each thread, t is fixed at $\frac{p}{n}$ for $n \in \{1, 2, 4, 8\}$ in the following figures (i.e. either 1, 2, 4 or 8 simulated processors were assigned to each thread).

B. Passive Backplane

Consider first the scalability results of the passive backplane. As noted earlier, this approach is somewhat analogous to both Embra [4] and Manjikian's earlier approach [10]. Multiple threads are run in parallel and pause periodically to synchronize clocks (the default synchronization period (simulation quanta) is 256 clock cycles). Access to caches is

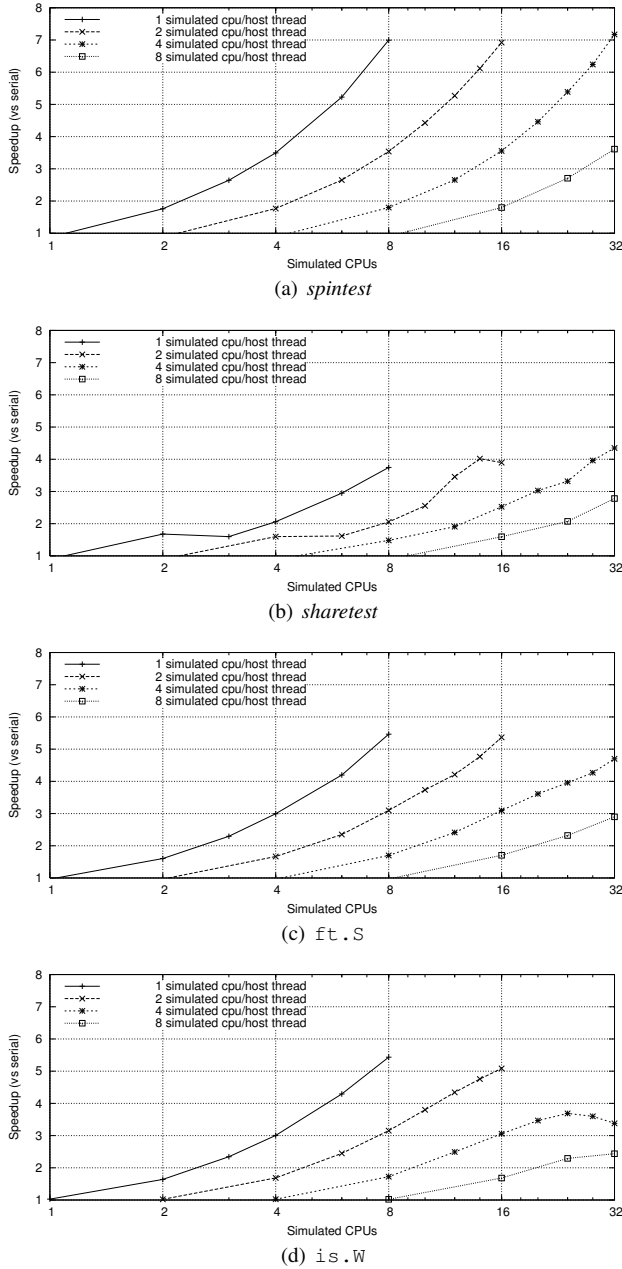


Fig. 2. Scalability results using passive backplane. For a given number of simulated processors the workload is fixed.

protected by an array of locks, but is otherwise unconstrained. This locking provides correctness but no guarantee of determinism.

Figure 2(a) illustrates the scalability of the *spintest* benchmark; this involves no memory references, and so demonstrates scalability subject only to the synchronization overhead. Without any lock contention, this parallelization approach demonstrates near-linear scalability, and thus significant potential.

Under heavy load the scalability deteriorates, yet still demonstrates some potential. Figure 2(b) illustrates scalability using the *sharetest* benchmark, with all threads subject

to a heavy stream of conflicting writes. The curves illustrated in this figure are noticeably distorted in comparison to Figure 2(a). This is due in part to the construction of the benchmark; although each spawned thread has a fixed reference stream, the presence of additional reference streams changes the timing and interaction of other threads. Due to the manner in which locking is used to coordinate cache access, this may have a significant effect on overall simulation speed. These interactions may also contribute to load imbalances as discussed below.

To ensure that the results demonstrated by synthetic benchmarks were reproducible on realistic workloads, the same analysis is also performed using the NAS Parallel Benchmarks. The NAS benchmarks were only examined up to 8 simulated processors owing to concerns over scalability of the benchmarks at the smaller problem sizes. Figure 2(c) illustrates the scalability of class S of the FT benchmark, while Figure 2(d) shows class W of the IS benchmark. In each case, scaling to 8 threads provides a $5.5\times$ speedup over serial simulation using the same configuration, while employing 4 threads provides a $3\times$ speedup.

One curious artifact of the *is.W* timing is that the speedup begins to deteriorate for 24 or more simulated processors with four processors per host thread. The cause of this deterioration remains unclear, however, the behaviour of scaling such a small benchmark to a large number of threads is believed to be responsible.

C. Active Backplane

In contrast to the passive backplane, the active backplane provides determinism. No locking is used in this implementation (with the exception of a few corner cases related to communicating data to the interconnect).

Figure 3(a) demonstrates the theoretical peak scalability of the simulator when subject to no memory traffic, once again using the *spintest* benchmark. This result is near identical to the earlier results for the passive backplane (Figure 2(a)),

In comparison, the results for *sharetest* (Figure 3(a)) are quite disappointing, with a peak speedup of just over 3 when using 8 threads. This benchmark generates substantial memory traffic, and our initial suspicion was that the high volume of memory traffic was increasing the simulation overhead of the interconnect, which acts as a serial component of the simulation. Instrumentation of the simulator revealed that although this benchmark did strain the interconnect, simulation of memory transactions was consuming less than 5% of runtime. Instead, the poor scalability stems from a severe load-balancing problem, which is discussed further below.

The *sharetest* benchmark proves to be something of a worst case scenario, as results of the NAS benchmarks illustrate. Figure 3(c) shows results from the S class of FT, while Figure 3(d) illustrates the W class of IS. Speedups in excess of 3 and 4.5 are achieved using 4 and 8 threads respectively on IS, while a more anaemic result of 2.5 and 3.5 is demonstrated on FT.

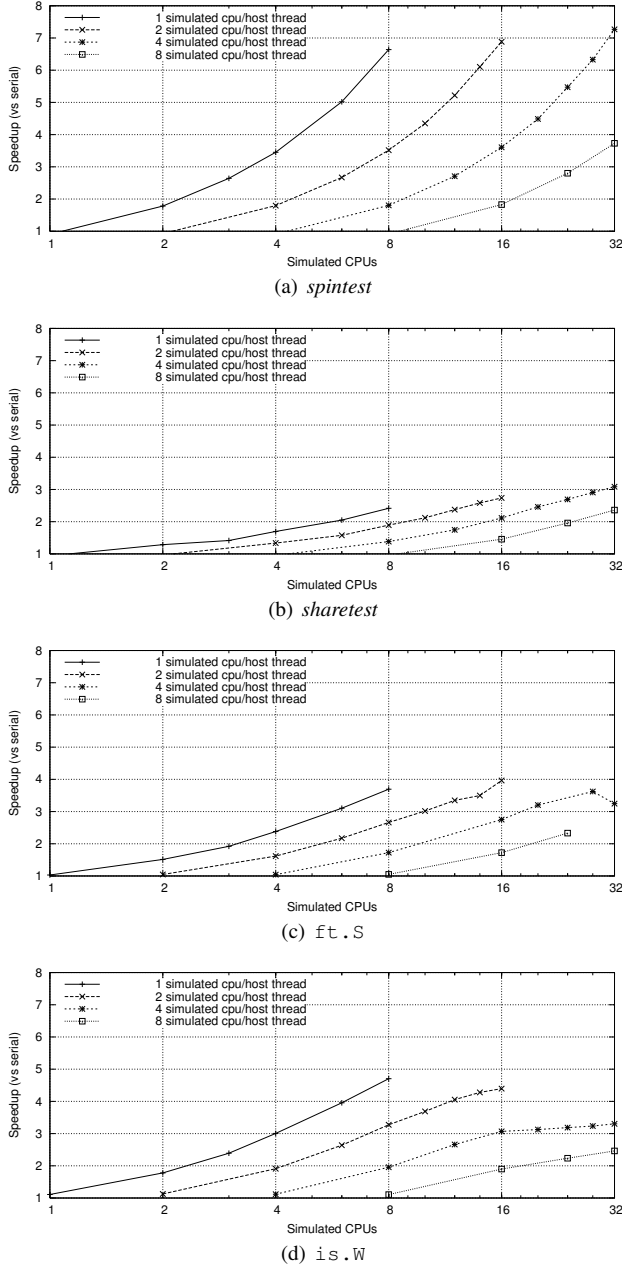


Fig. 3. Scalability results using active backplane. For a given number of simulated processors the workload is fixed.

V. ANALYSIS

While our results illustrate that achieving a substantial speedup is possible by parallelizing the simulator, the speedups obtained are disappointing, particularly in light of the results for *sharetst*. Although the use of frequent barrier synchronization (particularly in the active backplane) was initially suspected as an underlying scalability problem, the results on the *spintest* benchmark vindicate the parallelization.

The results make apparent that simulation speedup is clearly dependent upon the workload being simulated; some workloads scale almost linearly, while others barely scale at all.

A. Load Balancing

A close comparison of the first and second points for each plot in Figure 3(b) reveals that scalability for a given number of host threads is better if additional simulated processors are assigned to each thread. Scaling a 32 processor simulation to two threads is likely to yield a better speedup than scaling a two processor simulation to two threads.

This behaviour has previously been noted in the literature. Mukherjee et. al. [7] observed that the deviation in the time required to simulate a quantum decreased as the number of assigned processors increased. This improvement stems from what is effectively averaging of simulation time across multiple threads. If all threads take a similar amount of time to simulate their quantum, there will be little time wasted spinning on a barrier. The thread slowest to simulate is effectively the critical path; if its run time deviates substantially from the average, then speedup is constrained.

This effect is particularly pronounced given the processor simulation model used in Sulima. As the UltraSPARC III does not support non-blocking loads, on a load miss the processor will stall. The active backplane employs relatively small quanta, and an E-cache miss can render the processor model stalled (with the exception of snoop processing) for several successive quanta. This variation may be less pronounced with a longer simulation quantum, owing to a larger numbers of instructions being processed per quantum.

While this issue exists when running homogeneous workloads, a heterogeneous workload may prove significantly worse. In Sulima's processor model, any instruction stream demonstrating a high IPC (such as optimized floating-point code) will take longer to simulate than code with poor IPC (such as complex conditional logic).

Figure 4 shows histograms of the time taken to simulate a processor for one quantum using the active backplane. The simulations were all four processor benchmarks run on one, two or four host threads, with the quantum simulation time scaled for the scenarios with multiple simulated processors per thread (i.e. the one and two host thread simulations). The histograms have been normalised by dividing the individual counts by the total number of samples so that different results may be easily compared. Note that the y-axis is log scale, and the x-axis differs between graphs.

These figures give some indication of the extent to which simulation time varies between quanta, and also between benchmarks. The *spintest* (Figure 4(a)) benchmark shows almost entirely homogeneous behaviour, while the remaining three benchmarks show more varied distributions. The following analysis will demonstrate that the simulation of a benchmark can scale is closely tied to these distributions.

B. Theoretical Model

In order to quantify the overall impact of this variability in quantum simulation time, a theoretical model is developed. Allow the time taken to simulate a processor for one quantum to be represented by T_p , while the time taken to simulate the backplane is denoted by T_b , where each is considered

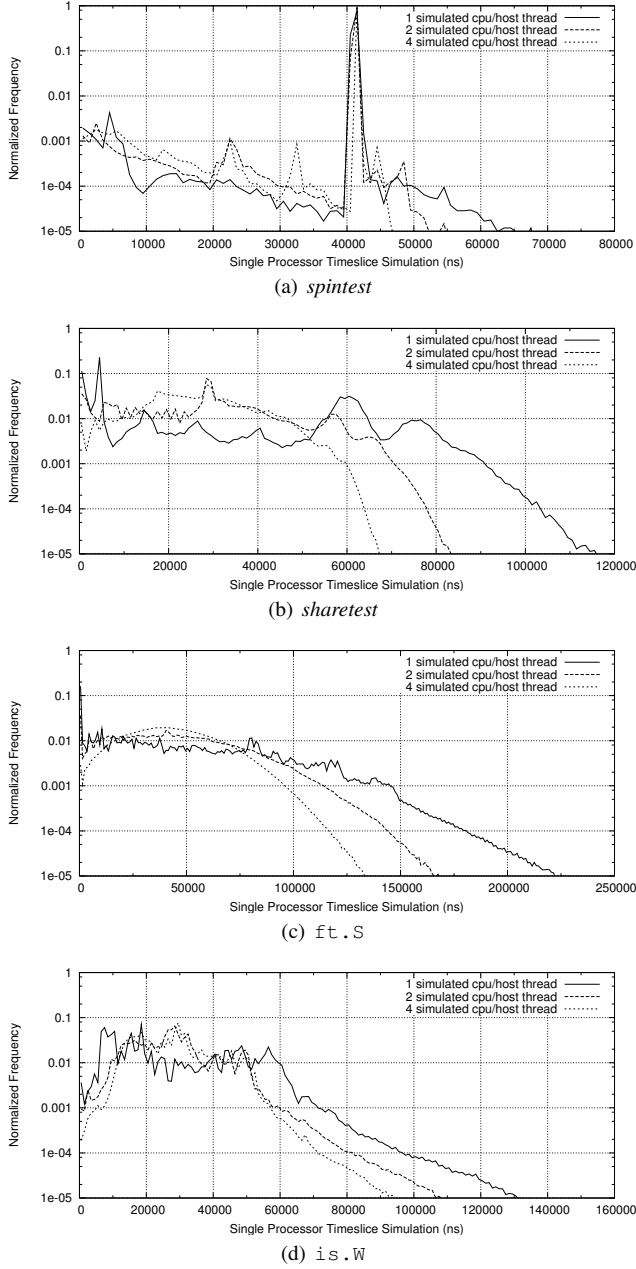


Fig. 4. Variability of quantum simulation time using the active backplane. Note the y-axis is log scale, and the x-axes have different scales.

a random variable. If T_{Np} represents the time required to simulate N processors for one quantum on N threads in parallel (assuming one simulated processor per thread), then T_{Np} may be calculated as shown in Equation 2.

$$T_{Np} = \max_{i=1}^N(T_{p_i}) \quad (2)$$

Let the pdf and cdf of T_p be denoted by $f(t)$ and $F(t)$ respectively, and the pdf and cdf of T_{Np} be $f_N(t)$ and $F_N(t)$. If it is assumed that multiple samples of T_p are independent and identically distributed, then using Equation 2 it is possible to derive an expression for $F_N(t)$ as follows.

$$\begin{aligned} F_N(t) &= P(T_{Np} \leq t) = P\left(\bigcap_{i=1}^N (T_{p_i} \leq t)\right) \\ &= \prod_{i=1}^N P(T_{p_i} \leq t) \quad (\text{independent}) \\ &= P(T_p \leq t)^N \quad (\text{identically distributed}) \\ &= F(t)^N \end{aligned} \quad (3)$$

Equation 3 yields an expression of the cdf of T_{Np} in terms of the cdf of T_p . It is readily apparent that as more threads are added, the distribution of T_{Np} will skew towards the upper end of the distribution of T_p . This equation provides sufficient information to calculate the expected value of T_{Np} . Observing that simulating N processors serially for one quantum will require N samples of T_p while simulating N processors in parallel requires a single sample of T_{Np} (see Figure 1), and applying the law of large numbers, we reach the following expression for the speedup for simulation using N parallel threads:

$$S_N = \frac{NE[T_p] + E[T_b]}{E[T_{Np}] + E[T_b]} \quad (4)$$

Equation 3 and Equation 4 may be numerically evaluated for a variety of scenarios by choosing an appropriate distribution for T_p . Based upon experimental data, we consider three different distributions with an expected value around $75 \mu\text{s}$, and an expected value of T_b of $5 \mu\text{s}$. These values were chosen empirically based upon the S class of FT.

The first distribution is continuous uniform distribution between $5 \mu\text{s}$ and $145 \mu\text{s}$. The second distribution is normal with mean of $75 \mu\text{s}$ and standard deviation of $5 \mu\text{s}$. The third distribution is the normalised weighted sum of two normal distributions; the first as described previously, the second with mean of $250 \mu\text{s}$, standard deviation of $20 \mu\text{s}$ and $\frac{1}{10}$ the weight of the first. This final distribution is intended to model infrequent large latencies.

Figure 5(a) demonstrates the expected value for T_{Np} as the number of host threads increases, while Figure 5(b) demonstrates the corresponding projected speedup. What is immediately apparent is that degree to which speedup is constrained by the variation in the quantum simulation time amongst processors. The best scalability is demonstrated by the single normal distribution, which remains tightly clustered about its mean. The expected simulation time for a distribution based on the dual gaussians very quickly deteriorates to almost triple the initial expected value, providing a corresponding restriction on efficiency.

Inadequate load balancing very quickly dominates scalability concerns within this timing model. Any simulation which demonstrates considerably different IPC between different threads will be subject to this effect. This analysis clearly demonstrates why scalability improves once multiple processors are simulated within a single host thread; this serves to moderate some of the variance by averaging simulation

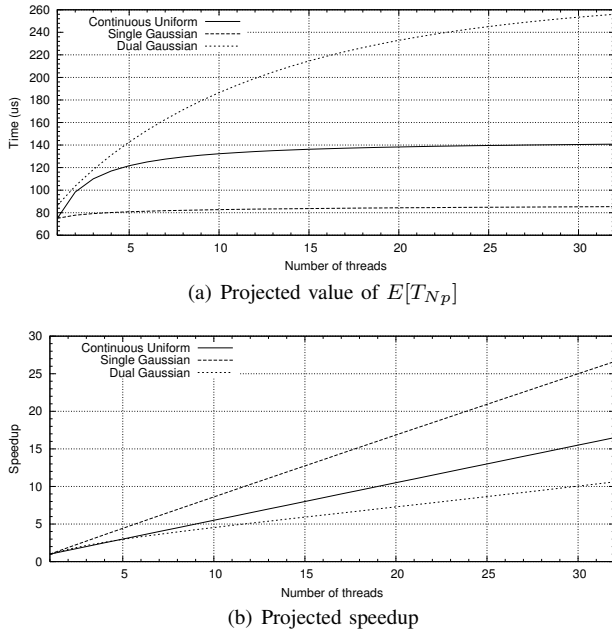


Fig. 5. Evaluation of theoretical speedup

time across multiple processors. In Sulima this problem is exacerbated by the simple processor model which performs no work in simulation quanta for which it is stalled on a load miss. A more detailed out-of-order model would likely demonstrate less severe variation owing to the absence of this stall behaviour.

Considering Figure 4, it is apparent that *spintest* scales well owing to the very narrow distribution evident in Figure 4(a), with the single peak three orders of magnitude above the remainder of the graph. On the other hand, *sharetest* shows two minor peaks, and a relatively flat distribution with a long tail. As the number of processors in the simulation is increased, this distribution will rapidly skew the maximum value upwards and severely constrain scalability. The effect of adding additional simulated processors to each thread is also readily evident in the smoothing and shifting to the left of the histograms shown in Figure 4.

That load balancing acts as a limit on scalability is a well known result, however, it is apparent that properties of the simulated workload can lead to load imbalance within the simulator which in turn can have a dramatic impact on the scalability of the simulator. Quoting performance on a single benchmark does not fully quantify the parallelization of the simulator, as it is highly workload dependent.

Falsafi and Wood [22] constructed a detailed theoretical model of the WWT's performance incorporating numerous additional effects unique to their framework. Parallel Sulima appears much more susceptible to load balancing concerns owing to its interpretive nature, and its much narrowed simulation quanta, compared to the WWT. A more detailed simulator which fully modelled all internal activity (instead of approximating some) on a cycle by cycle basis may scale

better owing to a reduced dependency upon the workload.

While this analysis has referred to the active backplane, it applies equally to the passive (although the passive backplane is also constrained by lock contention).

VI. SPEED-ACCURACY TRADEOFFS

It is well understood that the round-robin scheduling of the simulation of multiprocessors can affect accuracy by permitting unrealistic interleaving of events, an effect which becomes more pronounced as the simulation quantum increases. This affects the parallel simulation of the passive backplane of Sulima for similar reasons. As simulated processors are not kept in lock-step and are dependent upon host lock ordering, interleaving may vary from run to run and is also dependent upon simulation speed.

Let Δt denote the value of the simulation quantum, in terms of simulated cycles. A value of $\Delta t = 1$ will minimize this effect, but will result in a reduction in parallel simulation speed due to the increased frequency of barriers and the load imbalance created. As in serial simulation, increasing the value of Δt may improve simulator speed, due to greater opportunities for reuse of data representing a simulated processor's state.

In order to examine the speed-accuracy tradeoff in choosing the value of Δt , a measure of accuracy must be chosen. The most obvious is the error in simulated time (or equivalently the number of simulated cycles); however, a finer measure would be to compare errors in the counts of all simulated hardware events. These include counts of cycles lost due to different types of stalls (e.g. due to D/E-cache misses, floating point register dependencies and a full store buffer).

We evaluated serial and parallel multiprocessor simulator accuracy on these measures firstly using the benchmarks from Section IV and the whole of the S class of OpenMP NAS Parallel Benchmarks². Simulated events were counted over the parallel region of the benchmark run.

Figure 6 gives the relationship between error and speedup as a function of Δt . These are relative to the event counts (and simulation speed) at $\Delta t = 1$, which is assumed to be the most accurate. For the NAS S class, the averaged absolute error in the number of cycles and the stall cycle counts are given. The latter is computed as the sum of absolute errors for stall cycles of each type, divided by the total cycle count. This will indicate whether stall cycles were attributed to the correct cause. For the other benchmarks, the error in the simulated cycle counts was given; a negative value means the simulated time was lower than at $\Delta t = 1$. Relative speed is indicated on the left axis, while percentage error in event counts is indicated on the right.

is.w, *sharetest* and *spintest* showed very little errors, so their results are not shown here. The NAS S class benchmarks show a larger error, especially in *ft.S*, *cg.S*, *mg.S*, and *lu-hp.S*. The latter two were the only benchmarks to show a significantly higher error in stall cycle counts than in total cycles. A detailed inspection of the stall

²Excluding *ep.S* and *ua.S*, as these take a very long time to simulate.

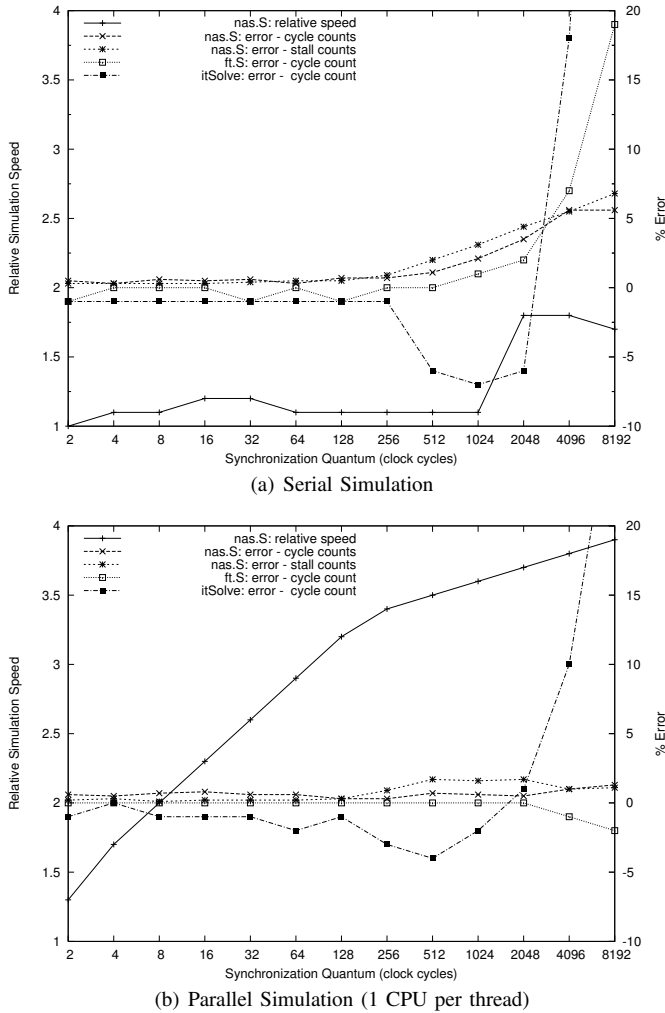


Fig. 6. Relationship between simulation time (synchronization) quantum, speedup and accuracy for simulation of a 4 CPU UltraSPARC III

cycle counts indicated that E-cache events related to sharing accounted for much of this difference, and only benchmarks with a large degree of sharing were likely to exhibit this behaviour. To confirm this, a mini-benchmark called *itSolve* was selected: it solved a small dense matrix of size $n = 60$ iteratively. At each step, memory sharing effects included a reduction, a barrier and an invalidate / cache-reload of a vector of length n . The error at $\Delta t = 8192$ was 69% and 30% for serial and parallel simulation, respectively. As expected, the error decreased as the matrix size n increased; for example at $n = 160$ and $\Delta t = 8192$, the error reduced to 27% and 14% respectively.

For most benchmarks, serial simulation became steadily more optimistic (negative error value) as Δt increased. This indicates fewer E-cache events occurring, such as cache line ping-ponging. An exception was for *ft.S*, and *itSolve*, where the error became sharply pessimistic for $\Delta t \geq 2048$. This indicates that the simulated application was spending much longer in barriers and other synchronization events. Both of these effects are due to the unrealistic memory event

interleavings produced by a large value of Δt .

Parallel simulation proved much more accurate, tolerating even large Δt for all benchmarks except *itSolve*, provided the host was not loaded. This is due to threads remaining roughly synchronized over the quantum, producing mostly realistic interleavings of memory events.

As expected, parallel simulation also had a much larger degree of speedup (at $\Delta t = 8192$, this was normally between 3 and 4; but *sharetest* had a speedup of 6.9). For serial simulation, the speedup was more modest; the increase for larger values of Δt is partially explained by the simulation becoming optimistic for the *ft.S* benchmark.

With the caveat that there is a strong dependence on the benchmark chosen, the ranges $16 \leq \Delta t \leq 128$ (serial) and $128 \leq \Delta t \leq 256$ (parallel) represent suitable values for the speed-accuracy tradeoff for Sulima.

It is noted that the error in event counts for simulation (particularly serial simulation) increases substantially above a quantum size of 256. The E-cache miss latency on this platform is approximately 250 cycles, and we conjecture that this increase in error is attributable to a single E-cache miss not forcing a stall to the end of the quantum, thereby allowing more atypical (and variable) event orderings.

For reference, the passive backplane serially simulates approximately 500k instructions per second and the active backplane serially simulates approximately 330k instructions per second for parallel workloads on a 900 MHz UltraSPARC II-Cu. These speeds are workload dependent.

VII. CONCLUSIONS

This work has considered two different approaches to the parallelization of an SMP simulator and has successfully demonstrated that in excess of threefold speedups on 4 host threads are possible on real workloads.

The first approach employs locking to ensure correct operation. This approach is simple to implement and is similar to previous approaches [10], [11]. Cache coherency is preserved, though it is rendered non-deterministic by the application of index-based locks within the caches to permit concurrent access. This approach ensures functional correctness.

As the threads are synchronized at the end of a quantum in simulated time, this results in a speed-accuracy tradeoff, which also applies to serial simulation. We have quantified this tradeoff over a variety of benchmarks, and found that parallel simulation is substantially more accurate for a given quantum, that accuracy can be compromised with a quantum larger than 256 cycles (a time interval corresponding to the E-cache miss penalty), and that both accuracy and speedup have a significant dependence on the workload being simulated.

In the second approach, PDES techniques similar to those employed in the WWTs [6], [7] have been adapted to parallelize a detailed interconnect model. By deriving sufficient lookahead from the cache coherency protocol, it was possible to allow individual processors to run in parallel, with the interconnect running serially. This approach demonstrated an effective speedup for certain workloads, and to the authors'

knowledge has not been employed on a shared memory platform.

Empirical and theoretical analysis has revealed that a workload induced load imbalance in the simulator may severely constrain speedup, owing to variations in the time required to simulate one quantum between threads. This is a workload-dependent effect which is likely common to most parallel simulators. Simulating multiple processors per thread ameliorates the problem by reducing the variation, which in turn acts to reduce the workload imbalance. Interestingly, a more detailed processor model may prove more scalable: a detailed model is likely to be slower (and therefore less affected by synchronization overhead), and also more likely to require some computation on each quantum (while Sulima may perform no work at all if blocked on a load miss). Finally this analysis demonstrates that any IPC variation between threads in a simulated workload can cripple speedup.

Parallel simulation is demonstrably feasible on modern shared-memory platforms, through either of the approaches demonstrated in this paper, or through prior efforts in the literature. We would suggest that the greatest benefits are to be gained from employing small numbers of threads on large-scale detailed processor simulators.

Two artifacts in our simulator implementation limit speed by increasing the variability in quantum simulation time: fast simulation of stall cycles and misses in the simulator's 'caching' of repeated calculations. Both are optimizations aimed at improving serial performance which end up compromising speedup, but not absolute performance.

Future work can consider extending the employed techniques to the simulation of both CMP and multichip multiprocessors (MCMP) platforms, and also the simulation of large scale CMPs (such as the UltraSPARC T1). Depending on the relative speed of simulation, multiple tiered techniques may be more suitable: employing the first approach for parallelization within each CMP dies and the second for parallelization between CMP dies. Furthermore, the effect of processor model features on simulation quanta (and the corresponding workload dependency) could be quantified further to confirm the suspicion that more detailed simulators are indeed more scalable.

The source code for the SparcSulima simulator, including its parallel implementation, is available from <http://ccnuma.anu.edu.au/sulima/>.

VIII. ACKNOWLEDGEMENTS

The authors wish to acknowledge the helpful feedback and proofreading efforts of colleagues, and the constructive comments provided by the reviewers. This work has been supported by Australian Research Council grant LP0347178 in conjunction with Sun Microsystems and Gaussian Inc.

REFERENCES

- [1] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay, "High-performance throughput computing," *IEEE Micro*, vol. 25, no. 3, pp. 32–34, May 2005.
- [2] S. R. Goldschmidt and J. L. Hennessy, "The accuracy of trace-driven simulations of multiprocessors," in *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. ACM Press, 1993, pp. 146–157.
- [3] M. Durbhakula, V. S. Pai, and S. Adve, "Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors," in *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, January 1999, pp. 23–32.
- [4] E. Witchel and M. Rosenblum, "Embra: Fast and flexible machine simulation," in *Proceedings of the 1996 SIGMETRICS conference on Measurement and Modeling of Computer Systems*, 1996, pp. 68–79.
- [5] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the SimOS machine simulator to study complex computer systems," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 1, pp. 78–103, Jan 1997.
- [6] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, "The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers," in *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. ACM Press, 1993, pp. 48–60.
- [7] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. Huss-Lederman, and J. R. Larus, "Wisconsin Wind Tunnel II: A fast, portable parallel architecture simulator," *IEEE Concurrency*, vol. 8, no. 4, pp. 12–20, October 2000.
- [8] M. Chidester and A. George, "Parallel simulation of chip-multiprocessor architectures," *ACM Transactions on Modeling and Computer Simulation*, vol. 12, no. 3, pp. 176–200, July 2002.
- [9] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," Computer Sciences Department, University of Wisconsin-Madison, Tech. Rep. TR-1342, 1997.
- [10] N. Manjikian, "Parallel simulation of multiprocessor execution: Implementation and results of SimpleScalar," in *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, Nov 2001, pp. 147–151.
- [11] J. Donald and M. Martonosi, "An efficient, practical parallelization methodology for multicore architecture simulation," *IEEE Computer Architecture Letters*, vol. 5, no. 2, p. 14, 2006.
- [12] B. Clarke, A. Czezowski, and P. Strazdins, "Implementation aspects of a SPARC V9 complete machine simulator," in *CRPITS '02: Proceedings of the Twenty-Fifth Australasian Conference on Computer Science*, vol. 4, 2002, pp. 23–32.
- [13] P. Strazdins, B. Clarke, and A. Over, "Efficient Cycle-Accurate Simulation of the UltraSPARC III CPU," in *CRPITS '07: Proceedings of the Thirtieth Australasian Conference on Computer Science*, Jan. 2007, to appear.
- [14] *UltraSPARC III Cu User's Manual*. Santa Clara, California, USA: Sun Microsystems, January 2004, version 2.21.
- [15] A. Charlesworth, "The Sun Fireplane System Interconnect," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*. ACM Press, New York, New York, USA, November 2001.
- [16] A. Over, P. Strazdins, and B. Clarke, "Cycle accurate memory modelling: A case-study in validation," in *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 85–96.
- [17] M. Thorn, "Using threading techniques to speed up SMP computer simulation," BSEng Thesis, Department of Computer Science, Australian National University, December 2004.
- [18] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, October 1990.
- [19] D. C. Burger and D. A. Wood, "Accuracy vs. performance in parallel simulation of interconnection networks," in *Proceedings of the 9th International Parallel Processing Symposium*, 1995, pp. 22–31.
- [20] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," NASA Advanced Supercomputing System Division, Tech. Rep. NAS-99-011, October 1999.
- [21] NASA Advanced Supercomputing, "NAS Parallel Benchmarks," January 2005, version 3.1. [Online]. Available: <http://www.nas.nasa.gov/Software/NPB/>
- [22] B. Falsafi and D. A. Wood, "Modeling cost/performance of a parallel computer simulator," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 1, pp. 104–130, 1997.