Performance Analysis of Four Memory Consistency Models for Multithreaded Multiprocessors *

Yong-Kim Chong and **Kai Hwang**

Technical Report No. 93 - 17
May 1993

* This paper was submitted to the *IEEE Transaction on Parallel and Distributed System* for consideration of publication. All rights reserved by the coauthors. Chong was supported by Nanyang Technological University of Singapore, while this work was carried out at USC.

Performance Analysis of Four Memory Consistency Models for Multithreaded Multiprocessors [1]

Yong-Kim Chong and **Kai Hwang**
Dept. of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089 - 2562

Abstract The performance of four shared memory models is analyzed for scalable multiprocessors using multithreaded processors as building blocks. The memory models being evaluated include the *sequential consistency* (SC) model by Lamport 1979, *weak consistency* (WC) model by Dubois et al. 1986, *processor consistency* (PC) model by Goodman 1989, and *release consistency* (RC) model by Gharachorloo et al. 1990. We use a stochastic approach to modeling the shared memory behavior. Specially specified stochastic timed Petri nets are developed for different memory consistency models. The embedded Markov chains are solved to obtain the processor performance curves under various memory-access constraints.

Our analytical results verify the believe that the SC model gives the lowest performance. The PC model requires to use larger write buffers, while the WC and RC models require smaller write buffers. The PC may perform even worse than the SC if small buffer is used. The performance of the WC model depends heavily on the synchronization rate in user code. For low synchronization rate, WC may perform as well as the RC model. Overall, the RC memory model gives the best performance. We also discuss the caching effects and compare the relative performance under different degrees of buffering and pipelining of remote memory accesses. The analytical results being reported correlate well with the trace-driven simulation results reported from Stanford University. The proposed Petri-net models can be also extended to study the effects of other latency tolerating mechanisms in scalable multiprocessors.

1. All rights reserved by the coauthors. Manuscript submitted to *IEEE Trans. on Parallel and Distributed System*, May 1993. This paper was also released as *Tech. Report No. 93 - 17*, Dept. of EE - systems, USC, Los Angeles, CA. May 1993. Chong was supported by Nanyang Technological University of Singapore, while studying at USC. For all future correspondence, please contact Kai Hwang at USC.

1. Introduction

The current trend in designing massively parallel computers is to build a scalable, distributed shared-memory system with thousands of processing nodes [LH89, NL91, Bell92, Hwa93]. This is done by forming a single global address space by combining the address space of each processing node. The main advantage of such a system is that it provides the *scalability* of a distributed-memory multicomputer, while keeping the *programmability* of a conventional shared-memory multiprocessor system. Some representative systems are the Stanford DASH [LLGx92], the Tera computer [ACCx90] and the CM5 [TMC91], etc.

One of the main challenge in building a scalable system is the ability to hide the long memory access latencies. Various latency hiding techniques [Hwa93] include the *coherent caches, relaxed consistency models, prefetching and multiple contexts* or *multithreading* suggested by various researchers. Simulation studies on a combination or some of these techniques were shown to be promising [WG89, GGH91, GHGx91]. Several analytical models were also proposed to model multithreaded processors with and without other latency hiding schemes [Saav90-91, Agar92, MH93].

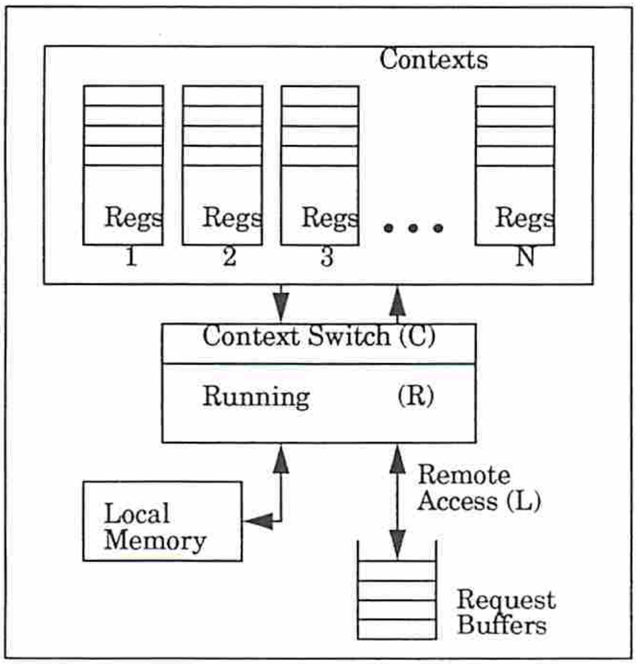
The primary objective of this paper is to develop several stochastic models of a multithreaded processor equipped with hardware coherent cache and supported by different memory consistency models. Memory consistency model defines the order by which the shared memory accesses from one process should be observed by other processes in the system. It imposes restrictions on the order of shared memory accesses initiated by each processor. This translate to the amount of buffering and pipelining of memory accesses allowed in each processor in trying to hide the latency. The consistency models that will be considered are the *Sequential Consistency* (SC) [Lam79], *Processor Consistency* (PC) [Good89], *Weak Consistency* (WC) [DSB86] and *Release Consistency* (RC) [GLLx90] memory models. The *Generalized Stochastic Petri Net* (GSPN) [Ajmo84] is used based on its capability in modeling both concurrency and synchronization operations, which is not possible under queuing models. The effects of some parameter changes on the relative performance of each model will also be studied.

This paper is organized into nine sections. Section 2 presents a basic GSPN model for the abstract multithreaded processor. In Section 3, the basic model is extended to model a multithreaded processor with coherent cache, under the SC consistency memory model. The architectural assumptions and parameters used for the model are defined. In Sections 4 - 6, the GSPN model is further extended to model three relaxed consistency memory models, namely the PC, WC and RC, with different degrees of memory access constraints. The performance curves obtained for four consistency models are compared under different parameter values. Section 7 includes a simple cache degradation model to improve the accuracy of the GSPN models.The performance curves predicted by the GSPN models are compared with the simulation results from Stanford University. This is followed in Section 8 by a summary of the relative merits of various memory models. Finally, we conclude on the research contributions and comment on further work needed.

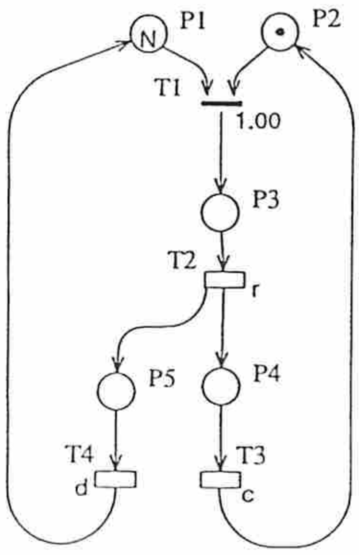
2. Petri Net Model of Multithreaded Processors

A multithreaded processor in a multiprocessor system can be described by the abstract model shown in Figure 1a. A total of N threads or contexts are running in each processor. Each context runs for R cycles before making a remote memory access, where it will be switched out and being replaced by another ready-to-run context. The context switch overhead is C cycles and each remote memory access operation takes L cycles to complete.

A number of assumptions were made in this abstract model. The synchronization accesses which are program dependent are ignored. Pipelining of remote memory access is allowed, with multiple servers in the network, thus allowing multiple outstanding requests being service simul-



(a) Abstract model



(b) The GSPN model

Figure 1: A multithreaded processor model

taneously. L is assumed much larger than C since it defeats the purpose of multithreading if L is equal or smaller than C. Finally, we also assume that N remains constant throughout the computation, which means that there is sufficient parallelism for execution within each processor.

The performance of a multithreaded processor is measured by an efficiency figure, given as the ratio of the busy time that the processor is executing useful instructions to the total time elapsed. The aim is to keep the processor busy with sufficient contexts, without having to wait for the completion of the remote memory accesses.

The GSPN model is constructed from the abstract processor model, as shown in Figure 1b. This is possible if we assume that all the parameters R, C and L to be random variables having exponential distribution. Hence the rate or probability of a remote memory request (r) is given by $1/R$, which is the inverse of the average run-length. Similarly, the service rate for performing context switch (c) is $1/C$ and the service rate for remote memory accesses (l) is $1/L$.

The assumption that R is a random variable is valid, since we would expect some variations in the actual run-length of each thread. This also applies to the parameter L since the latency is affected by the distance traversed and the network traffic density. For the case of context switch overhead, C, the assumption was made mainly to reduce the total number of possible states, with minimal effect on the accuracy of the model for small values of C. This is vital in determining the feasibility of applying numerical solution method on the large system of equations obtained. This assumption does not affect the accuracy of our model, compared with the analytical model used in [Saav90].

A token in place P1 indicates that a ready-to-run context is available for next switching. P1 is initialized with N tokens which represents the N contexts ready for execution at the start time. A token present or absent in place P2 indicate the availability of the processor. P2 is initialized with one (1) token to represent the single processing unit 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 that there are outstanding remote memory requests being serviced. Notice that P4 and P5 are in parallel to signify concurrent operations of context switching and servicing of remote memory accesses.

Transition T1 is an immediate transition with a probability of one (1). It controls the access of the processor for the next ready-to-run context, by serving as an access gate. This transition will be 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 ensures that the processor is either in the *Running* state or in the *Context Switching* state, but not in both. Transitions T2 and

T3 represent the rate of remote memory access (r) and service rate for context switching (c) respectively. Transition T4 represents the service rate of remote memory accesses. The service rate for T4 is *marking dependent*, given by

$$d = M_5 \times l \qquad (1)$$

where M_5 is the number of tokens present in place P5 at any instant of time. This is valid since we assume that the remote memory accesses are pipelined, or having multiple independent servers on the network, thus allowing multiple outstanding requests being serviced concurrently. Hence for m outstanding requests, the expected time to receive a reply is L/m, as compared to L in the case of a single outstanding request. This translate into an increase in the service rate by a factor of m.

The associated *Embedded Markov Chain* (EMC) of the GSPN model can be easily obtained from the reachability tree by considering only the set of states that the process enters after each transition. This includes both the *vanishing* and *tangible* states. The number of states generated under GSPN is much smaller as compared to the associated PN model due to the precedence rule introduced by the immediate transition T1. The number of states can be further reduced to that consisting of *tangible* states only by replacing all vanishing states with the appropriate transition probability [Ajmo84]. The reduced EMC will be used in constructing the transition probability matrix and the steady state solution for the GSPN model.

The corresponding EMC and reduced EMC of the GSPN model for the case of N=2 is as shown in Figures 2a and 2b respectively. The states are represented by 5-tuple $\{M_1\, M_2\, M_3\, M_4\, M_5\}$ which indicate the number of tokens in each place for a particular state. S_i and V_i represent *tangible* and *vanishing* states respectively. The total number of states in the reduced EMC for N contexts, S_N is given by

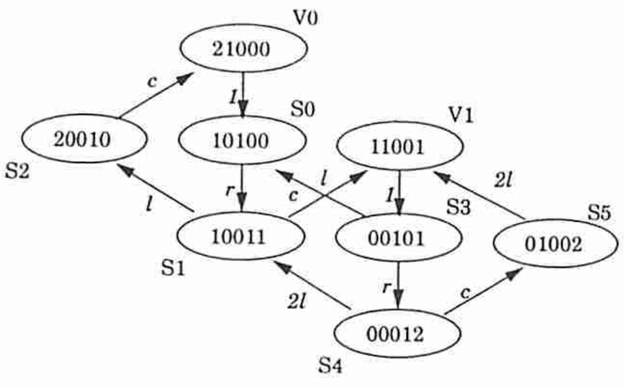
$$S_N = 2\,(N+1) \qquad (2)$$

which is linearly proportional to N, and is independent of the parameters R, C and L.

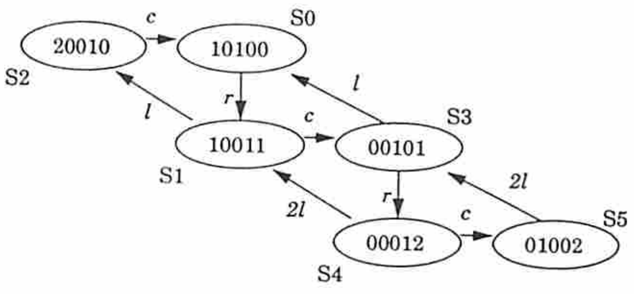
From the reduced EMC state transition rate diagram, the steady state solution of the Markov chain can be obtained by first solving the following system of linear equations:

$$\pi = \pi\ U \qquad (3)$$

where U is the transition probability matrix and π is the vector of the stationary probability distribution of the EMC. The element u_{ij} of matrix U is given by the transition probability from state i to state j of the EMC.



(a) Embedded Markov chain



(b) Reduced embedded Markov chain

Figure 2: State transition diagrams of the GSPN model for $N = 2$

The steady state probability, π_j, of state j can be obtained by first arbitrary taking any state, i, as a reference state. Compute the mean number of visits to state j between two subsequent visit to state i (V_{ji}), the average sojourn time in state j ($E[W_j]$) and the mean cycle time of state i (C_i). The steady state probability is given by the equation

$$\pi_j = \frac{V_{ji}\ E[W_j]}{C_i} \tag{4}$$

Again, for the case of N=2, the transition probability matrix for the GSPN model is

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

For specific values of the parameters r, c and l, numerical solution method such as Gaussian Elimination can be applied to solve the system of linear equations. The processor efficiency with N contexts, E_N, can be defined as *the probability of having a token in place P3 at steady state*, given by the equation:

$$E_N = \sum_{j \in G} \pi_j \qquad (5)$$

where G= {*set of tangible states with a token in P3*}. Thus the processor efficiency for N=2, E_2, can be obtained by adding the steady state probabilities of states S0 and S3,

$$E_2 = \pi_0 + \pi_3$$

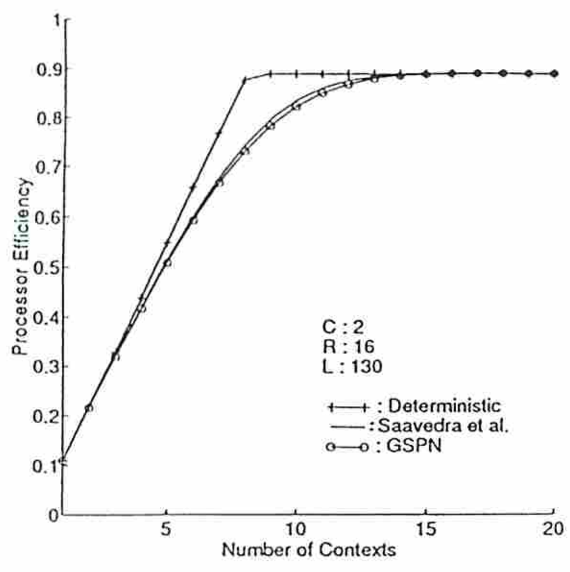
With N=2, R=16, C=2 and L=130, we obtain π_0=0.024087 and π_3=0.192036, which gives E_2=0.21612.

The performance or processor efficiency curves of the multuthreaded processor under various degree of multithreading ($N \geq 1$) can be obtained for specific values of R, C and L. However, for large value of N, it is only feasible to extract the reachability tree and solving the large system of linear equations with the help of available software tools. GreatSPN [Chi87] is one of the tools that was developed by Chiola et al. to assist researchers in analyzing GSPN models with reasonably large state space. It provides graphical input facilities for the building of GSPN model, extracting the reachability graph and performing steady state solution using Gauss-Seidel iterative solution of the embedded Markov Chain. We have used this tool in building and performing steady state solution to all our GSPN models.

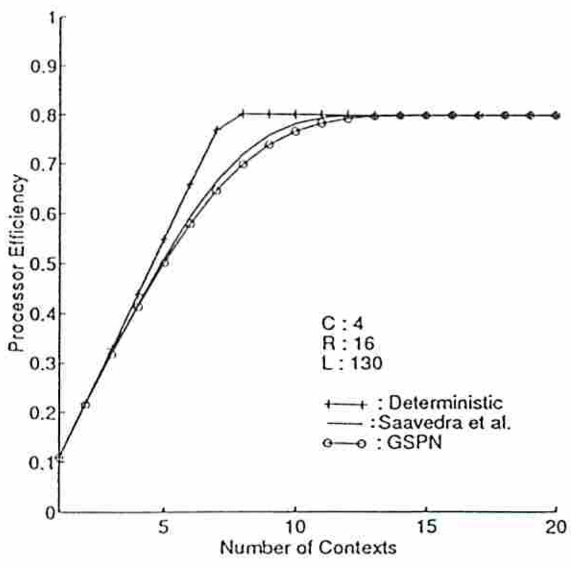
The processor efficiencies based on GSPN model of Figure 1b for N=1 to 20 are computed for C=2 and 4 cycles, using fixed values of R=16 and L=130 cycles. In order to verify the accuracy of the GSPN model, the processor efficiency curves are compared with the analytical solution obtained by [Saav90-91], as shown in Figure 3. For both cases, the two curves are very close to one another, and practically overlaps for small values of N. The slight difference around the *knee* of both curves is as expected since the GSPN model assumes stochastic nature for both L and C, which are taken as deterministic values in [Saav90-91].

3. Effects of Sequential Consistency Memory Model

We perform our analysis on the performance of a multiprocessor system under Sequential Consistency (SC) memory model, with multithreaded processor as building blocks. The system consists of P processing nodes connected through a scalable interconnection network. Physical memory is distributed among all processing nodes, forming a single shared virtual memory space.



(a) Context switch overhead $C = 2$



(b) Context switch overhead $C = 4$

Figure 3: Processor performance curves for R=16 and L=130

Cache coherence is maintained using hardware-supported, invalidating, distributed directory-based protocol [LLGx90].

We will assume that all instructions and private data are stored locally in each node, with a cache hit rate close to one. This allows us to assume that all references take single processor cycle to complete, except for shared-data references. The caches are assumed to be *lock-up free* using write through scheme, with sufficiently large *continuation name space* to cater for multiple outstanding memory requests from different threads. A memory request may be satisfied locally, either from the cache or from local memory module, or from a remote memory module of another processing node.

All reads are assumed to be *blocking*, where the processor continue execution of the same thread only after the value requested has been received. Thus no multiple outstanding read requests are allowed for a single thread. Independent requests from different threads may be pipelined, thus allowing concurrent accesses to both the local and remote memories. The pipeline stages are assume to be sufficiently large to cater for all pending independent accesses from different threads. A cache fill operation is performed following the reply from a read request without stopping the processor's execution of another thread. All synchronization variables for *acquire* *(lock)* and *release (unlock)* operations are assumed cachable, and take same amount of time as normal read-write operations. We will ignore the synchronization barriers in our model.

Based on the memory access constraints imposed on SC and the sufficient conditions provided by Dubois et al. [DSB86], we define the multithreaded processor under the SC as follows:

A. For each thread:

   a. *Read* : Processor issues read access and waits for read to perform. Context switch on cache miss.

   b. *Write* : Processor issues write access and waits for write to perform. Context switch on 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* when it is a *hit* in the cache, or a reply is received from either local or remote memory module. Although some buffering of write accesses within each thread is possible, the potential gain is rather limited. This is due to the fact that in most applications, read and write accesses are well interleaved, thus making most of the write latency visible to the processor [GGH91]. In addition, a more complicated context switching scheme is needed to avoid processor idling while waiting for previous writes to complete.

We will use the GSPN model of a multithreaded processor presented in Section 2 as the basic structure of our model. Assume that the average run length (R), memory access latencies (L_L,L_R), context switch overhead (C) and cache fill overhead (Y) are random variables having exponential distribution, with service rates given by the inverse of the average time consumed. All threads are assumed to be independent, so that accesses from different threads in the same node can be pipelined. Listed below are basic parameters used in our analysis:

N :	Number of contexts running in each processor ($N \geq 2$)
s :	The *shared-memory reference rate*, which is given by the inverse of the average run length between two consecutive shared-memory references ($s = 1 / R$).
s_w, s_{rel} :	The fraction or probability of shared memory references that are *writes* and *releases*, classified under write operations.

s_r , s_{acq}	:	The fraction or probability of shared memory references that are *reads* and *acquires,* classified under read operations.
m_r , m_w	:	The cache miss rates for shared read and write operations. The respective hit rates h_r and h_w can be obtained from $h = 1 - m$.
p_L , p_R	:	The fraction or probability of cache misses being serviced by *Local* or *Remote* memory respectively, with $p_R = 1 - p_L$.
c	:	The service rate in performing context switching, which is the inverse of context switching overhead ($c = 1 / C$).
l_L , l_R	:	The service rates for *Local* or *Remote* shared memory accesses, which is given by the inverse of the respective average access latencies ($l_L = 1 / L_L$, $l_R = 1 / L_R$).
y	:	The service rate in performing cache fill operation, which is given by the inverse of cache fill overhead ($y = 1 / Y$).

The set of parameters s, s_w, s_{rel}, s_{acq} and s_r depends mainly on the application and program behavior. The cache miss rates depend on both program behavior and cache performance. Parameters p_L and p_R are determined by the program behavior and the shared data distribution among all nodes. Parameters c, l_L, l_R and y are determined by the processor and system architecture.

The multithreaded processor under the SC model can be conveniently represented by a GSPN model as shown in Figure 4. Two distinct branches can be identified, representing the read and write operations. The net is described below:

Place P1 contains tokens that represent ready-to-run contexts. It is initialized with N tokens. A token present in place P2 indicate the availability of the processor for next context switch. It is initialized with one (1) token to signify single processing unit. Transition T1 serves as an access gate to the processing unit. A token present in place P3 indicate the processor is in the *Busy* state. A maximum of one (1) token can be present in P3 at any time, as controlled by T1.

The timed transition T2 is assigned with a firing rate s, representing the rate of shared 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 among the shared accesses, with

$$s_w + s_{rel} + s_{acq} + s_r = 1 \qquad (6)$$

Transitions T9, T10, T11 and T12 models the hit and miss rates of the coherent cache. T9 and T10 are assigned with probabilities of h_w and m_w respectively for the write accesses, whereas T12 and T11 are assigned with h_r and m_r respectively for the read accesses. A cache hit will put the processor back in the *Busy* state immediately, with a token in P3.

A cache miss will put the processor in the *Context Switching* state, represented by a token in place P9. Transition T13 fires at a rate of c, to signify the completion of context switching operation. Transitions T14, T15 are assigned with probabilities p_L and p_R, representing the probability of a write request being service by the local or remote memory module. The same applies to T16 and T17 for the read requests.

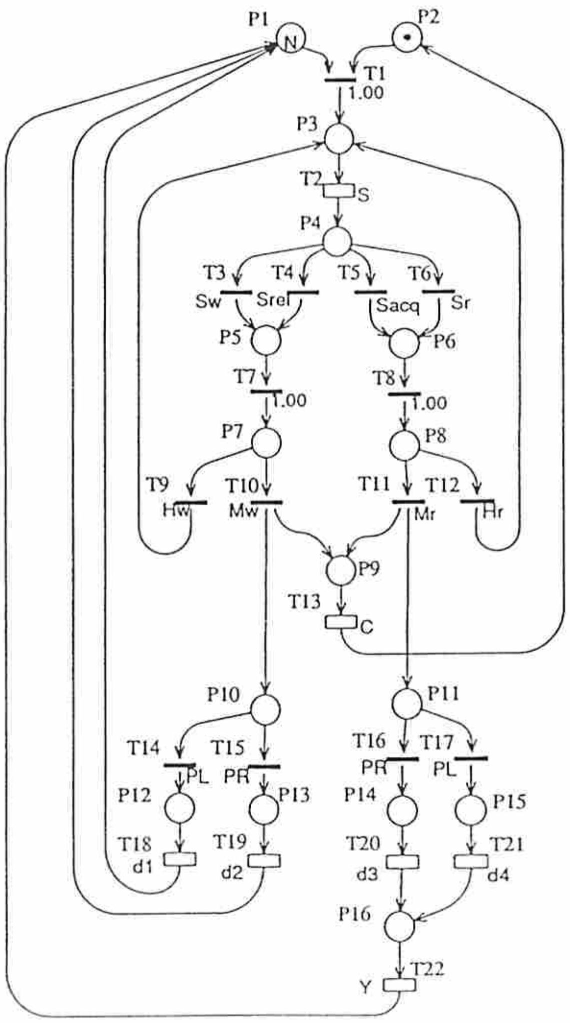


Figure 4: The GSPN model for the Sequential Consistency memory

Tokens present in places P12 and P15 represents pending memory requests for local memory. These requests are serviced at rates determined by firing rates of T18 and T21 which are *marking dependent,* given by:

$$d1 = M_{12} \times l_L$$
$$d4 = M_{15} \times l_L \tag{7}$$

Similarly, tokens present in place P13 and P14 represents pending memory requests for remote memories, with firing rates of T19 and T20 given by:

$$d2 = M_{13} \times l_R$$
$$d3 = M_{14} \times l_R \tag{8}$$

Transition T22 models the cache fill operation, with a firing rate of y. The completion of a read or write access places a token back to P1, adding to the number of ready-to-run contexts for next switching.

A set of default parameters, tabulated in Table 1, have been chosen to evaluate the impact of parameter changes on different memory consistency models. We assume that $s_w + s_{rel} = s_r + s_{acq} = 0.5$, unless stated otherwise. We will limit the number of contexts in our model to $N \le 6$ based on the large number of states generated by the GSPN and the computation time involved in computing the steady state probabilities. Previous studies [Saav90, GHGx91, Agar92] have shown that small number of contexts ($N \le 4$) are sufficient to achieve close to maximum efficiency. The processor efficiency for $N = 1$ is approximated by setting $c = 10.0$, such that $P\{Switching\} \approx 0$. The processor efficiency at saturation can be obtained using deterministic analysis. The effective run length, R_{eff}, is given by:

$$R_{eff} = \left(\frac{1}{(s_w + s_{rel})\, m_w + (s_r + s_{acq})\, m_r}\right)\left(\frac{1}{s}\right) \tag{9}$$

and the processor efficiency at saturation, E^{sat}, is given by:

$$E^{sat} = \frac{R_{eff}}{R_{eff} + \frac{1}{c}} \tag{10}$$

Using the default parameter values, we have $R_{eff} = 10$ cycles and $E^{sat} = 0.7143$.

The contribution of coherent cache can be seen by varying the parameters m_r and m_w as shown in Figure 5a. A lower cache miss rate will increase the effective run length of each thread,

Table 1: Default parameter values for the GSPN models

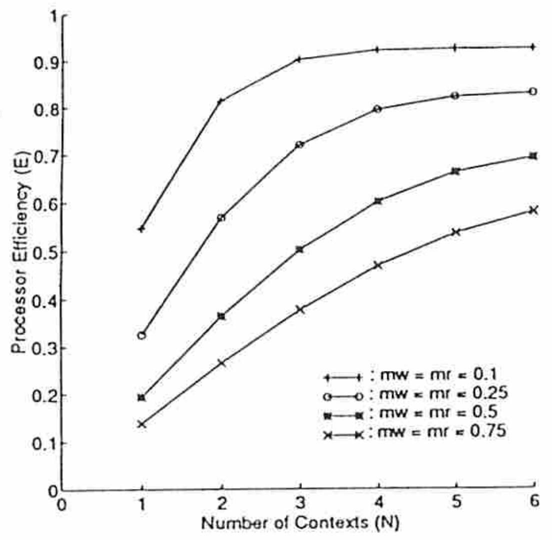
Parameters	Default Val.	Remarks
s	0.2	$R = 5$
s_w , s_r	0.5	
s_{rel} , s_{acq}	0.0	
m_r , m_w	0.5	
p_L , p_R	0.5	
c	0.25	$C = 4$
l_L	0.0625	$L_L = 16$
l_R	0.015625	$L_R = 64$
y	0.5	$Y = 2$

thus improving the processor efficiency in both linear and saturation regions. The case $m_r = m_w = 1$ represents a multiprocessor system without coherent caches, where all shared references results in a miss, followed by a context switch.

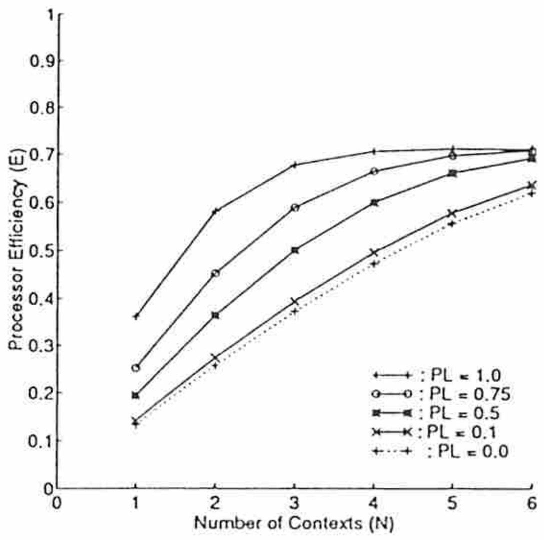
Since no buffering is allowed for write accesses under SC model, no significant change in the processor efficiency is observed by varying s_w. The difference between read and write latencies is the extra cache fill overhead incurred for read accesses, and in most cases L_L , $L_R >> Y$.

For the case of p_L, a higher value of p_L effectively reduces the average latency seen by the processor, thus increasing the processor efficiency in the linear region. This can be achieved by static allocation of frequently used data to local memory of a processor node, with prior knowledge of the access patterns. In the case of totally random access pattern, with equal probability to each node in the system, $p_L = 1 / P$, where P is the number of processor nodes in the system. Figure 5b shows the effect of varying p_L on the performance curve. Notice that the efficiency at saturation is not affected by p_L.

In general, the performance of the multithreaded processor under the SC model is considerably poor for small number of contexts. This is due to its inability in buffering and pipelining the memory accesses within each thread. Coherent cache generally improve the processor's performance by increasing the effective run length of each thread. Better shared data allocation scheme by assigning frequently used data to local memory of each node reduces the effective memory access latency, thus improving overall processor performance.



(a) With different m_w and m_r



(b) With different p_L

Figure 5: The performance curves of the sequential consistency memory model

4. Effects of Processor Consistency Memory Model

The *Processor Consistency* (PC) model introduced by Goodman [Good89] requires that writes issued from a 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. However, pipelining of write accesses within each thread are not allowed. We define the multithreaded processor under the PC model as follows:

A. For each thread:
   a. *Read* : Processor issues read access and waits for read to perform. Context switch on cache miss.
   b. *Write* : Processor sends write to write buffer, stall if buffer is full. The write request is retired from the buffer only after 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.

Notice that no context switching is taking place for write operations, since writes are sent to write buffer. We define a new parameter B, as the size of the write buffer for all threads. Clearly, B should be large enough to avoid processor idling due to buffer full situation.

The main task in modeling the multithreaded processor under the PC model involves accurate modeling 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. Figure 6 shows the GSPN model using a dynamic server allocation scheme [Cho93]. The basic structure of the net is the same as in the case for SC model, with *d1* to *d4* defined as:

$$\begin{aligned} d1 &= M_{15} \times l_L \\ d2 &= M_{16} \times l_R \\ d3 &= M_{17} \times l_R \\ d4 &= M_{18} \times l_L \end{aligned} \tag{11}$$

Transition T7 represents buffering of write request, which fires only if there is an available buffer entry. This puts the processor back into *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 from logical point of view, using an immediate transition T10. This is possible since in general, L_L, $L_R >> 1$.

Place P11 is initialized with B tokens that represents the total available buffer entries for write operations. P10 contains the token(s) that represents 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*. This assumes that all the pending write accesses are from the current *active* threads. A token is added to P10 when a new context starts running to signify additional degree of pipelining allowed. Likewise, a token is removed from P10 when a read request is completed. A high switching rate of *10.0* is assigned to transition T25 to ensure that negligible or no time is wasted in removing a token from P10. The following two possible sources of error will be ignored due to the complementing effects among them:

(1) The increase in efficiency due to probability of servicing requests from previous threads, other than those from *active* threads.

(2) The decrease in efficiency due to probability of removing the server while there are pending write accesses to be completed.

For the extreme case of $s_w = 1$, only single server is allocated, thus processor consistency is preserved. We define the index *processor stalling probability, P{stall}* as the *steady state probability of processor being stalled during execution of a thread*. Hence for the PC model, $P_{PC}\{stall\}$ is given by:

$$P_{PC}\{stall\} = P\{buffer\ full\} \qquad (12)$$

The value for *P{stall}* is affected by the buffer size used, the fraction or rate of write accesses and the effective service rate of write buffer.

Figures 7 shows the effects of different buffer sizes on processor efficiency and the corresponding *P{stall}* values. Notice that as *N* increases, the gain in performance over the SC model slowly diminishes since *P{stall}* increases with higher rate of write accesses. Increasing the buffer size generally improves the processor's performance by reducing the *P{stall}* values. For small buffer sizes, the fraction of processor time stalling for a write buffer entry could be very high

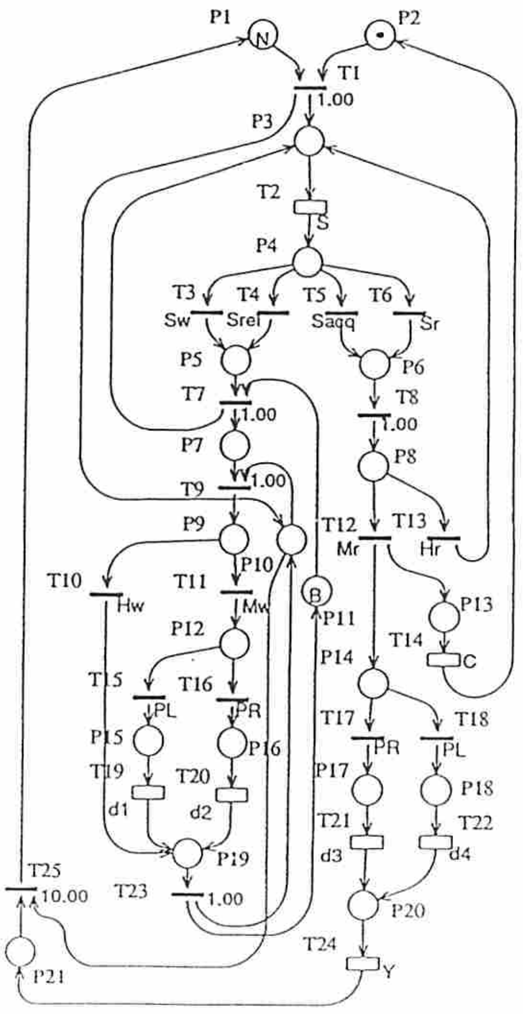
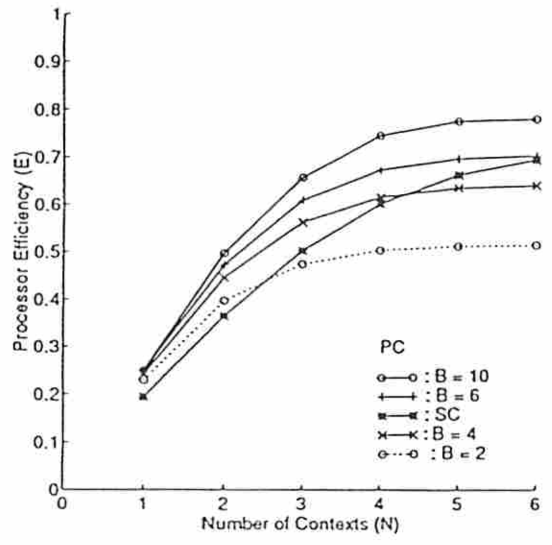
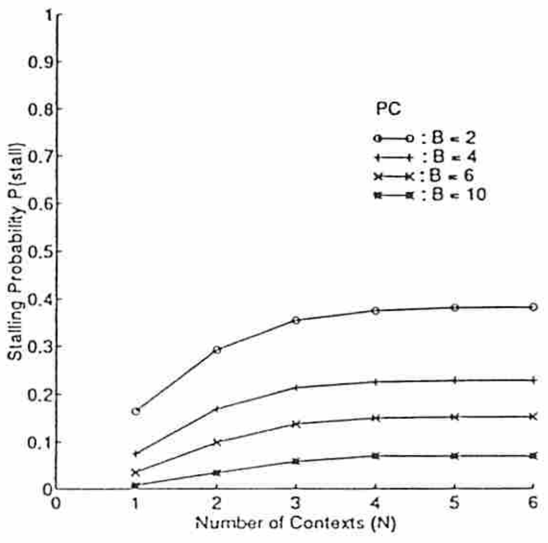


Figure 6: The GSPN model for the processor consistency memory

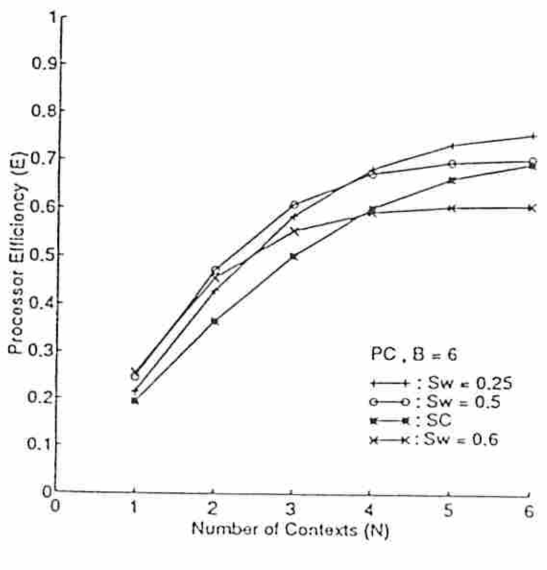
which effectively offsets the gain from buffering. The same problem happens when the fraction of write accesses s_w is increased for a fixed buffer size, as shown in Figure 7c. In stead of a higher expected performance with increased s_w, a loss in performance is observed for larger N due to excessive processor stalling time.



(a) Effects of different buffer size B



(b) *P{stall}* values with different B



(c) Effects of different s_w

Figure 7: The performance curves of the processor consistency memory model

5. Effects of Weak Consistency Memory Model

The *Weak Consistency* (WC) model proposed by Dubois et al. [DSB86] requires that memory is consistent only at the synchronization points. This allows for pipelining of both read and write accesses within the synchronization points.The requirement that synchronization accesses be sequentially consistent may limit the expected performance gain in applications with high synchronization rates. We define the multithreaded processor under the WC model as follows:

A. For each thread:

  a. *Read* : Processor stalls for pending release to perform. Processor issues read access and waits for read to perform. Context switch on cache miss.

  b. *Write* : Processor sends write to write buffer, stall if buffer is full. Writes are pipelined.

  c. *Acquire* : Processor stalls for pending writes and releases to perform. Processor sends acquire and wait for acquire to perform. Context switch on cache miss.

  d. *Release* : Processor sends release to write buffer, stall if 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 as in PC. Since all reads are blocking, pipelining of read accesses within each thread are not allowed. Write buffer is shared among all write and release requests. The GSPN model for WC is shown in Figure 8. An additional branch is added to the net to model the release operations. The main features of the 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 servicing of next *release* operation with pending writes.We assume that all pending writes are from the current running thread. This represents the lower bound in term of expected performance. Alternatively, the inhibitor arcs may be removed if we assume that all pending writes are from other threads, which represents the upper bound.

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

Inhibitor arcs from P27 to T11 and T12 are used to prevent servicing of *acquire* and *read* accesses before the pending *release* operation is completed. This poses another potential bottleneck for the WC model. Inhibitor arcs from P27 to T13 and T14 are used to prevent servicing of *write* accesses before the pending *release* operation is completed. The write requests are placed in the write buffer without stalling the processor. The firing rates *d1* to *d6* are given by:

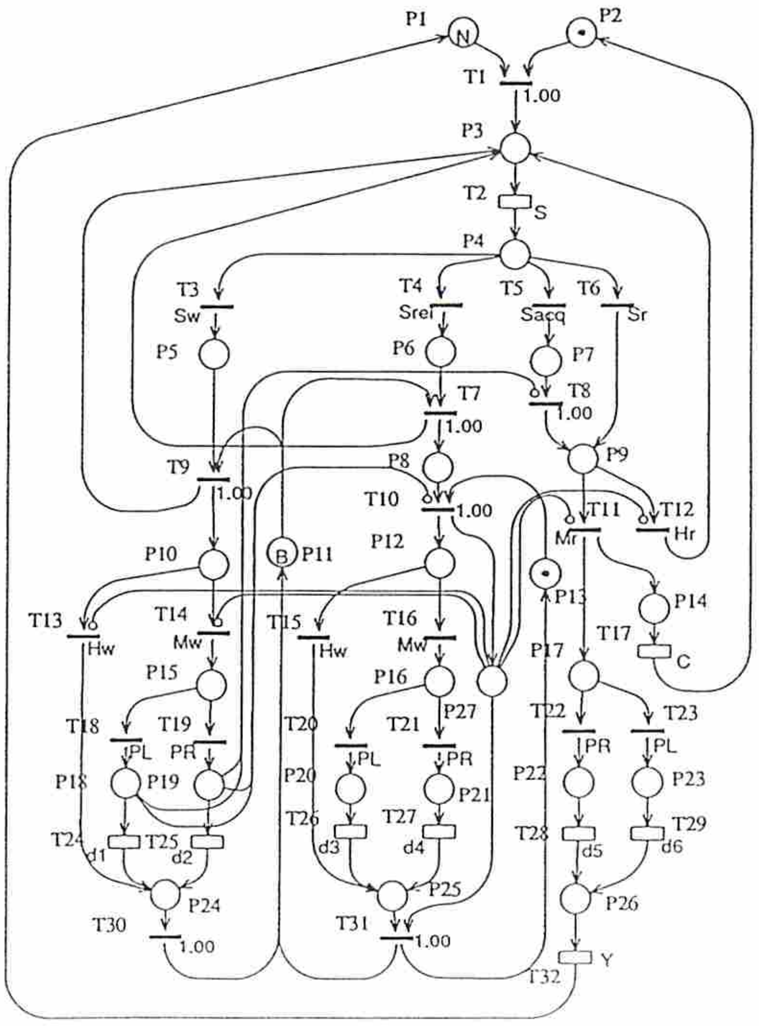


Figure 8: The GSPN model for the weak consistency memory

$$\begin{aligned} d1 &= M_{18} \times l_L \\ d2 &= M_{19} \times l_R \\ d3 &= M_{20} \times l_L \\ d4 &= M_{21} \times l_R \\ d5 &= M_{22} \times l_R \\ d6 &= M_{23} \times l_L \end{aligned} \tag{13}$$

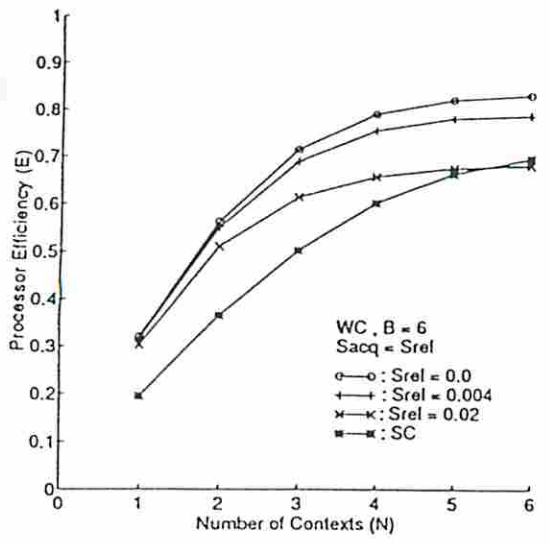
The processor stalling probability under the WC model, $P_{WC}\{stall\}$, can be obtained from:

$$P_{WC}\{stall\} = P\{buffer\ full\} + P\{acquire\ stall\} + P\{read\ stall\} \tag{14}$$

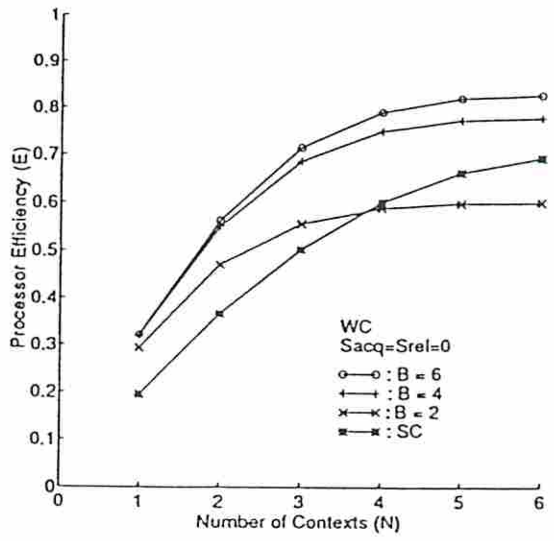
Figure 9a shows the performance curves under the WC model with different synchronization rates. For low synchronization rates ($s_{acq} \approx 0$), the gain in performance is significant, with all write latencies successfully hidden by the write buffer. The drop in performance under higher synchronization rate arise mainly from the increased *P{stall}* value, with the main part of *P{stall}* comes from *acquire* and *read* stalls. For applications with very high synchronization rate, the WC model may give lower performance than the SC model due to excessive processor stalling time, which does not happen under the SC model.

Increasing the write buffer size generally improves the processor's performance under WC model, as in the case of PC model. The buffer size required to completely hide the write latencies is much smaller than that for the PC model due to the higher service rate for write requests under the WC model. Figure 9b shows the effects on performance curves with different buffer sizes, under very low synchronization rate. For the default parameter values, $B = 6$ is sufficient to hide all the write latencies. The *P{stall}* curves for different values of B is similar to that of the PC model, with much smaller corresponding values.

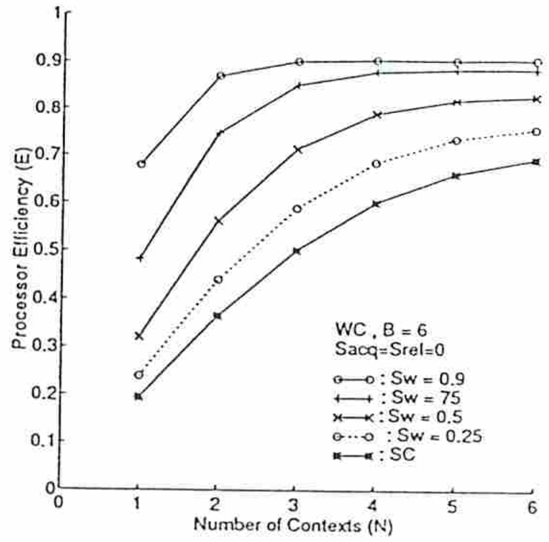
Since write latencies are effectively hidden under the WC model with low synchronization rate, we expect the performance will increase with higher values of s_w. Figure 9c shows the effects of different s_w values under the WC model. The gain is significant with higher s_w for small number of contexts until the constraint of finite buffer size is felt.



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



(b) Effects of different buffer sizes, B



(c) Effects of different s_w

Figure 9: The performance curves of the weak consistency memory model

6. Effects of Release Consistency Memory Model

The *Release Consistency* (RC) model proposed by Gharachorloo et al. [GLLx90] is an extension of WC by exploiting the information about synchronization points using explicit *acquire* and *release* accesses. This allows for pipelining of both read and write accesses within *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 in performance gain. Using the sufficient conditions specified in [GLLx90], we define the multi-threaded processor under the RC model as follows:

A. For each thread:

   a. Read : Processor issues read access and waits for read to perform. Context switch on cache miss.

   b. Write : Processor sends write to write buffer, stall if buffer is full. Writes are pipelined.

   c. Acquire : Processor sends acquire and wait for acquire to perform. Context switch on cache miss.

   d. Release : Processor sends release to write buffer, stall if 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. Also, since reads are blocking, pipelining of read accesses within each thread are 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 Figure 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 the writes, 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 stalling probability under the RC model, $P_{RC}\{stall\}$, is given by:

$$P_{RC}\{stall\} = P\{buffer\ full\} \qquad (15)$$

The performance curves computed for the RC 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 equations (9) and (10) by setting $m_w = 0$, giving $E^{sat} = 0.8333$ for the default parameter values. The performance of the RC model for different values of B and s_w are the same as the WC model with $s_{rel} = s_{acq} = 0$, as shown in Figures 9b and 9c respectively in the previous section.

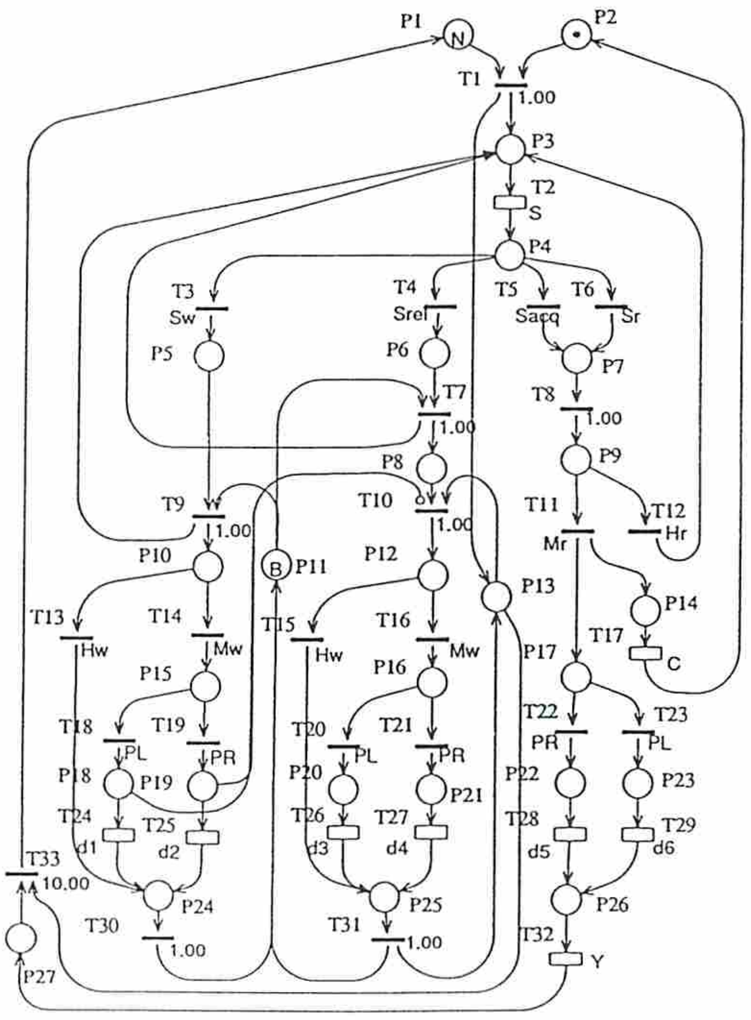
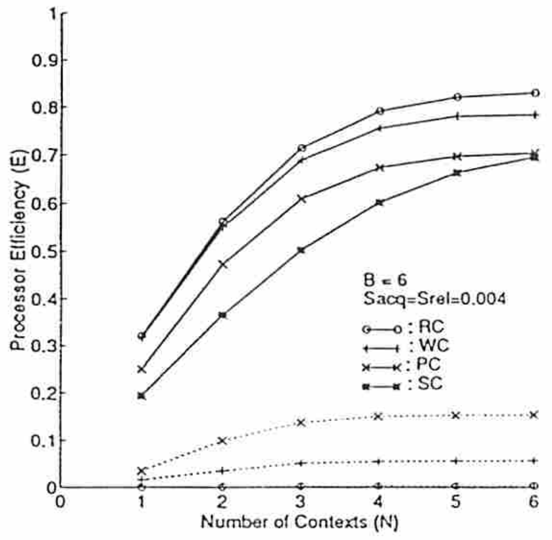
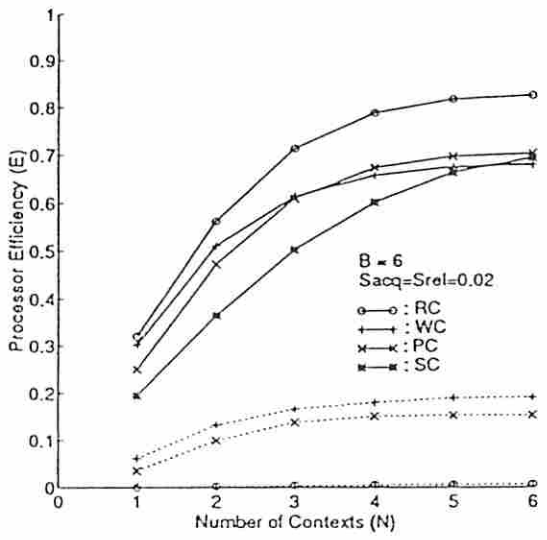


Figure 10: The GSPN model for the release consistency memory

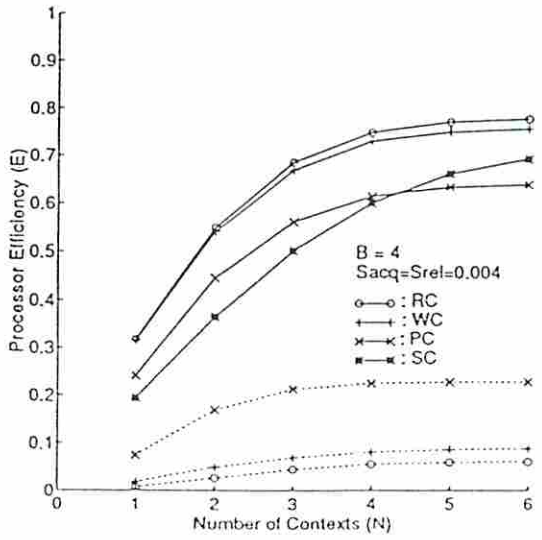
Figure 11 shows the combined performance curves which summarize the relative performance for all the memory consistency models, under different synchronization rates and buffer sizes. The corresponding *P(stall)* values are shown with dotted lines to account for the lost of performance under different consistency models. In general, the RC model gives the best performance for the same buffer size. The WC model gives comparable performance gain over the SC model only when the synchronization rate is reasonably low. The performance of the PC model depends very much on the buffer size used.



(a) Case with $s_{acq} = 0.004$ and $B = 6$



(b) Case with $s_{acq} = 0.02$ and $B = 6$



(c) Case with $s_{acq} = 0.004$ and $B = 4$

Figure 11: The combined performance curves for various memory consistency models

7. Cacheing Effects on Memory Models

In all the models we have considered so far, we assumed that the miss rate for shared data is constant for different degrees of multithreading. This is an optimistic assumption since cache interference occurs between shared data of different threads sharing the same physical cache. A better approximation is to consider the effect of increase in cache miss rate due to cache interference when multiple contexts are running on a processor. A higher number of cache misses will reduce the effective run length, which in turn will lower the processor efficiency.

We will use the cache model derived by Saavedra [Saav90] to estimate the effective miss rate due to multithreading. Assuming that the fixed component of cache interference is negligible and same physical cache is shared equally among all contexts, the miss rate for N contexts, $m(N)$, can be expressed as:

$$m(N) = \begin{cases} m(1)N^{k} , & if\ N \le \lfloor m(1) N^{-1/k} \rfloor \\ 1 , & otherwise \end{cases} \qquad (16)$$

where $m(1)$ represents the single context miss rate and k is defined as the *cache degradation constant*, which is a positive number determined by the workload. For $k \approx 1$ and small N, the model is close to that derived by Agarwal [Agar92], which shows linear relationship between $m(N)$ and the number of contexts.

To apply the cache model on our GSPN models, we need to replace $m(N)$ with $m_r(N)$ and $m_w(N)$, representing different miss rates for read and write accesses. Similarly, k is replaced with k_r and k_w. The performance curves for each memory consistency model can be obtained by substituting m_r and m_w with $m_r(N)$ and $m_w(N)$ respectively. Figure 12 shows the effects of cache degradation on the GSPN models for $k_r = k_w = 0.4$, using default parameter values. The corresponding dotted line shows the performance when cache interference due to multiple contexts is ignored. As expected, a drop in performance is observed for all the consistency models due to higher miss rate as N increases. The performance gain over SC from the relaxed consistency models are still significant, especially for PC where the problem of processor stalling, *P{stall}*, becomes less severe under higher cache miss rate.

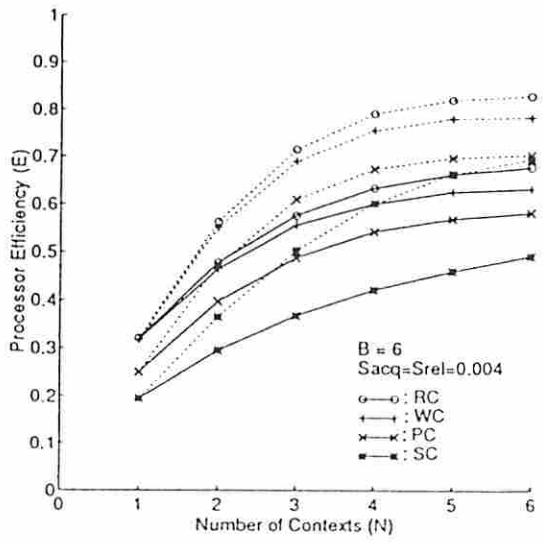


Figure 12: Effects of cache degradation on different memory consistency models

We will compare the performance predicted by our GSPN models against the trace driven simulation results reported by Gupta et al. [GHGx91] on the Stanford's DASH multiprocessor system. The DASH architecture resembles closely with our multithreaded processor model, with the exception that multiple hardware contexts is not used in the prototype construction. The benchmark applications used in the simulation are MP3D, LU and PTHOR. MP3D is a 3-dimensional particle simulator, LU performs LU-decomposition of dense matrices and PTHOR is a parallel logic simulator.

Table 2 lists the parameters that will be used for the GSPN models based on data reported on the benchmark applications. The memory access latencies L_L and L_R are *18* and *73* cycles respectively, with the cache fill overhead $Y = 8$ cycles. The value of L_R is approximated by taking the average of latencies between *Home* and *Remote* nodes. A fixed context switch overhead of 4 cycles is considered. The values of p_L are obtained using the data allocation scheme and accessing pattern of each application.

For MP3D, the particles assigned to a processor are allocated from shared memory in that processor's node, but the space cells are distributed evenly among all nodes. From data provided, 34% and 50% of the misses are from particle and space cell data structures respectively, thus giving $p_L \approx 0.4$, assuming the remaining misses are having the same proportion.

Table 2: Parameter values obtained for MP3D, LU and PTHOR

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 LU, columns are statically assigned to processors on an interleaved fashion, which are allocated from local shared memory. The ratio between local and remote references can be easily obtained as

$$\frac{p_L}{p_R} \approx \frac{N_c}{2} \tag{17}$$

where N_c is the number of columns assigned to each node. For a 200 x 200 matrix divided among 16 processors, $p_L / p_R \approx 6$, giving $p_L \approx 0.857$.

For PTHOR, the logic elements are evenly distributed among all nodes. We assume equal probability for each logic element to be activated at any point in time. Thus for 16 processors, $p_L = 1 / 16 = 0.0625$.

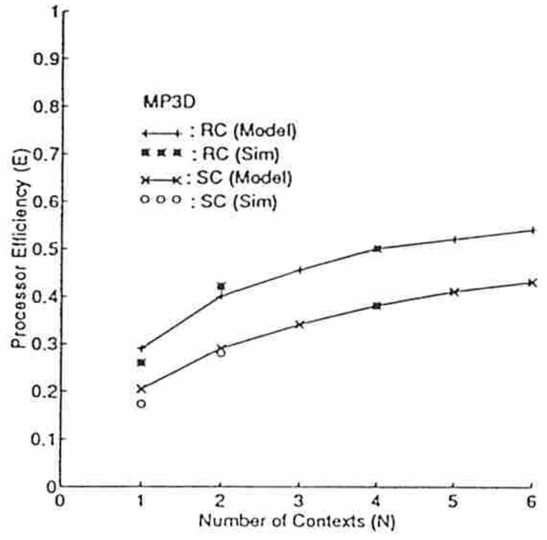
Based on the values of p_L obtained, the average latencies can be approximated by:

$$L_{eff} \approx p_L L_L + p_R L_R \tag{18}$$

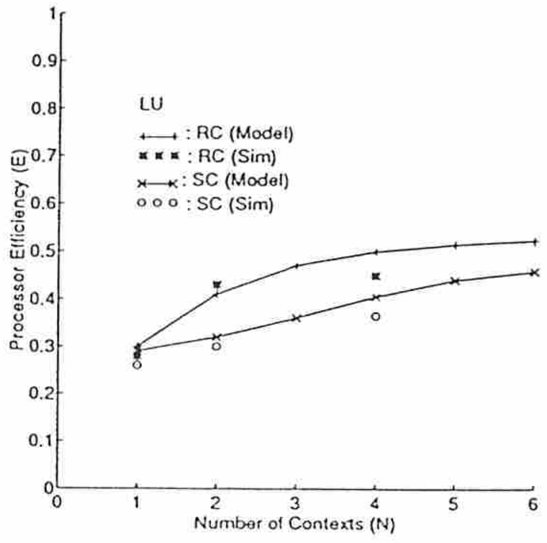
which gives $L_{eff} \approx 50$, 25 and 70 cycles for MP3D, LU and PTHOR respectively. This is close to the 50, 20 - 27 and 60 - 80 cycles as reported in the simulation result.

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

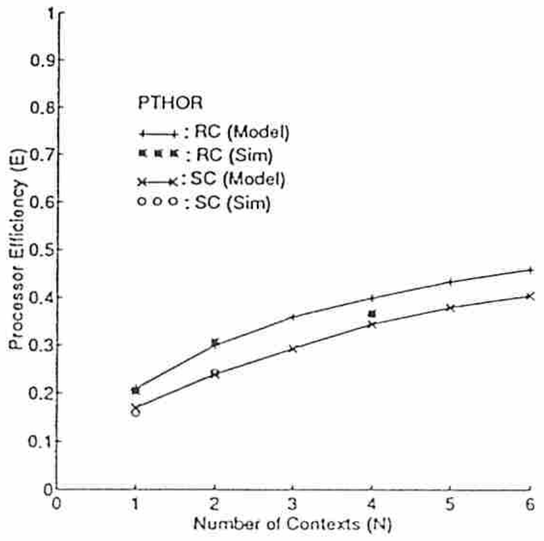
Figure 13 shows the performance curves of the three applications under the SC and RC models. The continuous lines represent the predicted results using GSPN models while the isolated marks represents values reported from trace simulation. In order to limit the number of states generated, we used a buffer size of $B = 6$ for the RC model, instead of $B = 16$ used in the simulation. This is sufficient since the *P[stall]* values due to buffer full condition are negligible in all three cases.



(a) MP3D with estimated $k_r=k_w=0.6$



(b) LU with estimated $k_r=0.4$ *and* $k_w=4.4$



(c) PTHOR with estimated $k_r=k_w=0.55$

Figure 13: Comparison of performance curves computed from the GSPN models with actual simulation results

For the case of LU, the predicted performance and the empirical values correlate well with each other for both the SC and RC models. The predicted values are generally slightly higher than that reported. For MP3D and PTHOR, the values predicted are very close to the simulation results for the case of $N = 1$. Although the validity of the predicted performance values for $N \geq 2$ seems questionable, we expect the actual values to be close to the predicted curves shown. The following are the possible sources of error that might account for some of the discrepancies:

(1) In the simulation on DASH, the processor is blocked from accessing the cache for 4 cycles while cache fill operation of other contexts completes. We did not include this blocking effect into our models, and considered only the overall cache fill overhead.

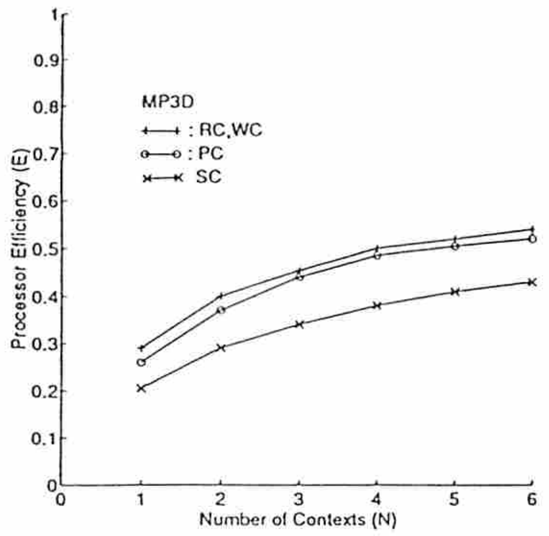
(2) Write hits takes two cycles to complete, as compared to zero cycle used in our model.

(3) Some of the parameters such as p_L and L_R are estimated based on information provided, thus some deviations are expected.

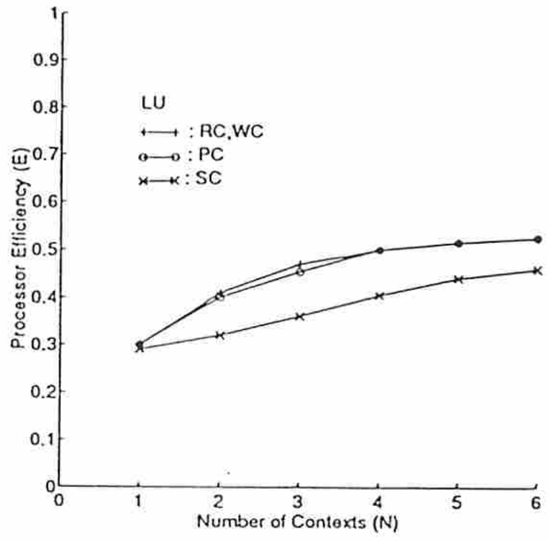
(4) Network and memory contentions which will effectively increase the access latencies were not considered in our model.

(5) Barriers and spinning on synchronization locks which causes additional waiting times were assumed to be negligible in our model.

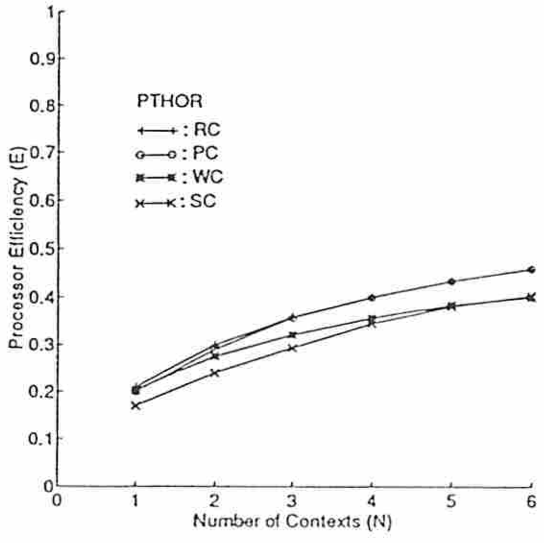
Figure 14 shows the respective predicted performance curves under the SC, PC, WC and RC memory models, using the estimated cache degradation constants obtained earlier. For MP3D, the WC and RC models gives the same performance since the synchronization rate is negligible. The PC model performs comparably well with some lost in performance due to buffer size limitation. For LU, all three relaxed consistency models give nearly the same result since the problems of limited buffer size and high synchronization rate does not exists. For PTHOR, the PC and RC models performs equally well since the buffer size problem does not arise due to low write access rates. The WC model gives considerably lower performance due to high synchronization rate, with $P\{stall\} \approx 0.15$ for $N = 4$. In general, all relaxed memory consistency models provide some performance gain over the SC memory model for $N \leq 4$.



(a) MP3D



(b) LU



(c) PTHOR

Figure 14: The performance curves of benchmarks MP3D, LU and PTHOR under different memory consistency models

8. Relative Merits of Various Memory Models

The GSPN models of multithreaded processors under various memory consistency, namely the SC, PC, WC and RC models were presented. These models are based on architectural assumptions that reads are blocking and context switching is allowed to take place only on a cache miss. Performance curves obtained from numerical solutions of the GSPN models are presented for different parameter values used.

While relaxed consistency models generally provide substantial performance gain over sequential consistency model in a single-threaded processor, the effect on a multithreaded processor is rather dependent on the system parameters and context switching policy used. The amount of processor time stalling due to write buffer being full or access constraints imposed by specific consistency model can have severe impact on the processor performance. A different context switching policy that performs context switch on stall conditions may be applied to reduce the processor stall time. However, the implementation issues on how to handle the current request and the additional context switching overhead incurred requires further investigation.

The size of write buffer determines the amount of processor time stalling for a buffer entry. The WC and RC models require relatively small buffer size to hide all the write latencies through pipelining of write accesses, without stalling the processor. However, the PC model requires a much larger buffer size due to its relatively slower service rate for write accesses since pipelining within each thread is not allowed. For small buffer sizes or high write access rates, the PC model may perform worse than the SC model due to excessive processor stalling time. Given sufficient buffer size and low write access rate, the PC model may perform better than the WC model in applications with high synchronization rate, such as PTHOR, since no synchronization constraints is imposed on the PC memory model.

The performance of the WC model is generally governed by the rate of synchronization in the program. For low synchronization rate, the WC model performs as well as the RC model in hiding all write latencies. For 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 low synchronization rate, such as MP3D and LU, the WC model is expected to give performance gain comparable to that from the RC model. In addition, higher fraction of write accesses provides greater potential for performance gain under the WC and RC models, with relatively small buffer sizes.

9. Conclusions

The basic GSPN model of an abstract multithreaded processor is proposed and the performance curves obtained are compared with Saavedra's results. The GSPN is first extended to model a multithreaded processor equipped with coherent caches, under the constraints of SC consistency model. The performance curves obtained under variations on some parameter values are analyzed. The GSPN is further extended to model the multithreaded processor under three relaxed consistency memory models, namely the PC, WC and RC. Each model is treated separately under different access constraints and a unified context switching policy.

The effects of different buffer sizes and synchronization rates are presented for each of the relaxed memory models. Based on the results computed, the RC model always gives the best performance by hiding all write latency with reasonably small buffer size. The performance of the PC model depends very much on the buffer size used and the effective service rate of the write buffer. The performance of the WC model is primarily affected by the synchronization rate of the application. With low synchronization rate and a large buffer size, all three relaxed consistency models gives comparable performance gain over the SC memory model.

A simple cache model is incorporated into the GSPN models to model the effect of cache degradation on multithreaded processor performance. A comparison was carried out between the predicted performance using the GSPN models and some simulation data 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 models require further validation due to lack of simulation results to compare with. The Petri net models can be extended to study the effects of other latency tolerating mechanisms, such as the effects of data prefetching reported by Mao and Hwang [MH93]. Also, the context switching policies and network and memory contention effects need to be analyzed further, both analytically and through simulation experiments.

References

[ACCx90] 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.

[Agar92] A. Agarwal, "Performance tradeoffs in multithreaded processors," *IEEE Trans. on Parallel and Distributed Systems*, 3(5) pp 525-539, Sept. 1992.

[Bell92] Gordon Bell, "Ultracomputers - A Teraflop Before Its Time," *Communications of the ACM*, 35(8), pp 27 - 47, Aug 1992.

[Ajmo84] Ajmone Marsan et al. "A class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems," *ACM Trans. on Comp. Sys.*, 2(2), pp 93-122, May 1984.

[Chi87] G. Chiola, "GreatSPN user manual version 1.3," *Technical report, Dipartimento di Informatica, Universita di Torino*, Torino, Italy, Sept. 1987.

[Cho93] Y.K. Chong, *Effects of Memory Consistency Models on Multithreaded Multiprocessor Performance*. MSc thesis, University of Southern California, May 1993.

[DSB86] M. Dubois, C. Scheurich and F. Briggs, "Memory Access Buffering in Multiprocessors," *Proc. 13th Int'l Symp. Comp. Arch.*, pp 434-442, June 1986.

[GGH91] K. Gharachorloo, A. Gupta and J. Hennessy, "Performance Evaluation of Memory Consistency Models for Shared-memory Multiprocessors," *Proc. 4th Int'l Conf. on Arch. Support and Prog. Lang. and O.S.* , April 1991.

[GHGx91] 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.

[GLLx90] 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.

[Good89] J.R. Goodman, "Cache Consistency and Sequential Consistency," Technical Report 61, IEEE SCI Committee, 1989.

[Hwa93] K. Hwang, *Advance Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill Inc. New York 1993.

[Lam79] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Computers*, 28(9), pp 241-248, Sept. 1979.

[LH89] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Comp. Sys.* 7(4), Nov 1989.

[LLGx90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta and J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," *Proc. 17th Int'l Symp. on Comp. Arch.*, pp 148-159, May 1990.

[LLGx92] 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.

[MH93] W. Mao and K. Hwang, " Queueing Analysis of Data Prefetching Effects on the Performance of Multithreaded Multiprocessors " *Technical Report No. 93 - 20*, Dept. of EE-Systems, Univ. of Southern California, Los Angeles, CA. April 1993

[NL91] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer*, Aug. 1991

[Saav90] R.H. Saavedra-Barrera, D.E. Culler and T. von Eicken, "Analysis of Multithreaded Architecture for Parallel Computing," *Proc. 2nd ACM Symp. Parallel Algorithm and Architecture*, July 1990.

[Saav91] R.H. Saavedra-Barrera and D.E. Culler, "An Analytical Solution for a Markov Chain Modeling Multithreaded Execution," Technical Report UCB/CSD-91/623, Computer Science Division, UC Berkeley, March 1991.

[TMC91] TMC, "The CM-5 Technical Summary," Thinking Machines Corp., Cambridge, MA, 1991.

[WG89] 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.