Performance Analysis of Four Memory Consistency Models for Multithreaded Multiprocessors

Yong-Kim Chong and Kai Hwang, *Fellow, IEEE*

***Abstract*—Stochastic timed Petri nets are developed to evaluate the relative performance of distributed shared memory models for scalable multiprocessors, using multithreaded processors as building blocks. Four shared memory models are evaluated: the *Sequential Consistency* (SC) model by Lamport (1979), the *Weak Consistency* (WC) model by Dubois et al. (1986), the *Processor Consistency* (PC) model by Goodman (1989), and the *Release Consistency* (RC) model by Gharachorloo et al. (1990). We assumed a scalable network with a sufficient bandwidth to absorb the increased traffic from multithreading, coherent caches, and memory event reordering. The embedded Markov chains are solved to reveal the performance attributes. Under saturated conditions, we find that multithreading contributes more than 50% of the performance improvement, while the improvement from memory consistency models varies between 20% to 40% of the total performance gain. Petri net models are effective to predict the performance of processors with a larger number of contexts than that can be simulated in previous benchmark studies. The accuracy of these memory performance models was validated with the simulation results from Stanford University. Our analytical results reveal the lowest performance of the SC model amongst four memory consistency models. The PC model requires to use larger write buffers, while the WC and RC models require smaller write buffers. The PC model may perform even lower than the SC model, if a small buffer was used. The performance of the WC model depends heavily on the synchronization rate in user code. For a low synchronization rate, the WC model performs as well as the RC model. With sufficient multithreading and network bandwidth, the RC model shows the best performance among the four models. Furthermore, we discovered that cache interferences cause very little performance degradation in all relaxed memory consistency models; as long as the network is contention-free even when multithreading has saturated the system.**

***Index Terms*—Distributed shared memory, memory consistency models, stochastic Petri nets, scalable multiprocessors, latency hiding techniques, multithreaded processors, context switching, performance evaluation.**

I. Introduction

TODAY'S scalable multiprocessors are mostly built with a distributed shared memory architecture [6]. The memory is physically distributed but logically shared [20], [23]. In other words, the address spaces generated by all processing nodes globally form a single address space. The main advantage of such a system lies in the scalability with distributed hardware and programmability of a shared virtual memory [17]. Representative systems include the Cray T3D [10], the Stanford Dash [22], the MIT Alewife [4], the Teracomputer [2], and Convex SPP [9].

A scalable system should be able to hide the long latencies of remote memory accesses. Several latency hiding techniques have been proposed: *Coherent caches* reduce the frequency of remote memory accesses by caching data close to the processor [22]. *Relaxed memory consistency* allows reordering of memory events and buffering or pipelining of remote memory accesses [1], [13], [15], [16], [29], [8]. *Data prefetching* attempts to hide long read latency by issuing read requests well ahead of time, with the expectation that the data will be available in the cache when it is referenced [21], [26]. *Multithreading* attempts to hide the long latency by context switching between several active threads, thus allowing the processor to perform useful work while waiting for remote requests or synchronization faults to complete [2], [3], [4], [24], [25], [28]. Most of these studies are based on simulation results. Several analytical models were developed in [24], [25], [3], [4] for multithreading and in [26] for data prefetching.

This paper develops stochastic models of multithreaded processors, equipped with hardware coherent caches under different memory consistency models. A consistency model defines the event ordering by which the memory accesses from one process should be observed by other processes in the system. It imposes restrictions on the order of shared memory accesses initiated by each processor. This translates to the amount of write buffering and pipelining of memory accesses allowed in each processor to hide the latency. We evaluate the *Sequential Consistency* (SC) memory model [18], the *Processor Consistency* (PC) memory model [16], the *Weak Consistency* (WC) memory model [12], and the *Release Consistency* (RC) memory model [15]. The *Generalized Stochastic Petri Net* (GSPN) [5] is used to model concurrency and synchronization operations, which are not possible under a queuing model. Petri net models are validated against simulation results from Stanford. Two orthogonal approaches: *processor multithreading* and *memory consistency relaxation*, are shown indeed complementary in hiding latency. The performance is shown not only sensitive to architectural properties, but also to program behavior. Our studies include the effects of cache coherence, hit ratio, read-write ratio, write buffer size, consistency constraints, synchronization rate, and network latency, etc.

To focus our studies on the effects of multithreading and memory relaxation, we assume write-through caches with coherence supported by hardware, as in Stanford Dash, and a scalable network which is not limited in bandwidth. We did

Manuscript received May 15, 1993; revised Mar. 15, 1994.

Y.-K. Chong is with the School of Electrical and Electronic Engineering, Nanyang Technological University, Nanyang Ave., Singapore 2263.

K. Hwang is with the Department of Electrical Engineering—Systems, University of Southern California, Los Angeles, CA 90089-2562. e-mail: kaihwang@aloha.usc.edu.

IEEECS Log Number D95045.

1045-9219/95$04.00 © 1995 IEEE

not include the effects of network contentions, data prefetching, or the use of write-back caches, because these were reported in previous studies by [3], [29], and [26]. We assume block context switch on cache misses leading to remote memory access [3], [4]. The effects of increased cache misses and context switching overhead are absorbed by the high network bandwidth assumed.

This paper is organized as follows: Section II presents a basic GSPN model for an abstract multithreaded processor. In Section III, the basic processor model is extended to model a scalable multiprocessor with coherent caches and the SC consistency memory. Architectural assumptions and program parameters are specified in the model. In Sections IV to VI, the GSPN model is extended to model the PC, WC, and RC memory models, respectively. Performance curves are obtained for four consistency models under variations of the key machine and program parameters. In Section VII, we include a simple cache degradation model to improve the accuracy of the GSPN models by considering the caching effects. The performance curves predicted by the GSPN models are compared against the simulation results from Stanford University. This validates the accuracy of the performance models being presented. Finally, we summarize research contributions and comment on further work needed to integrate various latency hiding techniques in the same system.

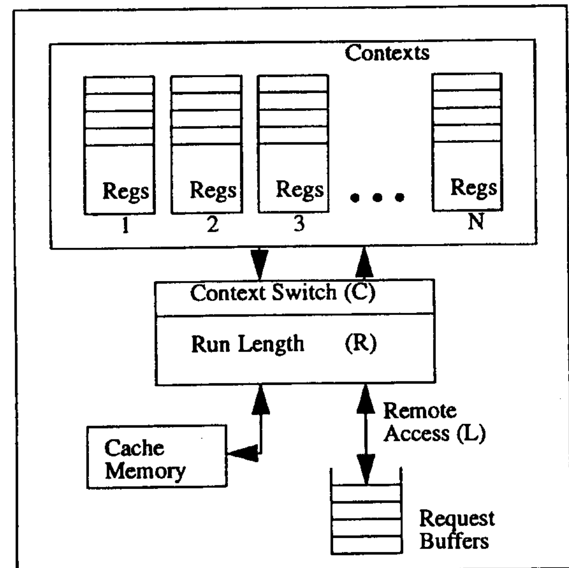
II. Modeling Multithreaded Processors With Petri Nets

Fig. 1 shows the basic model of a multithreaded processor containing multiple hardware contexts. Consider N threads or contexts running on each processor. Each context executes for a *run-length* of R cycles before making a local or remote memory access upon a cache miss. The process will be switched out and replaced by another ready-to-run process. The *context switch overhead* is C cycles. The average *local or remote memory latency* takes L cycles. In the basic model, we only consider the average latency between local and remote memory accesses for the time being. But we will consider them separately in the refined models to be presented in the subsequent sections.

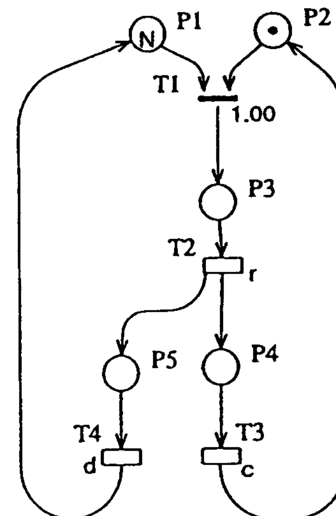
A number of assumptions were made: Pipelining of remote memory access is allowed, with multiple servers in the network, thus allowing multiple outstanding write requests to be serviced simultaneously. L is assumed much larger than C since it defeats the purpose of multithreading if L is equal or smaller than C. N is assumed a constant close to the saturation point, meaning that there is sufficient parallelism for execution within each processor.

The performance of a multithreaded processor is measured by the *processor efficiency*, given as the ratio of the busy time that the processor is executing useful instructions to the total time elapsed. The purpose is to keep the processor busy with sufficient contexts and to reorder memory events in order to overlap communication latency with useful CPU computations.

The GSPN model is constructed from the abstract processor shown in Fig. 1. Parameters R, C, and L are random variables with exponential distributions. Hence the *probability of a remote memory request* is given by $r = 1/R$. Similarly, the *service rate for performing context switch* is $c = 1/C$ and the *service rate for remote memory accesses* is $l = 1/L$. Single cycle latency is assumed to access the cache or local memory, because the remote latency is assumed much greater. The value of R varies with the actual run-length of each thread. The latency L is affected by the distance traversed and the network traffic density. The context switch overhead C is related to the process state complexity. In general, $L > R > C$.



(a) Abstract processor



(b) The GSPN model

Fig. 1. A multithreaded processor model.

The Petri net model works as follows: The place P1 is initialized with N tokens representing N contexts ready for execution at the start time. P2 is initialized with one token to

represent a single processor available for program execution. A token present in place P3 or P4 indicates that the processor is *running* or *context switching*, respectively. Finally, tokens present in place P5 indicates the number of outstanding remote memory requests being serviced. The places P4 and P5 represent concurrent context switching and remote memory access.

T1 is an immediate transition corresponding to a gate to execute the next ready-to-run context. This transition is enabled only when the processor is free (a token in P2) and there is at least one ready-to-run context (token or tokens in P1). Together, P2 and T1 ensure that the processor is either in a *running* state or in a *context switching* state, but not in both. Transitions T2 and T3 represent the *remote memory access rate* (r) and *context switching service rate* (c), respectively. Transition T4 represents the service rate $d = M_5 \times l$ of remote memory accesses, where M_5 is the number of tokens present in P5 at any instant of time. The remote memory accesses are assumed pipelined, thus allowing multiple outstanding requests to be serviced concurrently. Hence for m outstanding requests, the expected time to receive a reply is L/m, as compared to L for a single outstanding request. This increases the service rate by a factor of m.

The set of states that a process enters after each transition is represented by a *reachability tree* [5]. This allows us to construct an associated *Embedded Markov Chain* (EMC) of the GSPN model. A state in the EMC is considered *tangible*, if the process spends a finite amount of time in that state, and *vanishing* otherwise. The steady state solution of the Markov chain is obtained by solving the following linear system of equations:

$$\pi = \pi \cdot U \tag{1}$$

where U is the *transition probability matrix and*

$$\pi = \left(\pi_1, \pi_2, \pi_3, \ldots, \pi_{S_N}\right)$$

is the *stationary probability vector* of the EMC. The element u_{ij} of matrix U corresponds to the transition probability from state i to state j of the EMC.

In order to solve (1), we need to find the steady state probability π_j of an arbitrary state j. This can be obtained from the equation:

$$\pi_j = \frac{V_{ji} E\left[W_j\right]}{C_i} \tag{2}$$

where V_{ji} is the mean number of visits to state j between two adjacent visits to a reference state i, $E[W_j]$ is the average time in state j, and C_i is the mean cycle time of state i [5].

Given specific values of the parameters r, c, and l, Gaussian Elimination can be applied to solve the linear system of equations. The *processor efficiency* with N contexts, E_N, is obtained as the probability of having a token in place P3 at the steady state:

$$E_N = \sum_{j \in G} \pi_j \tag{3}$$

where G is the set of tangible states with a token in P3.

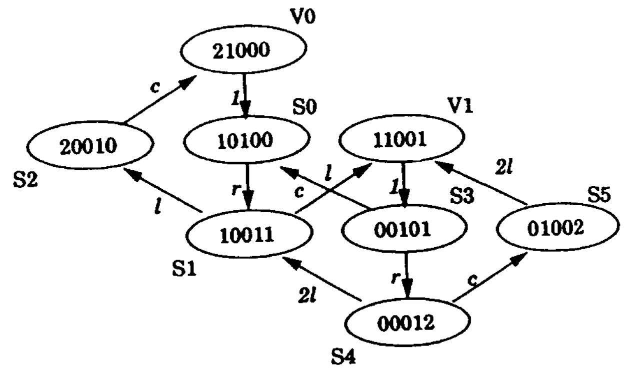
To illustrate the steps involved, we consider the GSPN model for the case of $N = 2$. The corresponding EMC state diagram is shown in Fig. 2. Each state is represented by a 5-tuple $\{M_1\ M_2\ M_3\ M_4\ M_5\}$ corresponding to the numbers of tokens in five places. S_i and V_i represent the *tangible* and *vanishing* states, respectively. The total number of states used is given by $S_N = 2\ (N + 1)$. Note that S_N is independent of the parameters R, C, and L.

The transition probability matrix U is given below:

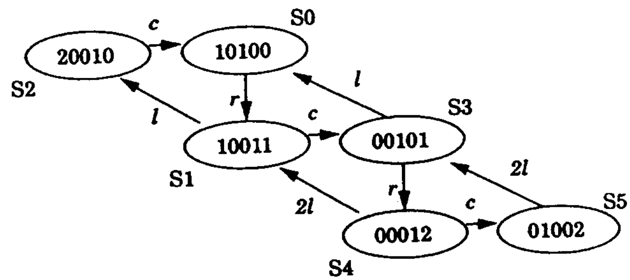
$$U = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{l}{l+c} & \frac{c}{l+c} & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ \frac{l}{l+r} & 0 & 0 & 0 & \frac{r}{l+r} & 0 \\ 0 & \frac{2l}{2l+c} & 0 & 0 & 0 & \frac{c}{2l+c} \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

The processor efficiency for two contexts, E_2, is obtained by adding the probabilities associated with the states S0 and S3, $E_2 = \pi_0 + \pi_3$. With $N = 2$, $R = 16$, $C = 2$, and $L = 130$, we obtain $\pi_0 = 0.02$ and $\pi_3 = 0.19$, giving an efficiency of $E_2 = \pi_0 + \pi_3 = 0.21$.

For large value of N, it is only feasible to solve the system with available software tools. The GreatSPN was developed by Chiola [7] to analyze the GSPN with a reasonably large state space. It provides graphical input facilities for building the GSPN model, extracting the reachability graph, and using Gauss-Seidel iterative method to solve the EMC.



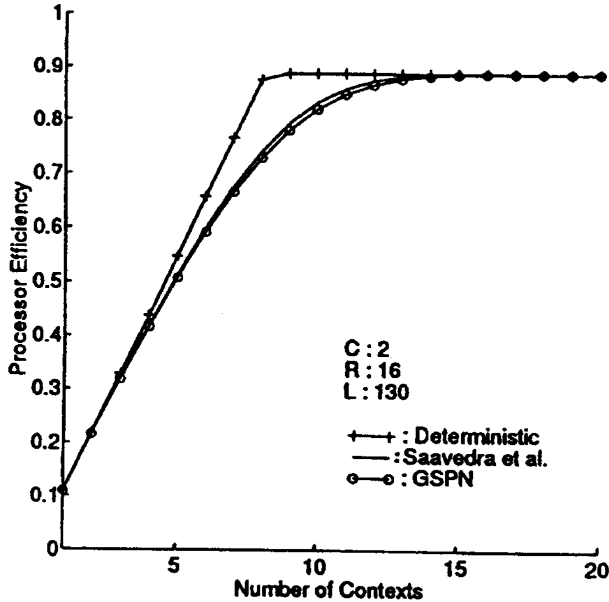
(a) Embedded Markov chain



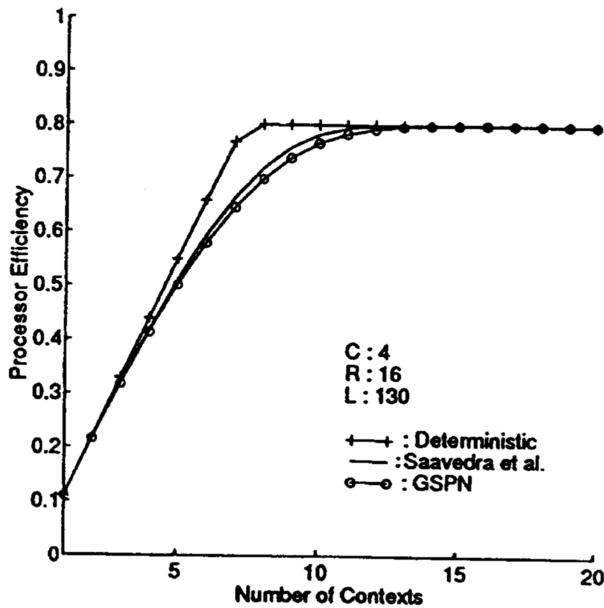
(b) Reduced embedded Markov chain

Fig. 2. State transition diagrams for the GSPN model of $N = 2$ contexts.

The processor efficiency is computed for $C = 2$ and 4 cycles, $R = 16$ cycles, and $L = 130$ cycles. To verify the accuracy of the GSPN model, our results are compared in Fig. 3 with the results predicted by [24], [25]. In both cases, the results are very close and almost overlap for small values of N. The deterministic curves shown are obtained by performing a deterministic analysis on processor efficiency for fixed values of R, L, and C, as derived in [24].



(a) Context switch overhead $C = 2$ cycles.



(b) Context switch overhead $C = 4$ cycles.

Fig. 3. Processor performance curves for $R = 16$ and $L = 130$ cycles.

III. SEQUENTIAL CONSISTENCY MEMORY MODEL

Multiprocessor system is first studied with the *Sequential Consistency* (SC) memory model. The system consists of P processing nodes connected through a scalable network. Physical memory is distributed among all processing nodes, forming a single shared virtual memory space. Distributed cache coherence is maintained using an invalidating distributed directory-based protocol, similar to that developed in Stanford Dash system [22].

All instructions and private data are stored locally in each node, with a high cache hit ratio. We assume one processor cycle to access the cache. The caches are assumed to be *lock-up free* using a write-through policy. The *name space* is sufficiently large to cater for multiple outstanding memory requests from different threads. A memory request may be satisfied locally, either from the cache or from the local memory, or from the memory of a remote processing node. Until Section VII, we ignore the caching effects for clarity purpose.

All reads are assumed to be *blocking*, thus no multiple read requests are outstanding for a single thread. Independent write requests from different threads may be pipelined, thus allowing concurrent accesses to local and remote memories. The pipeline stages are assumed sufficiently large to cater for all pending independent accesses from different threads. Following the reply from a read request, a cache fill operation is performed without stopping the processor from executing another thread. All synchronization variables for *acquire* (*lock*) and *release* (*unlock*) operations are assumed cachable, requiring the same amount of time of a normal memory operation.[1]

Based on the memory access constraints imposed on SC [12], [13], we specify the SC memory operations as follows:

A. For each thread:
- a. *Read*: Processor issues a read access and waits for the read to perform. Context switch on a cache miss.
- b. *Write*: Processor issues a write access and waits for the write to perform. Context switch on a cache miss.
- c. *Acquire*: Same as *Read*.
- d. *Release*: Same as *Write*.

B. Pending accesses from independent threads satisfying the above conditions can be pipelined.

An access is considered performed with a hit in the cache, or when a reply is received from the local or from a remote memory. Although buffering of write accesses within each thread is possible under SC, the potential gain is rather limited. This is due to the fact observed in [13] that in most applications, read and write accesses are well interleaved, thus making most of the write latency visible to the processor.

The *run length* (R), *local and remote memory access latencies* (L_l, L_r), *context switch overhead* (C), and *cache fill overhead* (Y) are all random variables having exponential distributions. All threads are assumed independent, so that accesses from different threads in the same node can be pipelined. Listed below are basic parameters used in our analysis:

1. We ignored the issues of load imbalance and synchronization overhead, which is beyond the scope of this paper.

N: Number of contexts running in each processor ($N \geq 2$)

s: The shared-memory reference rate, which is the inverse of the average run length between two consecutive shared-memory references ($s = 1/R$).

s_w, s_{rel}: The fractions of shared-memory *writes* and *releases*, respectively.

s_r, s_{acq}: The fractions of shared-memory *reads* and *acquires*, respectively.

m_r, m_w: The *cache miss rates* for shared read and write operations, respectively. The respective hit ratios $h_r = 1 - m_r$ and $h_w = 1 - m_w$.

P_l, P_r: The fractions of cache misses being serviced by a *local* or a *remote* memory, respectively, with $P_r = 1 - P_l$.

c: The service rate in performing context switching, which is the inverse of the context switching overhead, $c = 1/C$.

l_l, l_r: The service rates for *local* or *remote* shared memory accesses, given by $l_l = 1/L_l$, $l_r = 1/L_r$.

y: The service rate in performing cache fill operation, which is the inverse of the cache fill time, $y = 1/Y$.

The parameters s, s_w, s_{rel}, s_{acq}, and s_r depend on the program behavior. The cache miss ratios depend also on program behavior and cache architecture. Parameters P_l and P_r are determined by program behavior and data distribution among the nodes. Parameters c, l_l, l_r, and y are determined primarily by machine architecture.

The multithreaded processor under the SC model can be conveniently represented by a GSPN model shown in Fig. 4. Two separate branches represent the read and write operations as described below: Place P1 is initialized with N tokens representing N ready-to-run contexts. Place P2 is initialized with one token corresponding to one processing node. Transition T1 serves a gate to access the processing node. A token in place P3 indicates that the processor is in a *busy* state. A maximum of one token can be present in P3 at any time, as controlled by the transition T1.

The timed transition T2 is assigned with a firing rate s, corresponding to memory accesses by each thread. Transitions T3, T4, T5, and T6 are assigned with switching distributions s_w, s_{rel}, s_{acq}, and s_r, respectively, representing the probabilities of *write*, *release*, *acquire*, and *read* operations, respectively. Note that $s_w + s_{rel} + s_{acq} + s_r = 1$.

Transitions T9, T10, T11, and T12 correspond to the hit and miss ratios of the coherent cache. T9 and T10 are assigned with probabilities of h_w and m_w, respectively, for write accesses, whereas T12 and T11 are assigned with h_r and m_r, respectively, for read accesses. A cache hit will place a token in P3, returning the processor back to a busy state immediately. A cache miss will put the processor in a context switching state, represented by a token in place P9. Transition T13 fires at a rate c, to signify the completion of a context switching operation. Transitions T14, T15 are assigned with probabilities p_l and p_r, for a write request being serviced by a local or by a remote memory. Similarly, probabilities can be assigned with T16 and T17 for read requests.

Tokens present in places P12 and P15 represents pending memory requests for local memory. These requests are serviced at rates determined by firing the rates $d1 = M_{12} \times ll$ of T18 and $d4 = M_{15} \times l_l$ of T21. Similarly, tokens present in places P13 and P14 represent pending requests for remote memories, with firing rates $d2 = M_{13} \times l_r$ for T19 and $d3 = M_{14} \times l_r$ for T20. Transition T22 corresponds to the cache fill operation, with a firing rate y. The completion of a read or a write access places a token back to P1, adding to the number of ready-to-run contexts for next switching.

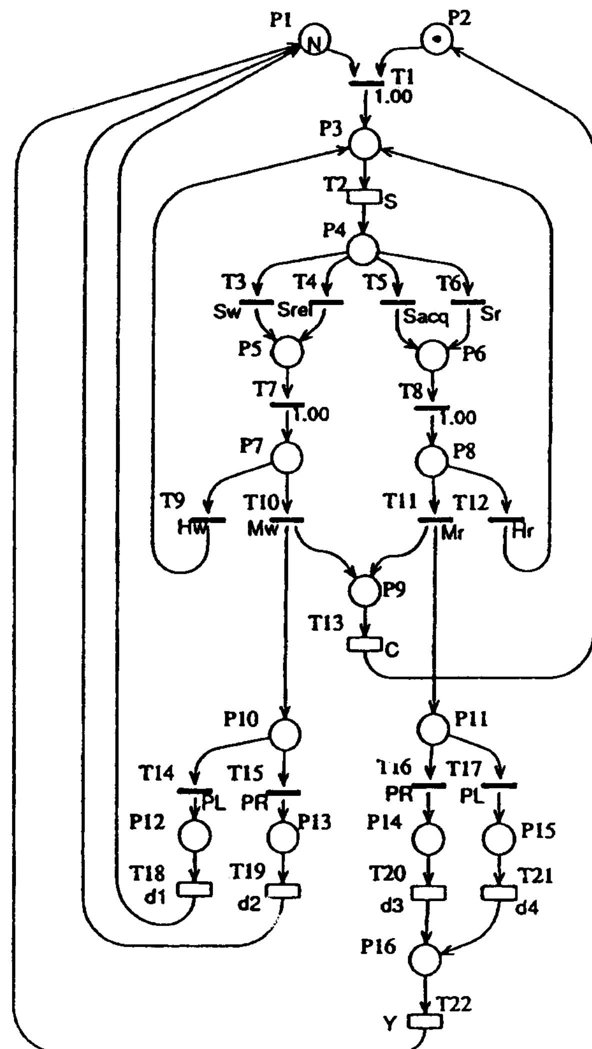


Fig. 4. The GSPN model for the sequential-consistency shared memory.

Table I lists the default parameters chosen to evaluate the effects of parameter changes on different memory consistency models. The cache fill overhead Y is set to 8 cycles by assuming a cache line of 16 bytes on a 32-bit memory system. These

default values are chosen to represent a hypothetical application. The choices are by no means to cover the wide variance in different applications. In order to limit the number of states to be generated by the GSPN models, we plotted all processor efficiency curves up to $N = 6$ contexts. The machine parameters are chosen to be close to those used in a small Dash prototype.

Previous studies [24], [14], [4] have shown that a small number of contexts ($N \leq 4$) are sufficient to achieve high processor efficiency. The processor efficiency at saturation is obtained with a deterministic analysis. The *effective run length*, R_{eff}, is given by:

$$R_{eff} = \left(\frac{1}{(s_w + s_{rel})m_w + (s_r + s_{acq})m_r} \right) \left(\frac{1}{s} \right) \quad (4)$$

and the *processor efficiency* at saturation, E_{sat}, is given by:

$$E_{sat} = \frac{R_{eff}}{R_{eff} + \frac{1}{c}}. \quad (5)$$

TABLE I
DEFAULT PARAMETER VALUES FOR THE GSPN MEMORY MODELS

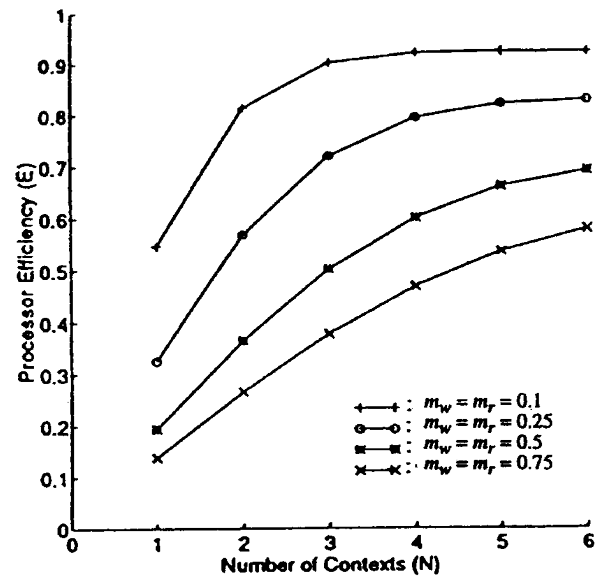
Parameters	Default Value	Remarks
s	0.2	$R = 5$
s_w, s_r	0.33 , 0.66	
s_{rel}, s_{acq}	0.006, 0.004	
m_r, m_w	0.6 , 0.4	
P_l, P_r	0.6 , 0.4	
c	0.25	$C = 4$
l_l	0.055	$L_l = 18$
l_r	0.014	$L_r = 73$
y	0.125	$Y = 8$

Fig. 5 shows the performance curves under the SC memory model. The contribution of coherent cache is seen by changing the parameters m_r and m_w in Fig. 5a. A lower cache miss ratio increases the effective run length of each thread, thus improving the processor efficiency in both the linear and the saturation regions. The case of $m_r = m_w = 1$ represents a multiprocessor system without coherent caches, where all shared references results misses, followed by a context switch.

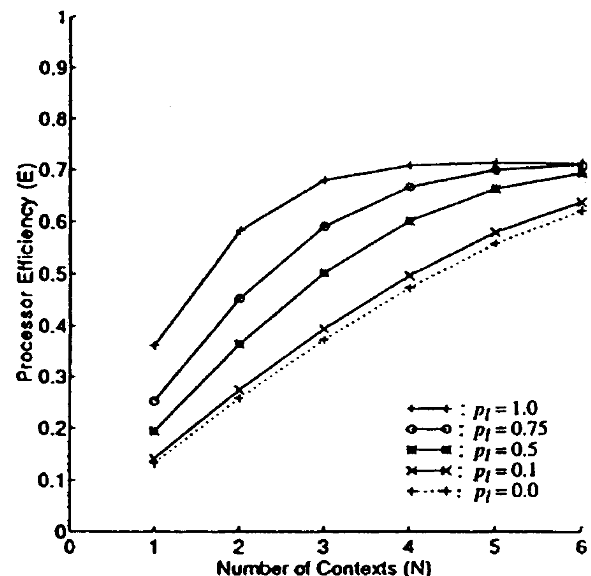
Since no buffering is allowed for write accesses under the SC model, no significant change in the processor efficiency is observed by varying s_w. The difference between read and write latencies is the extra cache fill time incurred with read accesses. In most cases, we have $L_l, L_r >> Y$. Similarly, changes in cache refill latency does not have significant effects on the processor efficiency since we assumed that cache refill operation does not stall the processor.

Fig. 5b shows the effect of varying P_l on the processor performance. A higher value of P_l effectively reduces the average latency seen by a processor, thus increasing the processor efficiency in the linear region. This can be achieved by a static allocation of frequently used data to the local memory of a processor node, with prior knowledge of the access patterns. In the case of a uniform access pattern with equal probability to each node, $P_l = 1/P$, where P is the number of processing nodes in the system. The processor efficiency at saturation is not affected by P_l, as seen in (4) and (5).

In general, multithreading performs poorly under the SC model because switching on both read and write misses can quickly exhaust the supply of ready-to-run threads. Coherent caches generally improve the processor's performance by increasing the effective run length of each thread. Frequently used data should be allocated to local memory of each node. This will reduce the effective memory access latency, thus improving the overall processor performance.



(a) With different cache misses m_w and m_r



(b) With different memory-access patterns P_l

Fig. 5. Performance of the sequential consistency memory model.

IV. Processor Consistency Memory Model

The *Processor Consistency* (PC) model introduced by Goodman requires that writes issued from the same processor are always in program order, but the order of writes from different processors can be observed differently. This allows reads following a write to bypass the write, thus allowing for buffering of write accesses. We specify the memory events under the PC memory model as follows:

A. For each thread:

a. *Read*:	Processor issues a read access and waits for the read to perform. Context switch on a cache miss.
b. *Write*:	Processor sends a write to the write buffer, stall if the buffer is full. A write request is retired from the buffer only after the write is performed.
c. *Acquire*:	Same as *Read*.
d. *Release*:	Same as *Write*.

B. Pending accesses from independent threads satisfying the above conditions can be pipelined.

No context switching is taking place for a write operation, since writes are sent to the write buffer. We define B as the size of the write buffer for all threads. Clearly, B should be large enough to prevent processor from idling.

To model the multithreaded processor under the PC model involves an accurate estimation of the number of independent write accesses that can be pipelined. This defines the number of servers for write accesses at any point in time. Fig. 6 shows the GSPN model using a *dynamic server allocation* scheme as described in [8]. The basic structure of the Petri net is similar to that for the SC model, with $d1$ to $d4$ defined by $d1 = M_{15} \times l_l$, $d2 = M_{16} \times l_r$, $d3 = M_{17} \times l_r$, and $d4 = M_{18} \times l_l$.

Place P11 is initialized with B tokens representing the available buffer size. Transition T7 represents the buffering of write requests, firing only if there is an available buffer entry. This puts the processor back to a *busy* state, with a pending write to be performed. In order to limit the number of states of the EMC, a write *hit* is modeled with an immediate transition T10, assuming a single cycle operation. This is possible since $L_l >> 1$ and $L_r >> 1$ in general.

Place P10 contains tokens representing the number of available servers for pipelined accesses. Instead of assigning a fixed value in P10, the number of servers is determined by the number of active threads in the processor. A thread is considered active if it is currently running or waiting for a read request to complete. All the pending write accesses are assumed from the current active threads. A token is added to P10 when a new context starts running to allow additional pipelining. Likewise, a token is removed from P10 when a read request is complete. A high switching rate of 10 is assigned to the transition T25 to ensure that a negligible time is wasted in removing a token from P10. In the extreme case of $s_w = 1$, only a single server is allocated, thus the processor consistency is preserved.

We define the *processor stalling probability*, $P\{stall\}$, as the steady state probability of processor being stalled during execution of a thread. Hence for the PC memory model, a processor stalls when the write buffer is full. The stall probability is determined by the buffer size, the write rate, and the service rate of the write buffer.

Fig. 7 shows the performance and stall probability of the PC memory model, with different buffer sizes and write access rates. For a single thread, the PC model always performs better than the SC since the latter would stall on each write, whereas the PC would stall only when the write buffer is full. As N increases, the performance gain over the SC model slowly diminishes since the stall probability increases with a higher rate of write accesses. For a small buffer size, the fraction of processor stalling time could be very high, offsetting the gain from buffering. Increasing the buffer size generally improves the processor's performance. The same effects are observed when the fraction of write accesses s_w is increased for a fixed buffer size, as shown in Fig. 7c. Instead of a higher performance with an increased s_w, a performance loss is observed at a higher degree of multithreading due to excessive processor stalling time.

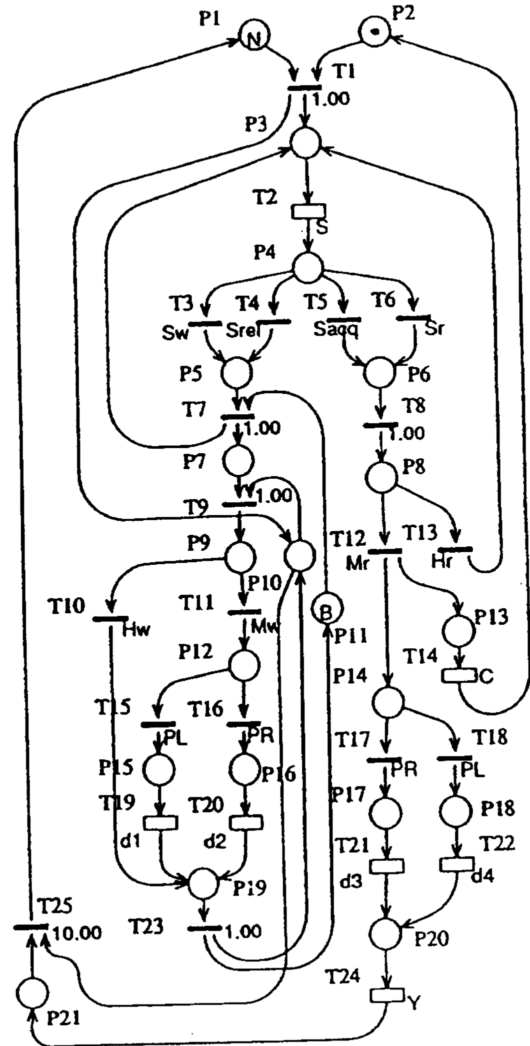
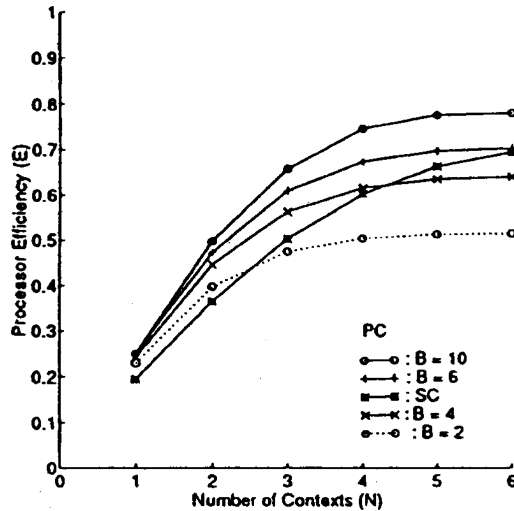
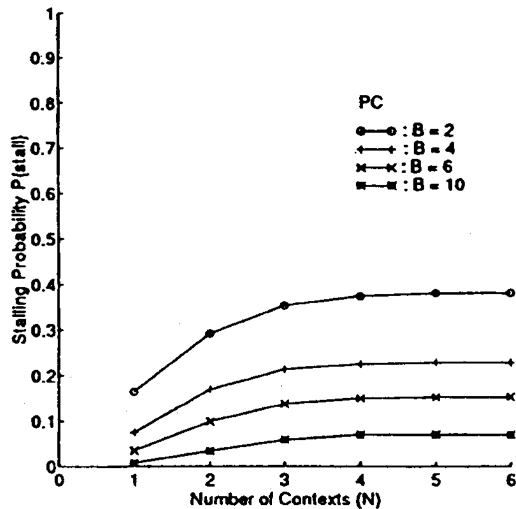


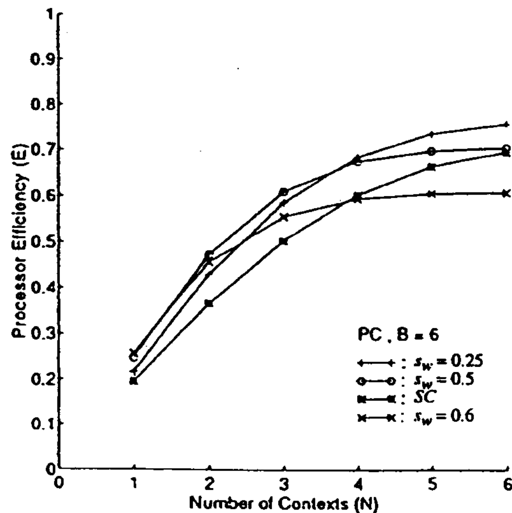
Fig. 6. The GSPN model for the processor-consistency shared memory.



(a) Effects of different buffer size B



(b) P{stall} values with different buffer size B



(c) Effects of different write rate s_w.

Fig. 7. Performance of the processor consistency memory model.

V. Weak Consistency Memory Model

The *Weak Consistency* (WC) memory model proposed by Dubois et al. requires the shared memory be consistent only at synchronization points. This allows for pipelining of both read and write accesses between two synchronization points. The requirement of synchronization accesses being sequentially consistent may limit the expected performance gain in applications with higher synchronization rates. We define the multithreaded processor under the WC memory model as follows:

A. For each thread:

a. *Read*:	Processor stalls for a pending release to perform. Processor issues a read access and waits for the read to perform. Context switch on a cache miss.
b. *Write*:	Processor sends a write to the write buffer, stall if the buffer is full. Writes are pipelined.
c. *Acquire*:	Processor stalls for pending writes or releases to perform. Processor sends an acquire and wait for the acquire to perform. Context switch on a cache miss.
d. *Release*:	Processor sends a release to the write buffer, stall if the buffer is full. The release request is retired from the buffer only after all previous writes are performed.

B. Pending accesses from independent threads satisfying the above conditions can be pipelined.

No context switching is taking place for write and release operations in the WC model. Since all reads are blocking, pipelining of read accesses within each thread are not allowed. The write buffer is shared among all write and *release* requests. The GSPN model for the WC model is shown in Fig. 8. An additional branch is added to model the release operations. Main features of the Petri net are described below:

For write operations, no server limit is imposed. This implies that sufficiently large pipeline stages are available to process all pending write accesses. The inhibitor arcs from P18 and P19 to T10 are used to prevent the servicing of the next *release* operations with pending writes. All pending writes are assumed from the current running thread. This represents a lower bound on the expected performance. Alternatively, the inhibitor arcs may be removed if all pending writes are assumed from different threads, corresponding to an upper bound on the processor performance.

Similarly, the inhibitor arcs from P18 and P19 to T8 are used to prevent the servicing of *acquire* operations with pending writes. This is a potential bottleneck for applications with higher synchronization rate, since the processor will be stalled while waiting for writes to complete. Only a *single* server is allocated in P13 for the *release* operations to enforce sequential consistency constraint for synchronization accesses.

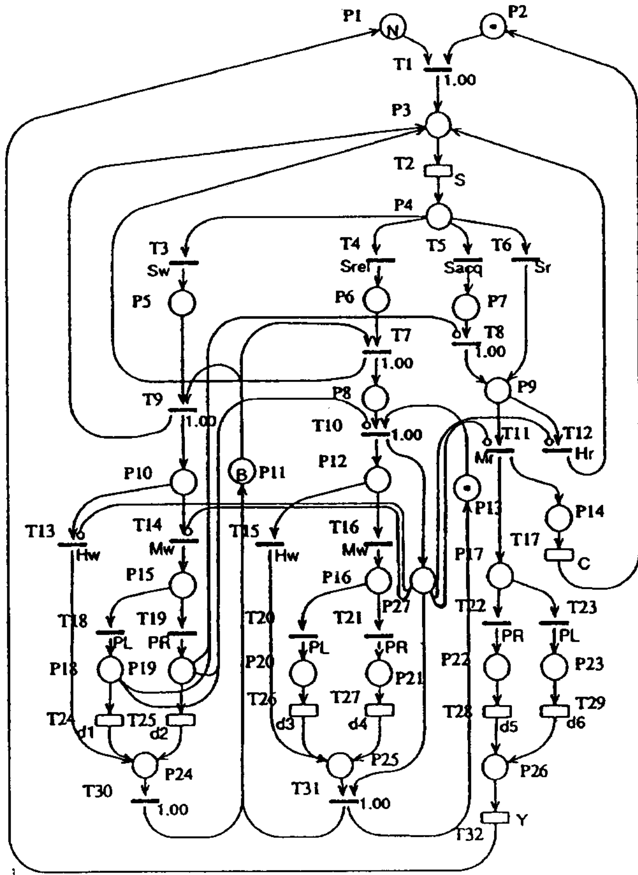


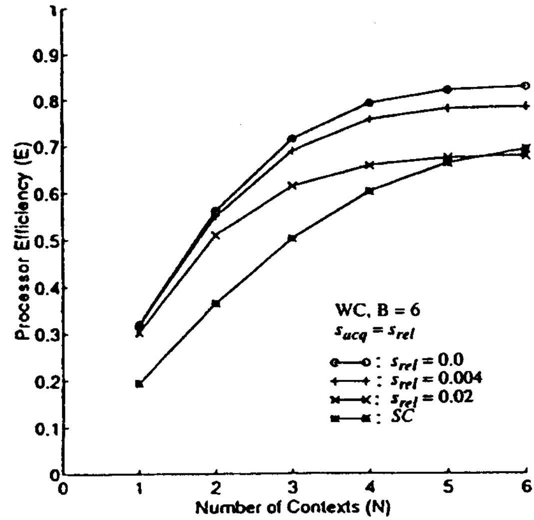
Fig. 8. The GSPN model for the weak-consistency shared memory.

Inhibitor arcs from P27 to T11 and T12 are used to prevent the servicing of *acquire* and *read* accesses before the pending *release* operation is complete. This poses another potential bottleneck for the WC memory model. Inhibitor arcs from P27 to T13 and T14 are used to prevent the servicing of *write* accesses before the pending *release* operation is complete. The write requests are placed in the write buffer without stalling the processor. The firing rates are defined as $d1 = M_{18} \times l_l$, $d2 = M_{19} \times l_r$, $d3 = M_{20} \times l_l$, $d4 = M_{21} \times l_r$, $d5 = M_{22} \times l_r$, and $d6 = M_{23} \times l_l$.

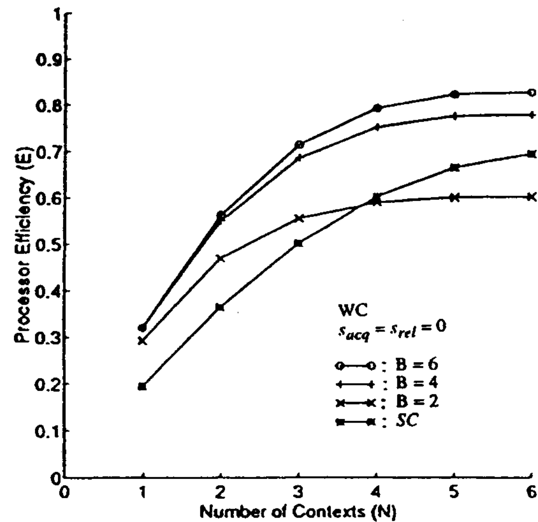
The *processor stalling probability* under the WC memory model is defined by:

$$P_{WC}\{stall\} = P\{buffer\,full\} + P\{acquire\ stall\} + P\{read\ stall\} \quad (6)$$

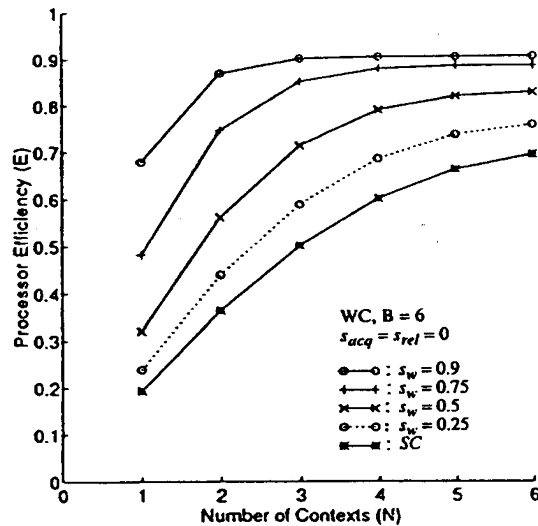
Fig. 9a shows the processor efficiency under the WC memory model with different synchronization rates. For a low synchronization rate ($s_{acq} \approx 0$), the gain in performance is significant, with all write latencies successfully hidden by the write buffer. The drop in performance under a higher synchronization rate arises mainly from the increased processor stall time from the *acquire* and *read* stalls. For applications with a very high synchronization rate, the WC memory model may result in a lower performance than the SC memory model due to excessive processor stalling time, if not properly controlled.



(a) Effects of different synchronization rates, s_{acq} and s_{rel}



(b) Effects of different buffer sizes, B



(c) Effects of different write rate s_w

Fig. 9. Performance of the weak consistency memory model.

Increasing the write buffer size generally improves the processor's performance under the WC memory model, as in the case of the PC model. The buffer size required to hide the write latencies completely is much smaller than that for the PC model due to a higher service rate for write requests under the WC model. Fig. 9b shows the effects on processor efficiency with different buffer sizes, under a very low synchronization rate. For the default, buffer size $B = 6$ is sufficient to hide all write latencies.

Since the write latencies are effectively hidden under the WC memory model with a low synchronization rate, we expect the performance to increase with a higher value of s_w. Fig. 9c shows the effects of different s_w values under the WC memory model. With a higher value of s_w, the performance gain over the SC memory is significant for a small number of contexts, until reaching the limit of a finite buffer size.

VI. Release Consistency Memory Model

The *Release Consistency* (RC) model proposed by Gharachorloo et al. is an extension of the WC memory model by exploiting the information on synchronization points using explicit *acquire* and *release* operations. This allows for pipelining of both read and write accesses within a pair of *acquire* and *release* as in the case of the WC model. In addition, *acquires* are allowed to bypass pending *releases* since synchronization accesses are processor consistent, thus providing higher potential for performance gain. Based on the sufficient conditions stated in [15], we specify the memory events under the RC memory model as follows:

A. For each thread:

a. *Read*:	Processor issues a read access and waits for the read to perform. Context switch on a cache miss.
b. *Write*:	Processor sends a write to the write buffer, stall if the buffer is full. Writes are pipelined.
c. *Acquire*:	Processor sends an acquire and wait for the acquire to perform. Context switch on a cache miss.
d. *Release*:	Processor sends a release to the write buffer, stall if the buffer is full. The release request is retired from the buffer only after all previous writes and releases are performed.

B. Pending accesses from independent threads satisfying the above conditions can be pipelined.

Same as in the PC and WC models, no context switching is taking place for write and release operations. Since reads are blocking, pipelining of read accesses within each thread is not allowed. The write buffer is shared among all write and release requests as in the case of the WC model.

The GSPN model for RC is shown in Fig. 10. The basic structure is the same as that for the WC model, except for the release accesses. The single server for release operation in the WC model is now being replaced by the dynamic server allocation structure used in the PC model. This is to model the synchronization accesses which are processor consistent under the RC model. All the inhibiting arcs from P13 have been removed since write, acquire and read accesses following release requests are allowed to bypass the pending releases. In addition, acquire accesses are allowed to be serviced without having to wait for pending write accesses to complete. In the extreme case when $s_w = 0$, RC model is equivalent to the PC model. The processor stalls under the RC model when the write buffer is full.

The processor performance for the RC memory model with different synchronization rates $s_{acq} \leq 0.02$ virtually overlaps with each other since the bottleneck for synchronization accesses has been removed. The processor stalling probabilities for all three cases are very small and thus can be ignored. The processor efficiency at saturation can be estimated using (4) and (5) by setting the parameter $m_w = 0$. The performance of the RC memory model for different values of B and s_w are the same as using the WC memory model with $s_{rel} = s_{acq} = 0$, as previously shown in Fig. 9b and Fig. 9c, respectively.

Fig. 11 summarizes the relative performance for all the shared memory consistency models, with respect to different synchronization rates and write buffer sizes. The corresponding processor stalling probability is shown with dotted lines to account for the lost of performance under different consistency models. In general, the RC memory model gives the best performance for the same buffer size. The WC memory model gives a comparable performance gain over the SC memory model only when the synchronization rate is reasonably low.

At a high synchronization rate and high multithreading degree, the WC memory model gives a worse performance than the PC or the SC memory models, due to the constraints on synchronization accesses, as shown in Fig. 11b. The performance of the PC model depends very much on the buffer size used. With a small buffer size and a high multithreading degree, the PC memory model performs worst than the SC memory model due to excessive processor stall time, as shown in Fig. 11c. Overall, the performance gain through multithreading is more than 50% of the total improvement, higher than 20% to 40% contributed by relaxed consistency in the shared memory.

VII. Caching Effects and Comparison With Simulation Results

In all the models considered, we have assumed that the miss ratio for shared data is constant under different multithreading degree. This is an optimistic assumption since some cache interferences may occur between data shared by different threads. A better approximation is to consider the cache miss increase due to cache interference when multiple contexts are running on a processor. A higher cache miss ratio will reduce the effective run length, which in turn will lower the processor efficiency.

We use the cache model derived in [24] to estimate the *effective cache miss ratio* due to multithreading. Assuming that

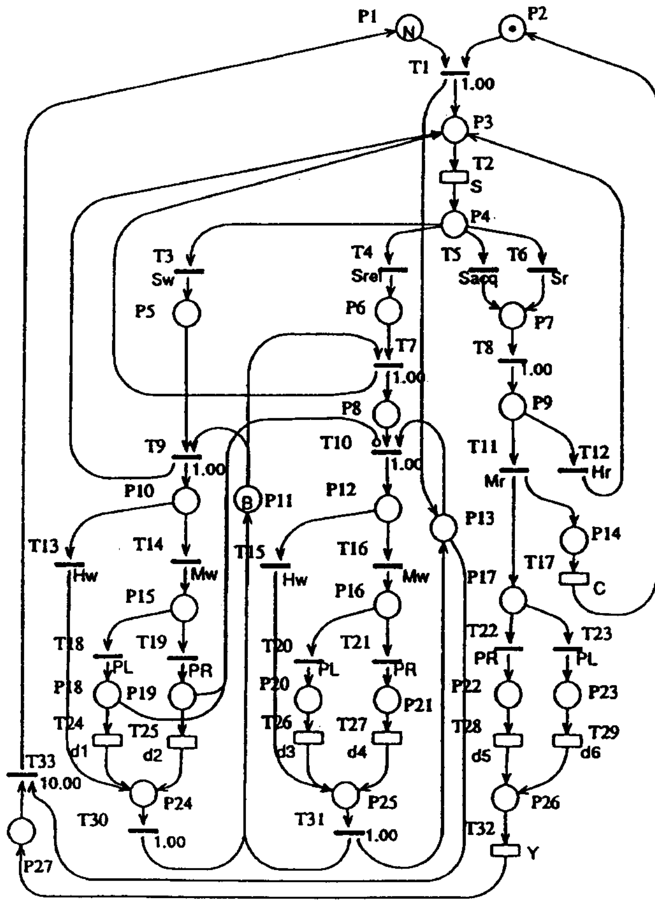


Fig. 10. The GSPN model for the release-consistency shared memory.

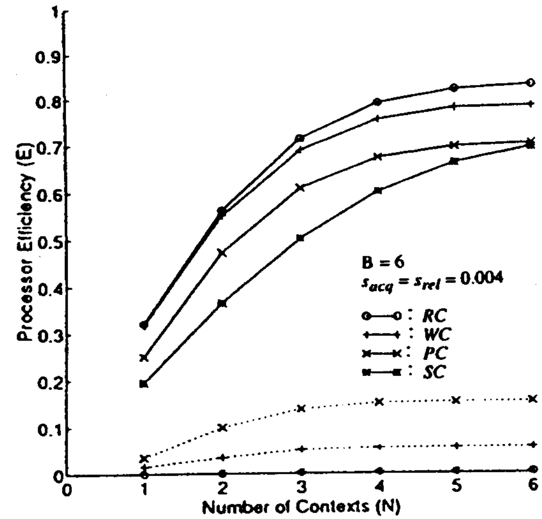
the fixed component of cache interference is negligible and the same physical cache is shared equally among all contexts, the miss ratio for N contexts, $m(N)$, is expressed by:

$$m(N)=\begin{bmatrix} m(1)N^{k}, & \text{if } N\leq\left\lfloor (m(1))^{-1/k} \right\rfloor \\ 1, & \text{otherwise} \end{bmatrix} \tag{7}$$

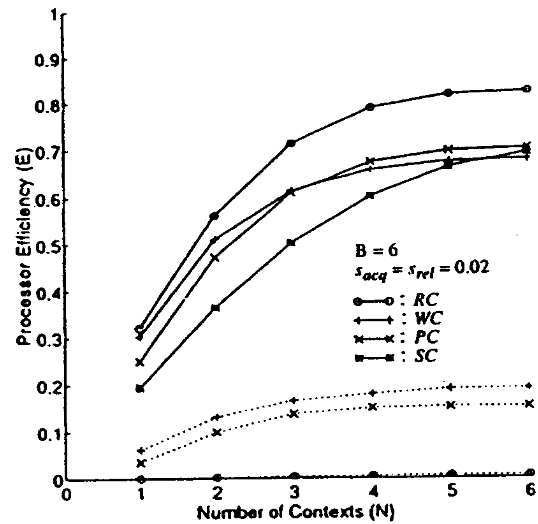
where $m(1)$ represents the single context miss ratio and k is the *cache degradation constant*, which is a positive number determined by the workload. For $k \approx 1$ and a small N, the model is close to a linear relationship between the $m(N)$ and the number of contexts [4].

To apply the Saavedra cache model on our GSPN models, we need to replace $m(N)$ by $m_r(N)$ and $m_w(N)$, representing different miss ratios for read and write accesses, respectively. Similarly, k is replaced with k_r and k_w for reads and writes, respectively. The performance curves for each memory consistency model is obtained by substituting m_r and m_w with $m_r(N)$ and $m_w(N)$, respectively. Fig. 12 shows the effects of cache degradation on the GSPN models for $k_r = k_w = 0.4$, using the default parameter values.

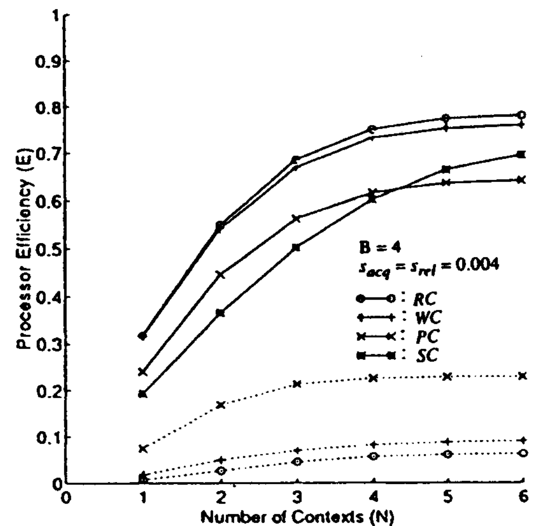
The dotted line corresponds to the performance when the cache interference due to multiple contexts is ignored. As expected, a drop in performance is observed for all memory consistency models due to a higher miss ratio as N increases. The performance gain of any relaxed memory



(a) Synchronization rate $s_{acq} = s_{rel} = 0.004$ and buffer size $B = 6$



(b) Synchronization rate $s_{acq} = s_{rel} = 0.02$ and buffer size $B = 6$



(c) Synchronization rate $s_{acq} = s_{rel} = 0.004$ and buffer size $B = 4$

Fig. 11. Relative performance of various memory consistency models under different synchronization rates and write buffer sizes.

models, i.e., the PC, the WC, or the RC model, over the SC memory model is still appreciative, even under cache degradation. This is especially true for the PC memory model where the processor stalling becomes less severe under a higher cache miss ratio.

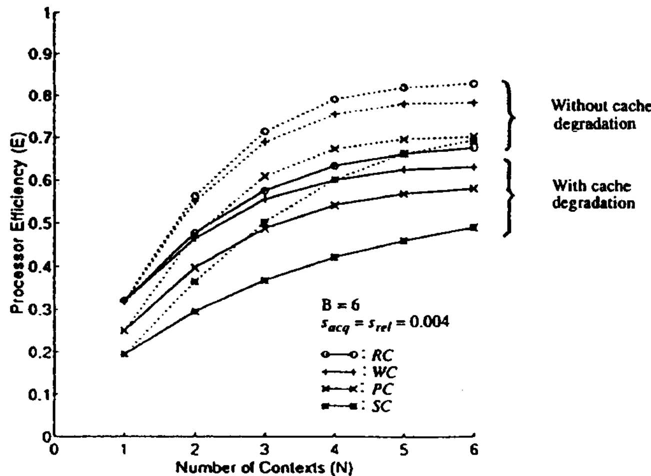


Fig. 12. Effects of cache degradation on the performance of four memory consistency models.

We compare the performance predicted by our GSPN models against the event-driven simulation results reported by Gupta et al. [14] on the Stanford's DASH multiprocessor system. The DASH architecture resembles closely with our multiprocessor model, with the exception that multiple hardware contexts were not built in the prototype construction. The benchmark applications used in the simulation are the MP3D, LU, and the PTHOR codes. The MP3D is a 3-dimensional particle simulator, the LU performs LU-decomposition of dense matrices and PTHOR is a parallel logic simulator.

Table II lists important parameters that were used for our GSPN models, extracted from simulation data reported from the benchmark applications. The memory access latencies L_l and L_r assumed 18 and 73 cycles, respectively, with a cache fill overhead $Y = 8$ cycles. The value of L_r is approximated by taking the average of the latencies between the home and remote nodes. A fixed context switch overhead of $C = 4$ cycles was used. The values of P_l were estimated with the data allocation scheme and accessing pattern of each application code.

TABLE II
PARAMETER VALUES OBTAINED FOR THE MP3D, LU, AND PTHOR CODES

Program	s	s_r	s_w	s_{acq}, s_{rel}	$m_r(1)$	$m_w(1)$	p_l
MP3D	0.295	0.688	0.312	0.0	0.2	0.25	0.4
LU	0.297	0.6697	0.3295	0.0004	0.34	0.03	0.857
PTHOR	0.23	0.8617	0.1037	0.0173	0.23	0.53	0.0625

Note: $L_l = 18$, $L_r = 73$, $Y = 8$, $C = 4$.

For the MP3D code, particles assigned to a processor are allocated from local memory of that processor, but the space cells are distributed evenly among all processing nodes. From data provided, 34% and 50% of the misses are from particle and space cell data structures, respectively, thus giving $P_l \approx 0.4$.

For the LU code, columns are statically assigned to processors on an interleaved fashion, which are allocated from local memories. The ratio between local and remote memory references was estimated by:

$$\frac{p_l}{p_r} \approx \frac{N_c}{2} \tag{8}$$

where N_c is the number of columns assigned to a node. For a 200 × 200 matrix divided among 16 processors, $P_l / P_r \approx 6$, giving an estimated $P_l \approx 0.857$.

For the PTHOR code, the logic elements are evenly distributed among all nodes. We assume equal probability for each logic element to be activated at any point of time. Thus for 16 processors, $P_l = 1 / 16 = 0.0625$. Based on the values of P_l estimated, the effective memory latency is approximated by:

$$L_{eff} \approx p_l L_l + p_r L_r \tag{9}$$

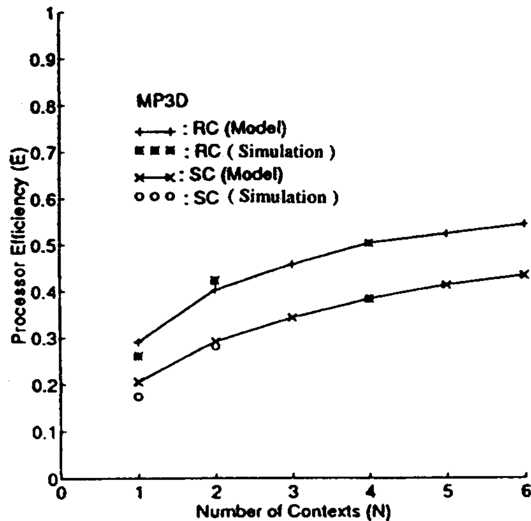
which gives $L_{eff} \approx 50$, 25, and 70 cycles for the MP3D, LU, and PTHOR codes, respectively. This is close to the 50, 20–27, and 60–80 cycles reported from Stanford results.

The cache degradation constants for the LU code are approximated using the reported miss ratios, giving $k_w \approx 4.4$ and $k_r \approx 0.4$, respectively. Since the miss ratios for $N \geq 2$ are not available for the MP3D and PTHOR codes, we have estimated the degradation constants, under the assumption $k_r = k_w$. The values obtained are $k_w = k_r \approx 0.6$ for the MP3D code and $k_w = k_r \approx 0.55$ for the PTHOR code.

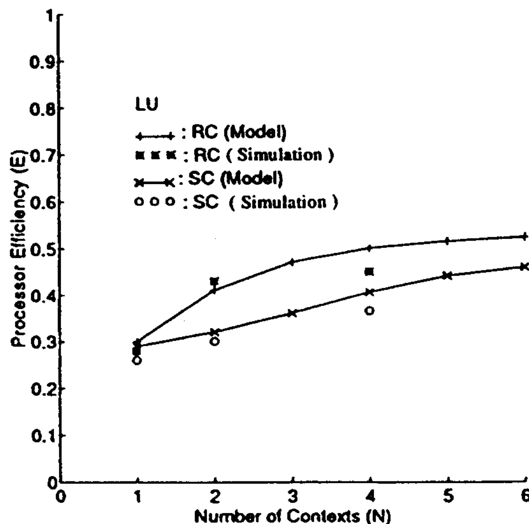
Fig. 13 shows the processor performance of the three applications under the SC and the RC memory models. The continuous lines represent the predicted results using our GSPN models, while the isolated dots represent values reported from Stanford simulation. To limit the number of states generated, we used a buffer size of $B = 6$ for the RC memory model, instead of $B = 16$ as used in Stanford simulation. This is sufficient since the processor stall time due to buffer full condition is negligible in all three relaxed memory models.

For the LU code, the predicted performance and the empirical results correlate well with each other using the SC and RC memory models. The predicted results are generally slightly higher than that reported by Stanford. For the MP3D and PTHOR codes, the predicted performances are very close to the simulation results using single thread processor. Although the predicted performance for $N \geq 2$ seems questionable, we expect the actual values to be close to the predicted results. Possible sources of error are identified below to account for the discrepancies between model and simulation in Fig. 13:

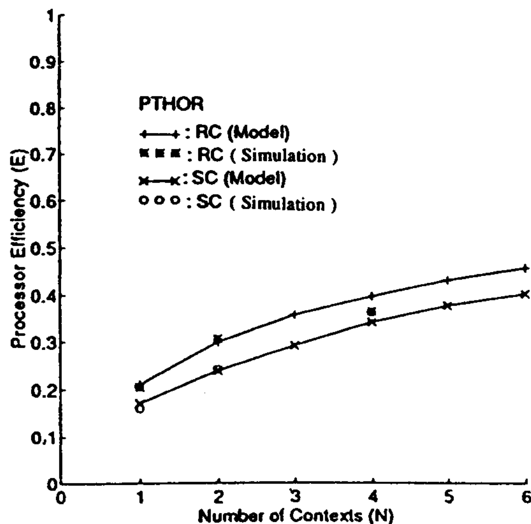
1) In the simulation experiments on the DASH, the processor is blocked from accessing the cache for 4 cycles while a cache fill operation of another context completes. We did not include this blocking effect in our models. Instead, we considered only the overall cache fill overhead.



(a) MP3D with cache degradations $k_r = k_w = 0.6$



(b) LU with cache degradations $k_r = 0.4$ and $k_w = 4.4$



(c) PTHOR with cache degradations $k_r = k_w = 0.55$

Fig. 13. The SC and RC memory performance under various cache degradation degrees, as compared with the Stanford simulation results shown by isolated dots.

1) A write hit took two cycles to complete, as compared to a single cycle used in our model.
2) Network and memory contentions due to limited network bandwidth were not considered in our model.
3) Barriers and spinning on synchronization locks causing additional waiting time were neglected in our model.

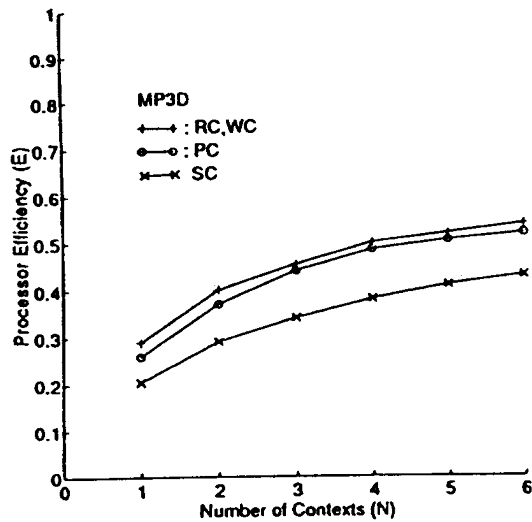
To summarize, Fig. 14 shows the performance curves under the SC, PC, WC, and RC shared memory models, using the estimated cache degradation constants obtained earlier. For the MP3D code, the WC and RC models gives the same performance since the synchronization rate is negligible. The PC model performs comparably well with some lost in performance due to limited buffer size. For the LU code, all three relaxed consistency models give nearly the same performance since the problems of limited buffer size and high synchronization rate does not exists in this program.

For the PTHOR code, the PC and RC memory models performs equally well since the buffer size problem does not arise due to a low write access rate. The WC memory model gives a considerably lower performance due to a higher synchronization rate, with a 0.15 processor idle probability for $N = 4$. In general, all three relaxed memory consistency models provide appreciable performance gains over the SC memory model for $N = 2, 3,$ or 4.

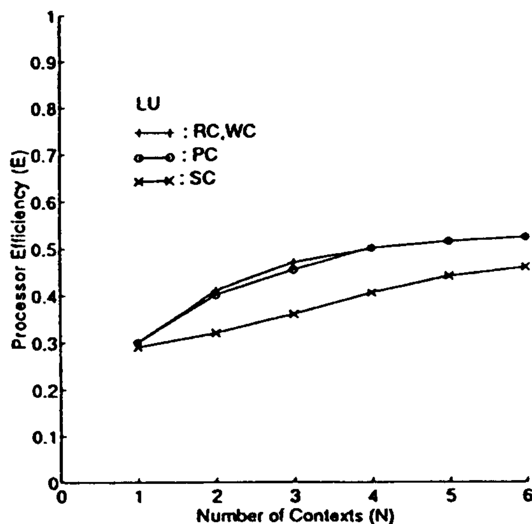
VIII. Conclusions

In the past, various relaxed consistency memory models were evaluated to have performance gains over the sequential-consistency model for mostly single-threaded processors. The effects of using relaxed consistency memory models on multithreaded processors are shown more sensitive to machine architecture and program behavior. The amount of processor stalling time due to write buffer being full or due to access constraints imposed by a specific consistency model can severely degrade the processor efficiency. We have presented four GSPN models for multithreaded processors under the SC, PC, WC, and RC memory consistency models, respectively. These models are based on the architectural assumptions that reads are blocking and context switching takes place on a cache miss. The effects of different buffer sizes and synchronization rates are revealed.

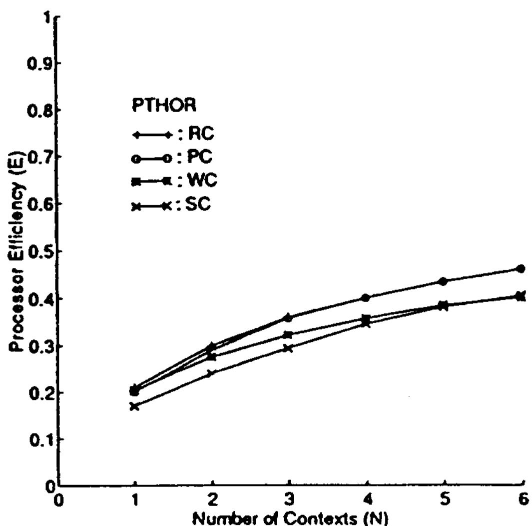
Based on the results presented, the RC memory model gives the best performance by hiding write latency completely with a reasonably small buffer size. The performance of the PC memory model depends on the buffer size used and on the effective service rate of the write buffer. For a small buffer size or a higher write access rate, the PC memory model may perform worse than the SC memory model due to excessive processor stalling time on buffer being full. Given a sufficient buffer size and a low write access rate, the PC model may outperform the WC memory model in applications with a higher synchronization rate. This was proved in the PTHOR code, where no synchronization constraint is imposed on the PC memory model used.



(a) MP3D



(b) LU



(c) PTHOR

Fig. 14. Relative performance of four shared memory consistency models using default parameters from Stanford benchmark suite.

The performance of the WC memory model is primarily limited by the synchronization rate in the application code. For a low synchronization rate, the WC memory model performs as well as the RC memory model in hiding write latencies. For a high synchronization rate, the performance is limited by the processor stalling for synchronization accesses, which have to be sequentially consistent. Given that most applications contains relatively a low synchronization rate, such as the MP3D and the LU codes, the WC memory model performs almost equally with the RC memory model. In general, with a low synchronization rate and a sufficiently large buffer size, all three relaxed consistency models provide comparable performance gains with minor differences over the SC memory model.

We have verified our analytical memory models by comparing the processor efficiency predicted by the GSPN models with the simulation results from Stanford University. For the SC and RC memory models, the GSPN models correlate well with the Stanford simulation results, while the PC and WC memory models require further validation due to lack of simulation results to compare with. In addition, both analytical and simulation results clearly show the dominating performance gain from multithreading, irrespective of the type of memory consistency model used. Our results are also qualitatively in agreement with that reported in [29], where relaxed consistency model shows good performance gain on single-threaded processors with write-back caches.

Some architectural assumptions on our GSPN models were simplified in order to limit the complexity of the models. These GSPN models can be extended to include the effects of context switching overhead, network contention, synchronization overhead, and prefetching operations. The remote memory access latency could be modified to reflect better the latency variations of a specific interconnection network. Previous studies reported in [4] and [26] have shown that network contention reduces the overall gains from supporting multiple outstanding requests in the network. Hence we expect slight drops in performance of all memory consistency models once these negative effects are included. To include the effects of synchronization overhead, one could consider using the *always-spin, always-switch-spin, always-block,* or the *two-phase* waiting algorithms suggested in [19]. Other possible extensions to the GSPN memory models include the use of cycle-by-cycle switching policy for finely multithreaded processors. One can also modify the study to use write-back caches which may further increase the complexity of the GSPN models to be used.

ACKNOWLEDGMENT

This work was supported by Nanyang Technological University of Singapore, while Chong studied at the University of Southern California in Los Angeles.

REFERENCES

[1] S.V. Adve and M.D. Hill, "A unified formalization of four shared-memory models," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 6, pp. 613-624, June 1993.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera computer system," *Proc. ACM Int'l Conf. on Supercomputing*, pp. 1-6, June 1990.

[3] A. Agarwal et al., "The MIT Alewife machine: A large-scale distributed-memory multiprocessor," Dubois and Shreekant, eds., *Scalable Shared Memory Multiprocessors*. Boston, Mass.: Kluwer Academic Publishers, 1992.

[4] A. Agarwal, "Performance tradeoffs in multithreaded processors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 3, no. 5, pp. 525-539, Sept. 1992.

[5] M. Ajmone et al. "A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems," *ACM Trans. on Comp. Sys.*, vol. 2, no. 2, pp. 93-122, May 1984.

[6] G. Bell, "Ultracomputers—a teraflop before its time," *Comm. of the ACM*, vol. 35, no. 8, pp. 27-47, Aug. 1992.

[7] G. Chiola, "GreatSPN user manual version 1.3," Technical report, Dipartimento di Informatica, Universita di Torino, Torino, Italy, Sept. 1987.

[8] Y.K. Chong, "Effects of memory consistency models on multithreaded multiprocessor performance," MSc thesis, University of Southern California, May 1993.

[9] Convex Computer, Inc., *The Examplar Architecture*. Richardson, Tex.: Convex Press, 1993.

[10] Cray Res. Inc., *The Cray T3D System Architecture Overview*. Madison, Wis.: Cray Res. Inc., 1993.

[11] M. Dubois and C. Scheurich, "Memory access dependencies in shared-memory multiprocessors," *IEEE Transactions on Software Engineering*, vol. 16, no. 6, pp. 660-673, June 1990.

[12] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," *Proc. 13th Int'l Symp. Comp. Arch.*, pp. 434-442, June 1986.

[13] K. Gharachorloo, A. Gupta, and J. Hennessy, "Performance evaluation of memory consistency models for shared-memory multiprocessors," *Proc. Fourth Int'l Conf. on Arch. Support and Prog. Lang. and O.S.*, Apr. 1991.

[14] A. Gupta, J. Henessy, K. Gharachorloo, T. Mowry, and W.D. Weber, "Comparative evaluation of latency reducing and tolerating techniques," *Proc. Int'l Symp. Computer Architecture*, pp. 254-263, May 1991.

[15] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, and J.Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *Proc. 17th Symp. Comp. Arch.*, pp. 15-26, May 1990.

[16] J.R. Goodman, "Cache consistency and sequential consistency," Technical Report 61, IEEE SCI Committee, 1989.

[17] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. New York: McGraw-Hill, 1993.

[18] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. 28, no. 9, pp. 241-248, Sept. 1979.

[19] B.H. Lim and A. Agarwal, "Waiting algorithms for synchronization in large-scale multiprocessors," *ACM Trans. Comp. Sys.*, vol. 11, no. 3, Aug. 1993.

[20] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comp. Sys.*, vol. 7, no. 4, Nov. 1989.

[21] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *J. of Parallel and Distributed Computing*, vol. 12, no. 2, pp. 87-106, June 1991.

[22] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam, "The Stanford Dash multiprocessor," *IEEE Computer*, pp. 63-79, Mar. 1992.

[23] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," *IEEE Computer*, Aug. 1991.

[24] R.H. Saavedra, D.E. Culler, and T. von Eicken, "Analysis of multithreaded architecture for parallel computing," *Proc. Second ACM Symp. Par. Algo. and Architecture*, July 1990.

[25] R.H. Saavedra and D.E. Culler, "An analytical solution for a Markov chain modeling multithreaded execution," Technical Report UCB/CSD-91/623, Computer Science Division, University of California at Berkeley, Mar. 1991.

[26] R.H. Saavedra, W. Mao, and K. Hwang, "Performance and optimization of data prefetching strategies in scalable multiprocessors," *J. of Parallel and Distributed Computing*, pp. 427-448, Sept. 1994.

[27] Thinking Machines Corp., *The CM-5 Technical Summary*. Cambridge, Mass.: Thinking Machines Corp., 1991.

[28] W.D. Weber and A. Gupta, "Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results," *Proc. 16th Int'l Symp. on Comp. Arch.*, pp. 273-280, June 1989.

[29] R.N. Zucker and J.L. Baer, "A performance study of memory consistency models," *Proc. 19th Int'l Symp. Comp. Arch.*, pp. 2-12, May 1992.

Yong-Kim Chong received the BEng degree in electrical engineering from the National University of Singapore in 1986 and the MS degree in computer engineering from the University of Southern California in 1993. He is currently a lecturer in the School of Electrical and Electronic Engineering at the Nanyang Technological University of Singapore. His research interests are in computer architecture, parallel processing, and scalable multiprocessors with distributed shared memory.

Kai Hwang is a professor of electrical engineering and computer sucience at the University of Southern California. He received the PhD degree from the University of California at Berkeley. An IEEE Fellow, he has served as a Distinguished Visitor of the Computer Society and on the ACM SIGARCH Board of Directors and is the founding Editor-in-Chief of the *Journal of Parallel and Distributed Computing*. He has published over 130 scientific papers and five books, most of which are related to computer architecture and parallel processing. His present research interests focus on scalable multiprocessors with distributed shared memory, software support for parallel programming, and embedded applications in multimedia, telecommunications, and distributed computing. Presently he leads the University of Southern California Spark group in performing STAP benchmark experiments on the IBM SP2, Intel Paragon, and Cray T3D.