Data Prefetching Effects on the Performance of Multithreaded Multiprocessors [†]

Weihua Mao, Kai Hwang, and **Rafeal H. Saavedra**

Technical Report No. 93 - 20
May 1993

† This paper was submitted to the Journal of *Parallel and Distributed Computing*, Special issue on Scalability of Parallel Algorithms and Architectures. All rights reserved by the authors.

Data Prefetching Effects on the Performance of Multithreaded Multiprocessors [1]

Weihua Mao, Kai Hwang, and Rafael H. Saavedra

Departments of Electrical Engineering
and Computer Science
University of Southern California
Los Angeles, CA 90089

Abstract:

We study the effects of software-controlled data prefetching on the performance of multithreaded multiprocessors with distributed shared memory. With prefetching on single-thread processors, the processor efficiency increases almost linearly with respect to the hiding of latency for remote memory accesses. The performance gain relies on the coverage factor of prefetching. When the coverage factor is large (> 0.6), the remote access latency can be hidden effectively. Otherwise, the prefetching does not help much.

To combine prefetching with multithreading, we propose a closed-loop queueing network to model the behavior of scalable multiprocessors. When the degree of multithreading is small (≤ 6), prefetching can enhance the processor efficiency by a factor of 10-40%, depending on the coverage factor and the effective hiding of the access latency. After the processor reaches saturation, prefetching is shown to be ineffective to improve the efficiency. A queueing network transformation technique is presented to obtain an approximate solution of the performance model. Our queueing results are verified by the numerical results obtained from equivalent Petri net simulation experiments.

We also studied the adverse effects caused by the increase of network latency and the replacement of prefetched cache blocks. The results show that prefetching cannot improve the processor efficiency in a heavily loaded network. Also the prefetching instructions should be issued closer to the actual reference in order to avoid potential conflicts in cache.

Index Term: shared-memory multiprocessors, latency hiding techniques, software-controlled data prefetching, stochastic Petri nets, machine repair model, multithreading, scalability analysis.

[1]All rights reserved by the authors. Manuscript submitted to the *Journal of Parallel and Distributed Computing* special issue on Scalability of Parallel Algorithms and Architectures, April 1993. This paper was also released as *Technical Report* No. 93-20, Dept. of EE-Systems, USC, Los Angeles, CA 90089-2562, USA. The authors can be reached via Email "kaihwang" or "mao" @aloha.usc.edu, or saavedra@pollux.usc.edu.

Contents

1 Introduction

In a shared-memory multiprocessor, as the number of processors increases, the interconnection network imposes a corresponding longer latency between the processing nodes. Each processor may experience an increasing time delay in remote memory access or in interprocessor communication, which implies the processor efficiency may drop sharply. Various latency-reducing or latency-hiding techniques were proposed in recent years [12]. These include the use of distributed shared memory [16], multiple-context processors [1, 3, 19, 20, 21], relaxed consistency memory models [7, 8, 9], software-controlled data prefetching [10, 11, 18], and hardware-supported coherent caches [2, 4, 15].

Prefetching uses prior knowledge about the expected use of data structure in a program. The purpose is to move the demanded data block closer to the processor before it is actually referenced. Software controlled prefetching is achieved by inserting prefetching instructions in program codes. This can be done explicitly by the programmer or automatically by the compiler by using data-dependency analysis. This technique is more effective for scientific codes, where the data demand inside highly nested loops tend to follow predictable patterns.

We concentrate on the use of multithreaded processors and software-controlled data prefetching to hide the latencies for remote memory accesses. We study the effects of data prefetching without binding. The prefetched data block is destined for the cache, not for the local memory. The prefetched data remains visible to the cache coherence protocol and thus kept consistent, until the processor actually reference the prefetched data in cache.

In the past, prefetching effects were studied primarily through trace-driven simulation experiments, as reported in Mowry and Gupta [18] and in Gupta et al [11] at Stanford University. We apply queueing networks and Petri nets to model the problem analytically. Our queueing model covers the effects of both multithreading and prefetching. An approximate solution to the queueing model is obtained. Based on the approximate solution, processor efficiency curves are plotted and interpreted.

The queueing results are verified with the numerical results obtained from equivalent Petri net simulation experiments. The main conclusion from our study is that prefetching is only effective when the number of threads per processor is small (≤ 6) or the coverage factor is high. The effectiveness of prefetching is assessed with respect to the case of no prefetching. The percentage of cache misses (that would occur without prefetching), which can be covered by the issues of prefetch instructions, is defined as the *coverage factor*.

Prefetching coverage results in three possible consequences, which affect the latency hiding degree and the cache hit ratio:

1. The prefetched data block has been loaded into local cache resulting in a no cache miss for the particular data block.

2. The data block has been prefetched but not been loaded into the cache yet. This corresponds to the situation that the data is still on the fly.

3. The data block being prefetched has been lost due to cache invalidation, or network contention, or memory conflicts, etc.

Only Case 1 can reduce the cache miss ratio. Case 1 can hide the memory latency fully. Case 2 can hide the latency partially, but cannot avoid the cache miss. Case 3 corresponds to the situation in which the prefetch fails to hide the latency. The performance is very sensitive to the coverage factor for the prefetched data to be actually referenced. When the coverage factor is large (> 0.6), the latency will be mostly hidden. To increase the coverage factor involves the effort from the algorithm designers, the programmers, and the compiler.

There are other limitations of the software-controlled prefetch too. For example, we cannot prefetch across synchronization points and we cannot prefetch variables used to implement synchronization mechanisms. We ignore these in our paper by assuming they have been taken care of by the programmer or the compiler already.

The remaining of the paper is organized as follows: Section 2 presents a general model for scalable multiprocessor supported with various latency tolerating mechanisms. Section 3 studies the prefetching effects on single-thread processors. Section 4 reveals the prefetching effects on multithreaded processors. Section 5 presented the verification results from Petri nets experiments. Finally, we conclude on research contributions and comment on future work needed.

2 A Scalable Multiprocessor Model

A scalable multiprocessor system is conceptually depicted in Figure 1. The system consists of P processors with distributed physical memories, which are globally shared with a single address space by all processors [16]. The system components are interconnected by a scalable interconnection network, which can be modularly constructed from small to larger network sizes.

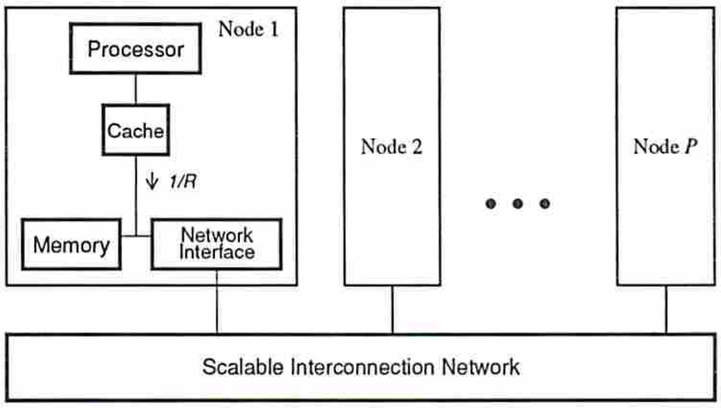


Figure 1: The architecture of a multiprocessor.

Two mechanisms are included in our multiprocessor model. The ultimate goal is to enhance the scalability and programmability of the system. First, data prefetching is software-controlled. Second, multithreading is achieved using multiple-context processors. We will use the terms *context* and *thread* interchangeably. Therefore, *multiple contexts* and *multithreading* are equivalent terms. The Stanford DASH multiprocessor [15] most resembles this model, except multiple-context processors have not yet been built into the current DASH prototype. Summarized below are basic parameters to be used in the multiprocessor model:

1. N: The *number of threads* or *contexts* that can be executed simultaneously in each processor. The value of N is determined by the complexity of the processor architecture.

2. C: The *context switching overhead* accounts for the cycles lost in performing context switching in a processor. We assume a context switching policy upon every cache miss. The value of C is assumed small with only a few cycles.

3. R: This is the *average run length* of a single thread, before it issues a memory request or gets switched out. The inverse $1/R$ represents the *memory request rate* issued from a processor. The run length R depends on the thread granularity as well as the program behavior. Typical value of R ranges between tens to hundreds of processor cycles.

4. q: The *percentage of cache misses* resulting in remote memory requests.

5. L_l: The *access latency for local memory*, which requires tenths of processor cycles depending on the memory technology.

6. L_r: The *average latency* of a processor to access a remote memory through the network, upon a cache miss. This memory latency is a variable and depends on the machine architecture, network contention, memory organization, and coherence protocol. For a deterministic analysis, one can assume a constant L_r representing the *average memory latency*. Typical values of L_r range from hundreds to thousands of processor cycles.

7. f: The *coverage factor*, which is the percentage of cache misses (that would occur without prefetching), which can be covered by the issues of prefetch instructions.

8. V: The *overhead for prefetching*, which corresponds to the extra time for executing the prefetch instructions inserted in the original code.

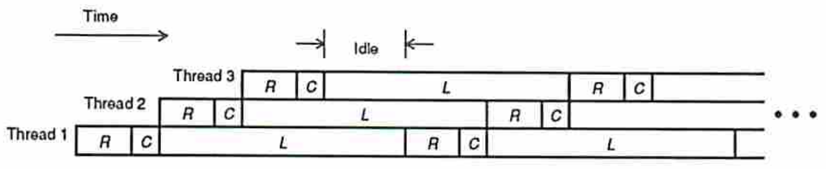
9. L_p: This is the *average latency* for remote memory accesses after data prefetching. Again, L_p is a random variable in the range $0 \leq L_p \leq L_r$. An effective prefetching will hide the remote memory latency with $L_r - L_p$ cycles. In the worst case of $L_p = L_r$, no latency hiding is possible. In the best case of $L_p = 0$, the full remote latency L_r is hidden by prefetching.

10. E: The *processor efficiency* is defined as the percentage of time a processor is busy during the run-length period (R). A processor is considered idle during context switching (C), or when executing prefetching instructions, or when the processor is waiting for a local or remote memory access (with latency L_l and L_r respectively) to complete.

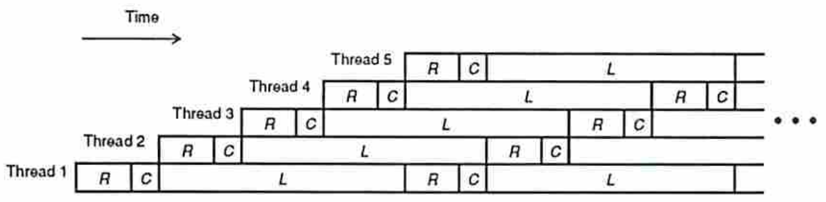
In general, we can assume that $L_r > L_l > C$ and $L_r > R > C$. For example, typical values are: $L_r = 120$ cycles, $L_l = 10$ cycles, $R = 16$ cycles, and $C = 2$ cycles. Agarwal [1] has shown that only for a small number of contexts, i.e. N equals to 2 to 6, multithreading (without prefetching) shows appreciable performance advantages.

In this paper, we use the subscripts m to denote the effect of multithreading, and the subscripts p to denote the effect of prefetching, For example, E_m denotes the processor efficiency with multithreading alone. E_{mp} refers to processor efficiency associated with the combined use of prefetching and multithreading. Saavedra, et al [20] has derived expressions to represent the multithreaded processor efficiency E_m, as a function of R, L_r, C, and N. A typical multithreaded processor efficiency curve is shown in Figure 2c.

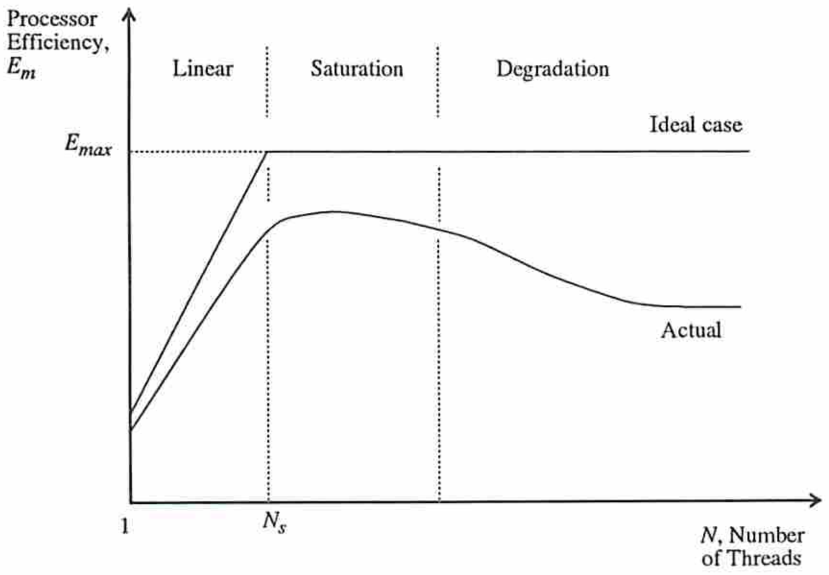
The *linear region* corresponds to unsaturated situation, in which the number of



(a) Unsaturated situation (linear)



(b) Saturated situation



(c) Processor efficiency of a multithreaded multiprocessor

Figure 2: The processor efficiency of a multithreaded multiprocessor.

threads is not enough to hide the memory latencies as shown in Figure 2a. Processor idle periods could exist in the linear region. The *saturation region* corresponds to the situation in which all memory latencies are hidden through multithreading. The saturation point separates the linear region from the saturation region. As the number of contexts becomes very large, the cache miss may become high, a problem called *cache thrashing* by Agarwal [1]. When this happens, the processor enters a third region, called *degradation region* [2] as shown on the right hand side of Figure 2c.

The ideal operating range will be around the saturation point. It is not advantageous to operate in the degradation region, in which the cache hit ratio becomes almost zero. Both the performance and cost factors become very bad to operate in the degradation region.

3 Prefetching Effects on Conventional Processors

Conventional processors support single-thread operations. Such processors are studied with a deterministic analysis. The effects of hiding memory latency through prefetching are revealed below. We derive first the processor efficiency under the assumption that L_r, L_p, and f are independent of each other. The in Section 3.2 and 3.3, we will study the prefetching effects on network latency and replacement of cache blocks, in which L_r and L_p become function of f.

3.1 Software-Controlled Data Prefetching

In software-controlled data prefetching, the processor must execute a prefetch instruction to initiate the operation [18]. Prefetches are introduced either explicitly by the programmer, or automatically by the compiler, or dynamically by the runtime system of a programming environment. Comparing with hardware-controlled prefetching, the software-

[2]Both Agarwal [1] and Saavedra, et al [20] have studied the cache degradation effects on multithreaded multiprocessors. We ignore this effects in paper but will include this effect in the further researches.

controlled prefetching allows the prefetching to be done selectively and extends the possible time interval between prefetch issue and actual reference. The disadvantage is that programmer or software intervention is required, which adds more burden on the programmer. Further, no programmer or compiler can do a perfect job, so unnecessary prefetches may be introduced.

Figure 3 shows the effect of data prefetching. Without prefetching, the processor has to wait for five remote memory latencies (L_1 to L_5) as shown in Figure 3a. In Figure 3b, four prefetch instructions (f_2 to f_5) are issued for the remote memory requests 2 to 5, respectively. A few extra cycles are added to perform prefetching instructions at point f_2 to f_5. However, the remote memory latencies, L_3 and L_4, are totally hidden, while the remote memory latencies, L_2 and L_5, are reduced to L'_2 and L'_5, respectively. As a result, the processor spends less time waiting. Therefore the processor efficiency can be improved.

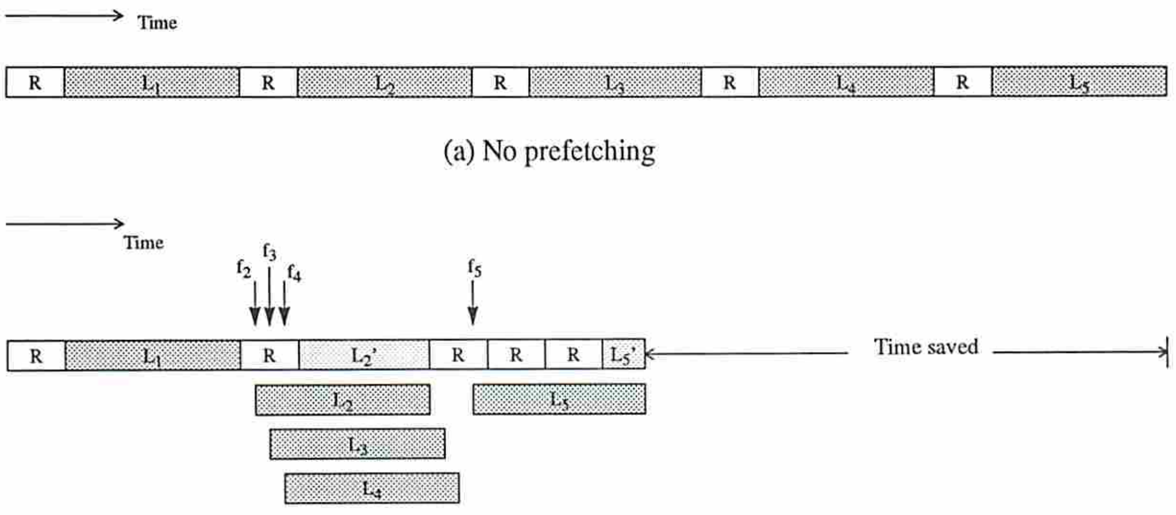


Figure 3: The effect of prefetching (shaded areas corresponding to memory access latencies).

The major benefit of data prefetching is to hide the latency seen by the processor. Effectively, prefetching will reduce the remote memory latency from L_r to L_p. If a memory request has been prefetched, the time spent on the request equals $V + L_p$.

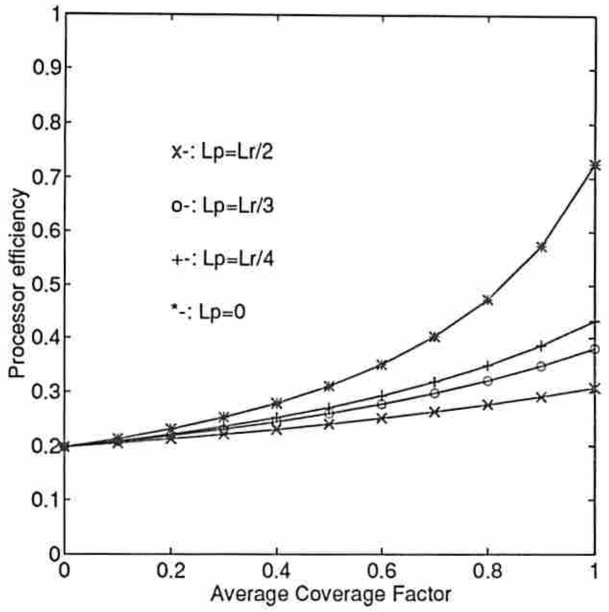
Let f be the fraction of cache misses covered by prefetches. Consider a thread running on a processor for R cycles before issuing a memory request. If the request is for local memory (with probability $1 - q$), it needs to wait for L_l cycles. If the request is for remote memory request (with probability q), it needs to wait for L_r cycles with a probability $1 - f$ (no prefetch), or for L_p cycles with a probability f (covered by prefetch). In addition, the processor spends fqV extra cycles to perform the prefetching. Therefore, the processor efficiency E_p of a single-thread processor with prefetching is represented as:

$$E_p = \frac{R}{R + (1-q)L_l + q[f(V + L_p) + (1-f)L_r]} \tag{1}$$

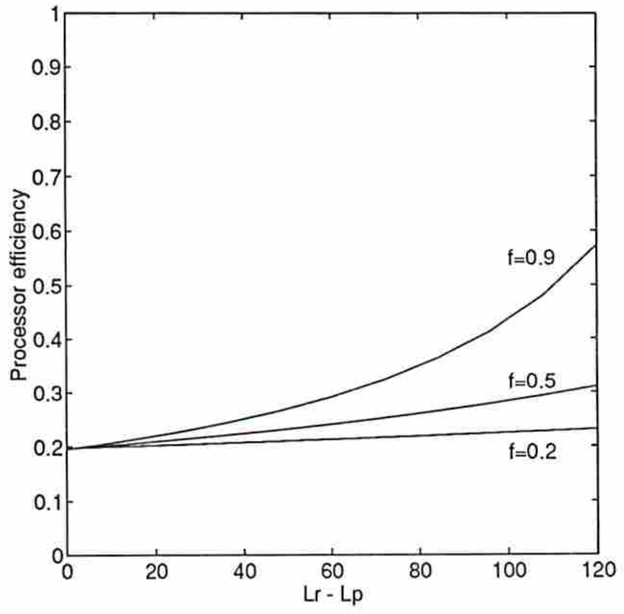
Figure 4 plots Eq. 1 showing the effect of coverage factor and of prefetched latency on the performance of single-thread processor. The parameters used are summarized in Table 1. We plot the processor efficiency as a function of the coverage factor. These four curves correspond to the cases that the latency is reduced to zero, to one half, one third, and one quarter of the full latency, respectively. The processor efficiency depends on the values of f, L_p and V. When the coverage factor f is small (< 0.4), the effect of prefetched latency L_p is almost negligible. Only when f is sufficiently large (> 0.6), the prefetched latency L_p has greater impact on the processor efficiency.

Table 1: Parameters Used in Processor Efficiency Plotting and in Petri Net Simulations

Parameter	R	N	C	V	f	q	L_l	L_r
Value	16 cycles	1-10	4 cycles	2 cycles	0.8	0.5	12 cycles	120 cycles



(a) Effects of coverage factor.



(b) Effects of prefetched latency.

Figure 4: Effect of prefetching on a single-thread processor performance

3.2 Effects of Network Traffic on Memory Latency

Next, we consider the variation of the memory latency L. So far, we have assumed that the memory latency is constant with prefetching or without prefetching. Since the prefetch instructions increase the network traffic, the memory latency may actually be prolonged. The network latency can be related to the network traffic by the hypothetical curve shown in Figure 5.

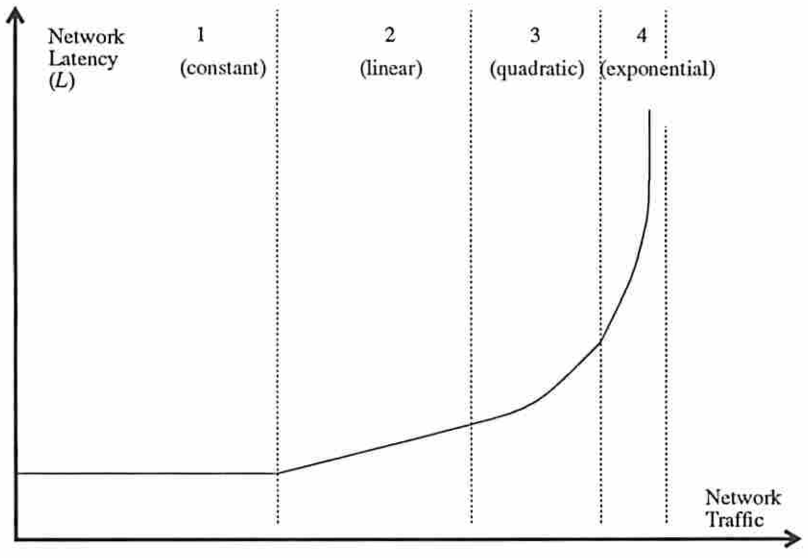


Figure 5: Hypothetical relation between network latency and network traffic.

In region 1, the network is under loaded, increasing the traffic will not affect the latency. In region 2, the latency increases linearly. In region 3 the latency increases quadratically. In region 4, the network is saturated with heavy traffic and the latency increases exponentially. The interconnection network of a scalable multiprocessor should operate in the lower regions only.

Now, since the L_r and L_p increase as f increase, we use the notation $L_r(f)$ and $L_p(f)$ to represent the remote latency and prefetched latency as functions of the coverage factor f. Note that $L_r = L_r(0)$ and $L_p = L_p(0)$ when $f = 0$. Let $Q(f)$ be the traffic in the network as a function of f. The following expression represents a simple general model for network latency in terms of the network traffic:

$$L_r(f) = Q(f)^i L_0 \tag{2}$$

where L_0 is the latency of a completely unloaded network and i depends on the characteristic of the network. When the network latency increases linearly or quadratically as network increases, i equals to 1 and 2, respectively. We can express $L_r(f)$ in terms of L_r as:

$$L_r(f) = \left[\frac{Q(f)}{Q(0)}\right]^i L_r = [\Delta Q(f)]^i L_r \tag{3}$$

where $\Delta Q(f)$ is the relative increase in network traffic due to prefetching. $\Delta Q(f)$ can be computed by taking the ratio of the number of remote requests with and without prefetching:

$$\Delta Q(f) = \frac{R + qL_r + (1-q)L_l}{R + q[f(V + L_p(f)) + (1-f)L_r(f)] + (1-q)L_l} \tag{4}$$

The prefetched latency $L_p(f)$ is more difficult to calculate. Let's define a term called *running cycle*, which the time that the processor runs for R cycles and then waits for a memory request for certain cycles. When we ignore the network effects, there are $n = \frac{L_r - L_p}{R+qL_r+(1-q)L_l}$ running cycles between the time that prefetch was issued to the time that processor reference the data. Each running cycles takes $R + qL_r + (1-q)L_l$ cycles long. When we consider the network effects, there are still n running cycles between the time that prefetch was issued to the time that processor reference the data. But now each of these running cycles takes $R + q[f(V + L_p(f)) + (1-f)L_r(f)] + (1-q)L_l$ cycles long. Therefore we have the following equation:

$$n = \frac{L_r - L_p}{R + qL_r + (1-q)L_l} = \frac{L_r(f) - L_p(f)}{R + q[fL_p(f) + (1-f)L_r(f)] + (1-q)L_l} \tag{5}$$

After rearranging items and combined with Eq. 3, we obtained the following equation set:

$$\begin{aligned} L_r(f) &= [\Delta Q(f)]^i \, L_r \\ L_p(f) &= L_r(f) - n\,[R + q(fL_p(f) + (1-f)L_r(f)) + (1-q)L_l] \end{aligned} \tag{6}$$

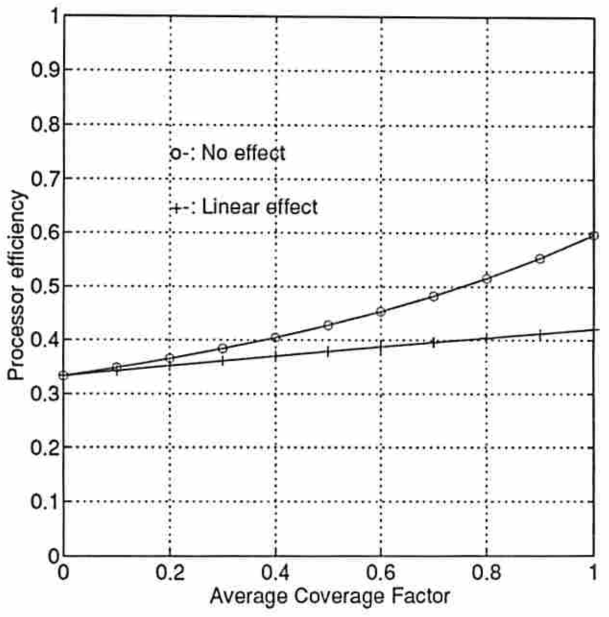
This is a non-linear equations with unknowns of $L_r(f)$ and $L_p(f)$. When $i = 1$, $L_r(f)$ and $L_p(f)$ can be solved analytically by solving a quadratical function. When $i \neq 1$, we can only use numerical method to obtain $L_r(f)$ and $L_p(f)$. The processor efficiency can be obtained by substituting the values of $L_r(f)$ and $L_p(f)$ into the following equation which is modified from Eq. 1:

$$E'_p = \frac{R}{R + (1-q)L_l + q\,[f(V + L_p(f)) + (1-f)L_r(f)]} \tag{7}$$

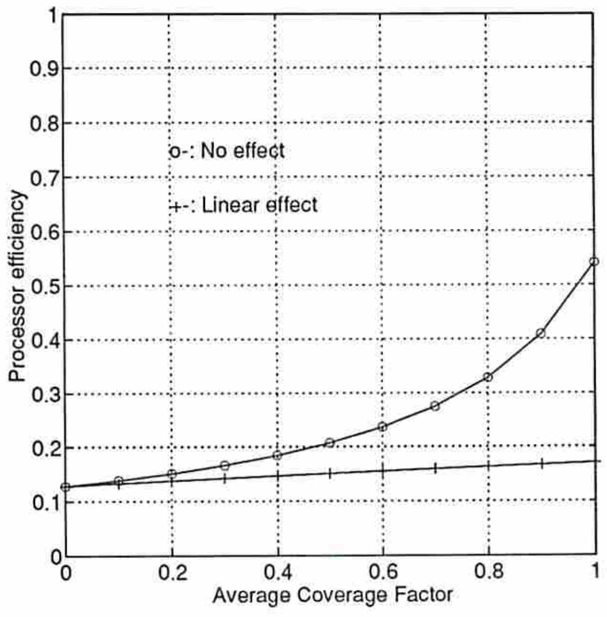
Figure 6 plots Eq. 7 showing the effects when network traffic affects memory latency linearly. Figure 6a has fewer remote accesses ($q = 0.2$), and Figure 6b has more remote accesses ($q = 0.9$). As the network latency is affected by the traffic, the advantages of prefetching decrease, especially when there are more remote memory accesses. When the network latency changes from not be affected by traffic, to linearly increases as the traffic increases, the improvement of processor efficiency drops from 200 to 300 percent, to only 20 to 40 percent. In other words, prefetching will hide the long latency, but will not help a saturated network.

3.3 Replacement of Prefetched Cache Blocks

Now we consider another adverse effect of prefetching. In Eq. 1, it is assumed that $L_p \geq 0$, i.e. the processor still has to wait for L_p cycles before the prefetched request is completed. In this case, the prefetched cache blocks will not interfere each other since they will be referenced by the processor immediately after they are loaded into the local cache.



(a) The case of $q = 0.2$.



(b) The case of $q = 0.9$.

Figure 6: Adverse prefetching effects, which increase the memory latency in a multiprocessor built with single-thread processors.

If the prefetch operations are completed before the processor actual reference there is a probability that this prefetched block may be replaced out before the processor actually reference it.

Consider the example illustrated in Figure 7, where r_j indicates the time when request j is referenced by processor and t_j indicates the time when request j is prefetched into the cache. Assume $t_j < r_j$, If any of the request among $r_{i+1}, \ldots, r_{j-1}$ is mapped into the same location in cache as r_j, the prefetched data block j will be replaced before r_j. As a result, the effect of this prefetch is canceled and request j has to spend the full latency L to be fetched from the memory again.

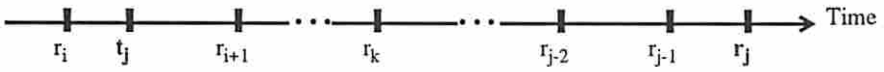
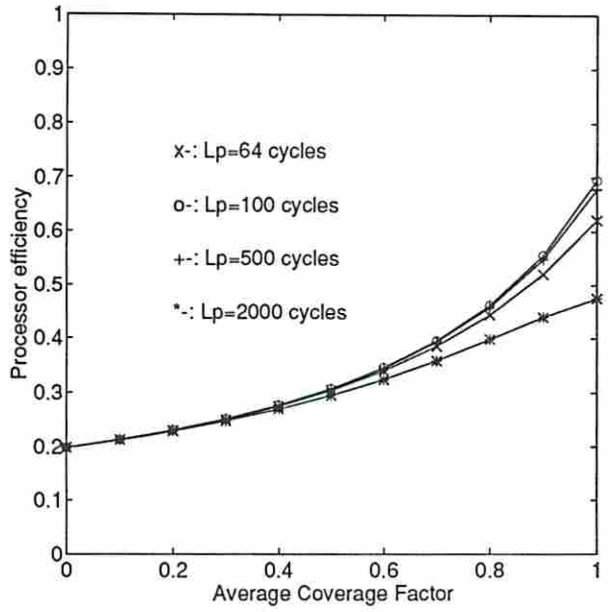


Figure 7: Timing sequence for the requests and prefetches

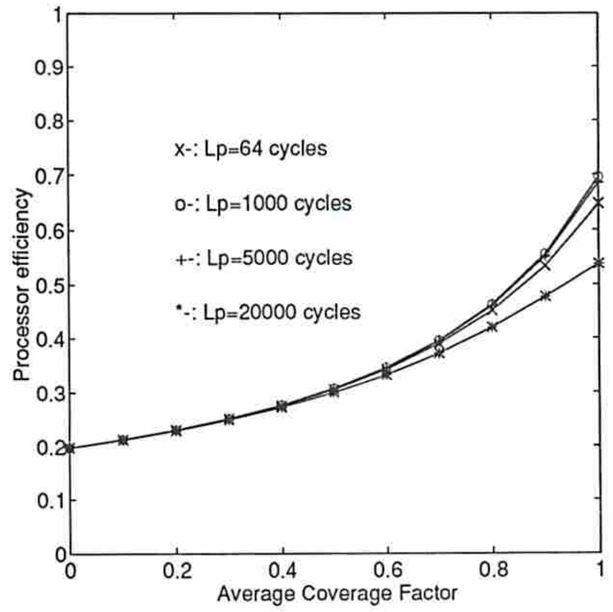
Let B be the cache size in term of blocks. Assume blocks are mapped into the cache uniformly using directly mapping scheme. The probability that two memory requests are mapped into the same block frame is $\frac{1}{B}$. Assume the probability for different requests are independent . Thus the number of requests which mapped into the same location follows a *binomial distribution.* If the prefetch instruction for j was issued L_h cycles before the actual processor reference. There are approximately $\frac{L_h - L_r}{R+(1-q)L_l+q(1-f)L_r}$ requests between t_j and r_j. The probability that there is at least one request between t_j and r_{j-1} is mapped into the same location as request j is $p_c = 1-(1-\frac{1}{B})^{\frac{L_h-L_r}{R+(1-q)L_l+q(1-f)L_r}}$. Therefore Eq. 1 can be modified as:

$$E_p = \begin{cases} \frac{R}{R+(1-q)L_l+q[fV+fL_p+(1-f)L_r]} & \text{if } L_h < L_r \\ \frac{R}{R+(1-q)L_l+q[fV+(p_c+1-f)L_r]} & \text{if } L_h \geq L_r \end{cases} \tag{8}$$

Figure 8 shows the processor efficiency curves with and without the block replacement effects. The cache block size for Figure 8a and b are 1024 blocks and 16K blocks, respectively.



(a) A small cache with 1024 blocks



(b) A large cache with 16K blocks

Figure 8: Processor efficiency with and without cache replacement effects.

There are some other adverse effects too. For example, if prefetched request j (Figure 7) replaces the block which was originally in cache, and then the exact block is referenced by processor between time t_j and r_j, there will be an additional cache miss. As a result, the processor will have to fetch the block from memory again. The magnitude of this adverse effect should be about the same as in Eq. 8, since the probability this even will happen is about the same as p_c under the same assumptions about the cache.

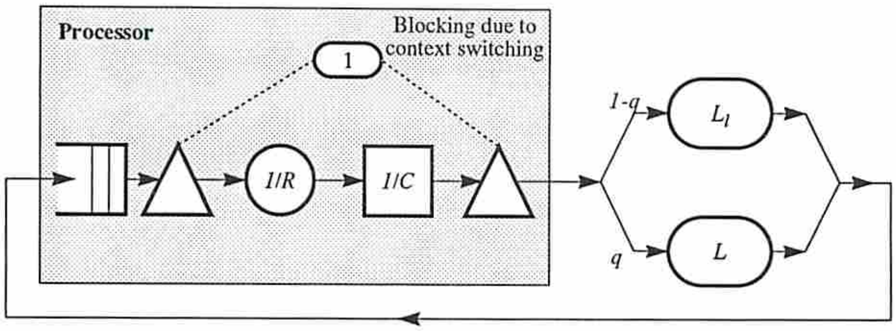
The prefetching timing should be made to minimize the unnecessary replacement of prefetched cache blocks. The replacement timing becomes more important for prefetch with large coverage factors (say ≥ 0.8). Based on the results shown in Figure 8, prefetch far ahead before the actual reference increases the chances to be replaced. The replacement effects also depend on the replacement algorithm used as well as the program behavior. In order to reduce these adverse effects, it is better to issue prefetching instruction within a few running cycles before the processor needs to reference the data.

4 Prefetching Effects on Multithreaded Processors

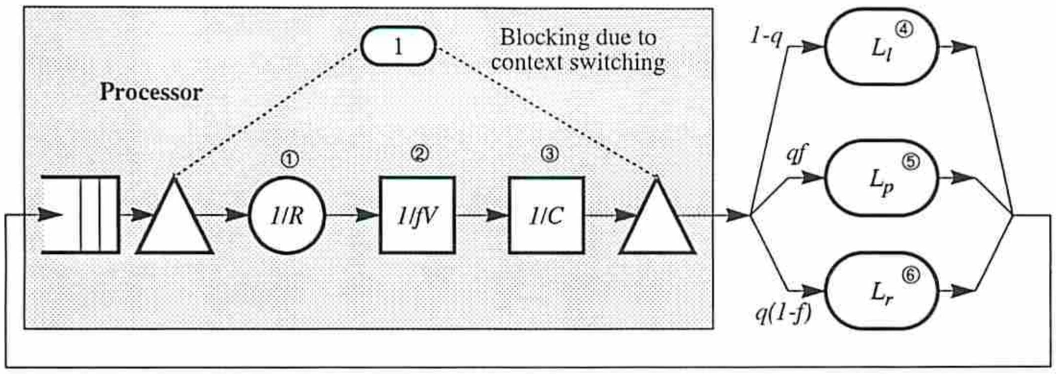
Through a queueing analysis, we model the combined effects of prefetching and multithreading on each processor. An approximate solution of the queueing model is obtained. We present the processor efficiency curves plotted from the approximate solution. Again, we first solve the problem without consider any side effects. Then in Section 4.3, we will study the network effects.

4.1 The Queueing Network Model

The interleaved execution of multiple contexts in each processor can be modeled by a closed queueing network. The model is shown in Figure 9a for a multithreaded processor without data prefetching. All the ready-to-run threads wait in a queue in the processor. The queueing discipline in FCFS. A multithreaded processor is modeled by several servers.



(a) Without prefetching.



(b) With prefetching.

Figure 9: The closed-loop queueing network models

The thread being executed first gets serviced at server (1). The service requirements of this server assumes an exponential distribution with a mean service rate $1/R$. When a remote memory is issued from the executing thread, another server (3) is used to switch the context and enable another thread for execution. The context switching rate is assumed a constant $1/C$. A blocking mechanism is used to ensure that only one thread gets serviced are a time. Two servers (4) and (5) are used to handle the local and remote memory access with a mean service rate $1/L_l$ and L_r, respectively. When the memory access is complete, the thread goes back to the queue. We assume that there is no dependency between different threads.

With prefetching, the multithreaded processor is modeled in Figure 9b. We need additional server 2 to model the prefetching processor with a constant service rate of $1/fqV$. The three servers inside the processor form a cascade of services. Another server (6) is used to reflect the prefetching effect with a prefetched access latency L_p such that $L_p < L_r$. A remote memory access is serviced by server 5 with a probability $1 - f$ and by server 6 with a probability f.

All access servers can assume some general distributions for their service times, based on the interconnection network and memory hierarchy used. All latencies L_l, L_r, and L_p used in our analysis are the mean values, derivable from their respective latency distributions. Note that there are at most N threads (contexts) in the queueing network. With one executing thread, the maximum queue length is $N - 1$ for holding all the waiting threads.

It is rather difficult to solve this 6-server queueing model directly. We will reduce the server complexities in the next section by going through some network transformations. The queueing model is developed with a few parameters. In case of adding other latency tolerating mechanisms, such as relaxed memory consistency and coherent caches, the model needs further refinement which is beyond the scope of this paper.

4.2 Approximate Solution from Model Transformation

The queueing model presented in Figure 1b can be simplified by combining the effects of the servers. We explain below the sequence of queueing network transformations. The purpose is to obtain exact solution of the transformed queueing model. The solution can be used to approximate the solution of the original queueing model.

First, the three servers (1), (2), and (3), inside the processor can be lumped together as a combined server with a mean service rate μ_1, as shown in Figure 10a. Because the mean service time:

$$\frac{1}{\mu_1} = R + fqV + C \tag{9}$$

we thus obtain:

$$\mu_1 = \frac{1}{R + fqV + C} \tag{10}$$

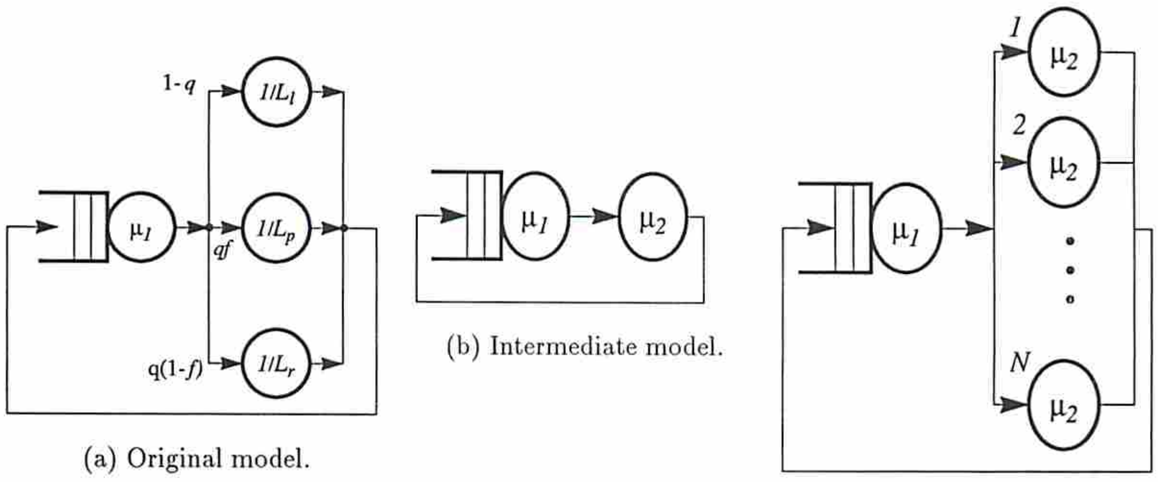
Secondly, the three servers (4), (5) and (6) are combined as a single memory-access server with a service rate (Figure 10b):

$$\mu_2 = \frac{1}{q[(1-f)L_r + fL_p] + (1-q)L_l} \tag{11}$$

for the fact that $1/\mu_2 = q[(1-f)L + fL_p] + (1-q)L_l$ is the expected service time for this combined server. This corresponds to the transformation from the original queueing network to the simple network shown in Figure 10b.

We assume the remote memory accesses are pipelined throughout the network and memory hierarchy. Considering the N threads in the queueing system, there are at most N overlapped memory requests that can be issued from each processor. Thus the memory server μ_2 can be replicated N times as in Figure 10c. The final queueing network obtained is identical to the *machine repair model* described in Lavenberg [14] which is based on the original work of Lassettre and Scherr [13].

The above four queueing networks (Figure 9 and Figure 10) can all be used to model the combined multithreading and prefetching activities in each processor. In the closed queueing system, at most N threads can coexist. Let p_i be the probability that exactly i threads are in the processor, for $i = 0, 1, 2, \ldots, N$ and $\sum_{i=0}^{N} p_i = 1$. In particular, p_0 represents the probability that the processor is idle executing no thread. Note that the summation $p_0 + p_1 + p_2 + \ldots + p_N = 1$.



(a) Original model.

(b) Intermediate model.

(c) Machine repair model.

Figure 10: The sequence of queueing network transformations from the original model to a machine repair model.

These probabilities can be computed from a finite Markov chain modeled by the state-transition diagram shown in Figure 11. In this diagram, the node with a label k represents the state that there are exactly k threads in the processor; one being executed by the μ_1 server and $k-1$ threads waiting in the queue. The remaining $N-k$ threads are suspended due to waiting for the remote memory access to complete through the μ_2 server in Figure 10c.

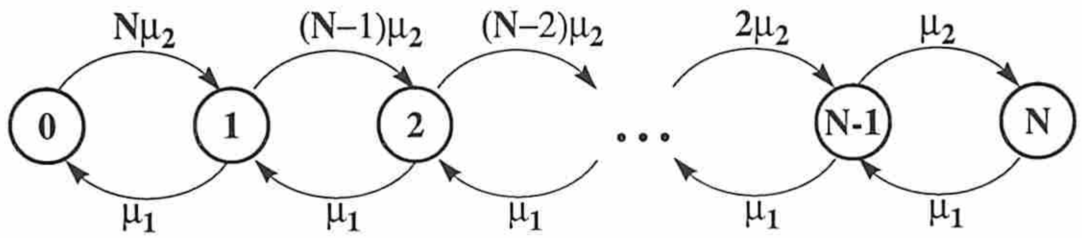


Figure 11: State-transition diagram of the machine repair model.

The above probabilities were computed by Lavenberg [14] as follows:

$$p_k = \frac{(\frac{\mu_1}{\mu_2})^{N-k}}{(N-K)!\sum_{i=0}^{N}\frac{(\frac{\mu_1}{\mu_2})^i}{i!}} \quad \text{for } k = 0, 1, 2, \ldots, N. \tag{12}$$

The *efficiency* (utilization) of the μ_1 server in Figure 10 can also be computed by the probability that there is at least one thread in μ_1 server, which can be expressed as:

$$\rho = 1 - p_0 = 1 - \frac{(\frac{\mu_1}{\mu_2})^N}{N!\sum_{i=0}^{N}\frac{(\frac{\mu_1}{\mu_2})^i}{i!}} = \frac{\sum_{i=0}^{N-1}\frac{(\frac{\mu_1}{\mu_2})^i}{i!}}{\sum_{i=0}^{N}\frac{(\frac{\mu_1}{\mu_2})^i}{i!}} \tag{13}$$

From Figure 9, the processor efficiency E_{mp}, is expressed as:

$$E_{mp} = \frac{\lambda}{\frac{1}{R}} = \frac{\lambda}{\mu_1}\cdot\frac{\mu_1}{\frac{1}{R}}, = \frac{\lambda}{\mu_1}\cdot\frac{R}{R + fqV + C} \tag{14}$$

where λ is the *arrival rate* of the server 1 in Figure 9. At steady state, this is the same arrival rates to server 2 and 3 in Figure 9 and to server μ_1 in Figure 10. The ratio $\lambda/\mu_1 = \rho$ is the efficiency of the μ_1 server.

Define a *normalizing factor*:

$$A_N = \sum_{i=0}^{N}\frac{(\frac{\mu_1}{\mu_2})^i}{i!} = \sum_{i=0}^{N}\frac{[\frac{q[(1-f)L+fL_p]+(1-q)L_l}{R+fqV+C}]^i}{i!} \tag{15}$$

Thus the efficiency ρ in Eq. 13 can be simplified as $\rho = A_{N-1}/A_N$. Substituting all these terms into Eq. 14, we obtain:

$$E_{mp} = \rho\frac{R}{R + fqV + C} = \frac{A_{N-1}R}{A_N(R + fqV + C)} \tag{16}$$

where A_n was defined in Eq. 15.

4.3 Analytical Performance Results

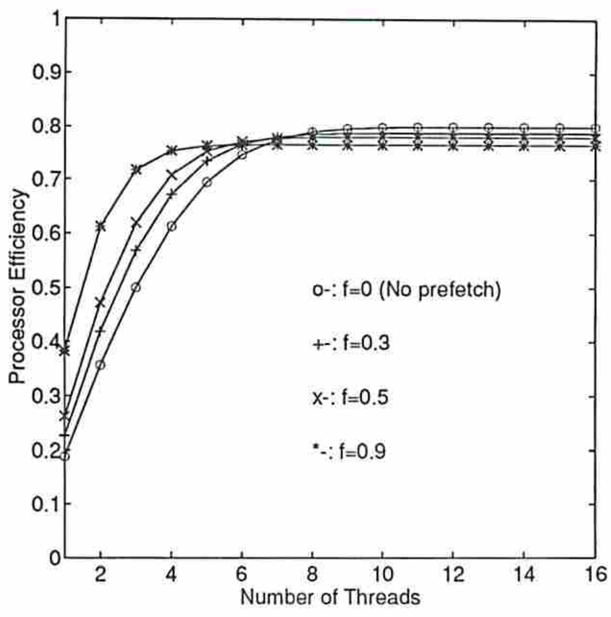
The processor efficiency E_{mp} obtained in Eq. 16 is plotted below as a function of the number of threads N, the coverage factor f, the prefetching overhead V, and the prefetched memory latency L_p from prefetching. Figure 12 plots Eq. 16 showing the effects of N, f, and V on E_{mp}, where we assume the same set of parameter as Table 1

With a few threads, i.e. $N \leq 4$, the E_{mp} increases steadily with increasing coverage of prefetching. With large number of threads, say $N \geq 6$, the E_{mp} becomes saturated and even drops in performance with increasing prefetching coverage.

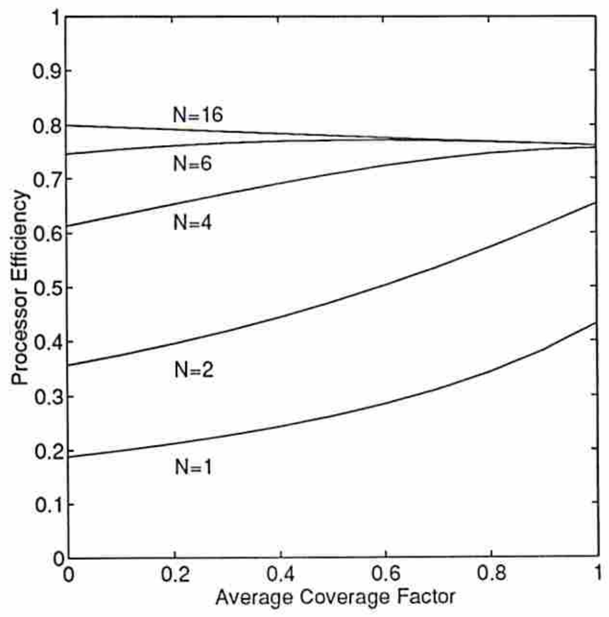
As we discussed in Section 3, the network traffic may increase with more potential for network contention or memory conflicts. Therefore, the increasing value of f may have an adverse effect on the memory latencies L_r and L_p. Figure 13 plots Eq. 16 with L_r and L_p are calculated according to Eq. 6. We plotted these curves with respect to two hypothetical cases: no adverse effect (cycle-labelled curves), and linear increase of L_r and L_p (cross-labelled curves), as the coverage factor f increases. These curves show that the similar adverse effects on E_{mp} as in the case of single-thread processor. However, this adverse effect becomes less significant when the degree of multithreading is large (> 6).

5 Verification with Petri Net Simulations

The approximate results from analytical modeling are verified below with equivalent Petri net simulation results. We first derive the state transition diagram from the embedded Markov chain. Numerical solution from the Petri net simulator are then compared with the queueing results.

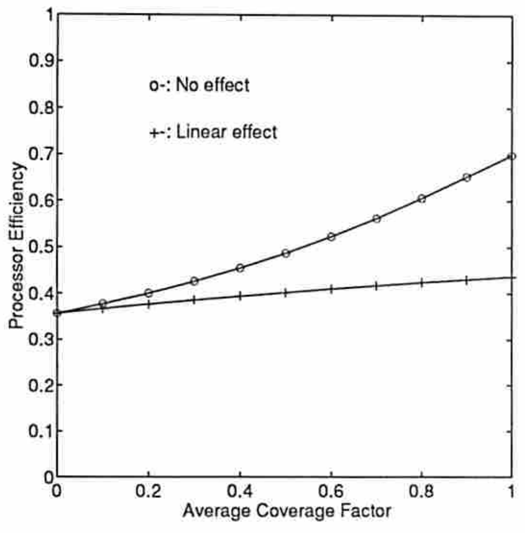


(a) E_{mp} as a function of N.

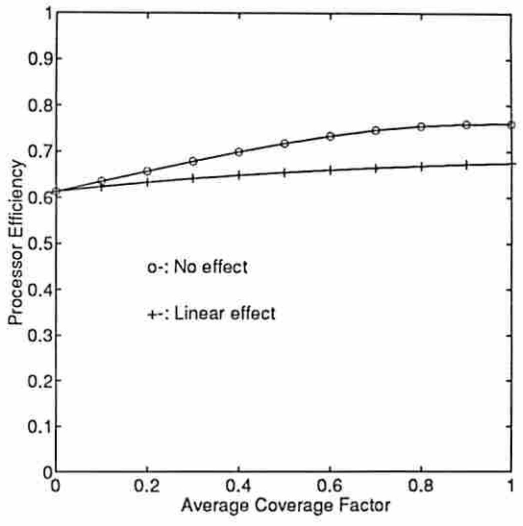


(b) E_{mp} as a function of f.

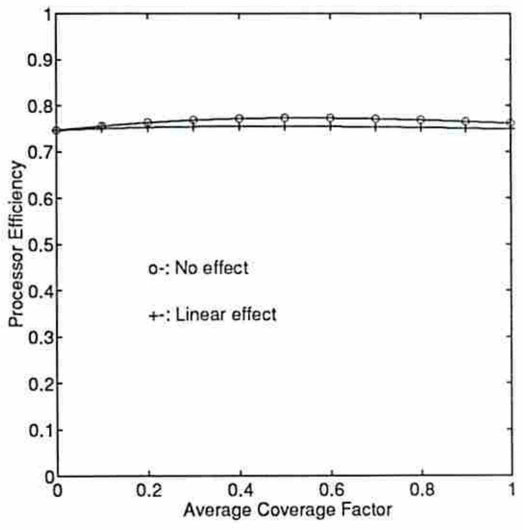
Figure 12: Effects of f, V, and N on the multithreaded processor efficiency E_{mp}



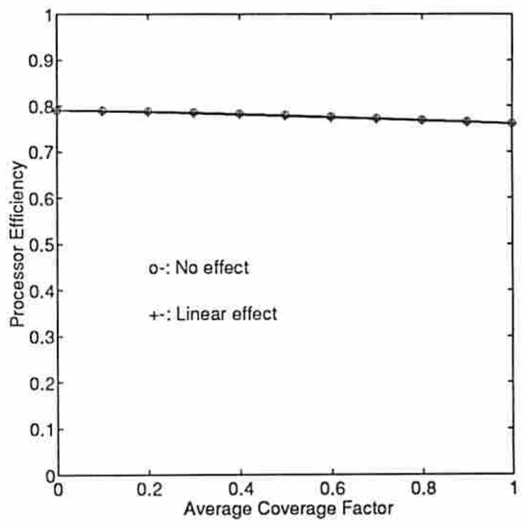
(a) $N = 2$.



(b) $N = 4$.



(c) $N = 6$.



(d) $N = 8$.

Figure 13: Adverse prefetching effects on the memory latency increase in a multithreaded multiprocessor under various traffic and prefetching coverage conditions.

5.1 Stochastic Timed Petri Nets

Petri Nets can model parallel computers with stochastic state transitions and timed services [17]. The queueing network presented in Figure 12 can be transformed to a equivalent *generalized stochastic Petri net* (GSPN). Figure 14 shows the GSPN model of a multithreaded processor with N threads and data prefetching; characterized by the same set of parameters f, V, R, C, L_l, L_r, and L_p.

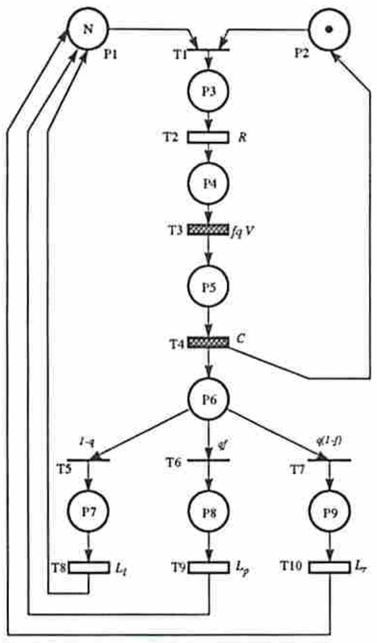


Figure 14: Petri Net modeling of the prefetching effects on a multithreaded processor.

The places are represented as circles "○" and transition as bars or boxes. Edges connect transitions to places and places to transitions. Tokens are denoted by dots "•". The thin bar indicates *immediate transitions* (T_1, T_5, T_6, T_7) without delay. The unshadded boxes (T_2, T_8, T_9, T_{10}) represent transition with *exponentially-distributed delays.* The shaded boxes (T_3, T_4) represent transitions with *constant delays.* The numbers beside the immediate transitions indicate the probability of firing when two or more immediate transitions are enabled at the same time by the same token. The number beside the exponential transition and deterministic transitions indicate the transition rates. Table 2 summarizes the meanings of the eight *places* $(P_1, P_2, \ldots, P_9)$ Table 3 specifies the eight *transitions* $(T_1, T_2, \ldots, T_10)$ used in this GSPN model.

Table 2: The Places in the Petri Net Model

Place	Meanings
1	Threads are ready when $N \geq 1$
2	The blocking mechanism for executing one thread at a time
3	A thread enters execution
4	A prefetching request is issued
5	Context switching from one thread to another
6	A thread just being switched out
7	Local memory requests in progress
8	Prefetched memory requests in progress
9	Non-prefetched memory requests in progress

5.2 State-Transition Diagram for Numerical Solution

The accuracy of the approximate solution of the queueing model can be verified with an numerical solution from numerical simulation of the equivalent GSPN model in Figure 14, we obtain the state transition diagram in Figure 15a. The state is represented by the 9-tuple $(b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9)$ where b_i is the number of threads at place i, $i = 1, \ldots 9$. Note that $0 \leq b_2, b_3, b_4, b_5, b_6 \leq 1$, $0 \leq b_1, b_7, b_8, b_9 \leq N$, and $b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7 + b_8 + b_9 = N$. N is the total number of threads in the network.

The shaded nodes in Figure 15a are *vanishing states*, while the remaining nodes are *tangible states.* Since there are at most one thread at P_3, P_4, and P_5, there are four types of

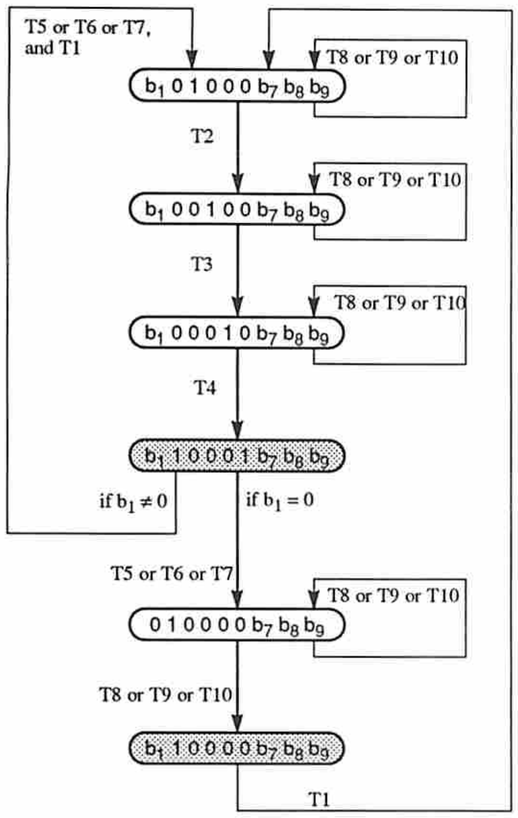
Table 3: The Transitions in the Petri Net Model

Transition	Type	Delay
1	Immediate	0
2	Exponential	R
3	Constant	fV
4	Constant	C
5	Immediate	0
6	Immediate	0
7	Immediate	0
8	Exponential	L_l
9	Exponential	L_p
10	Exponential	L_r

tangible states: $(b_1 00000 b_7 b_8 b_9)$, $(b_1 01000 b_7 b_8 b_9)$, $(b_1 00100 b_7 b_8 b_9)$, $(b_1 00010 b_7 b_8 b_9)$, and two types of vanishing states: $(b_1 10000 b_7 b_8 b_9)$, $(b_1 00001 b_7 b_8 b_9)$, for this model.

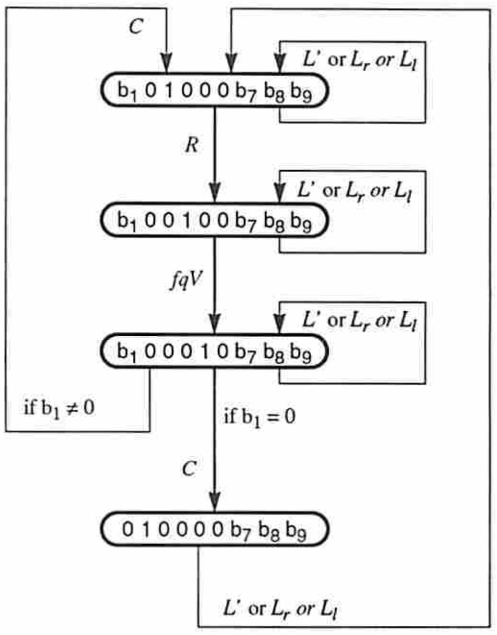
The transitions are labelled along the edges. The vanishing states can be ignored in a steady-state analysis. Figure 15b shows the reduced state transition diagram, after removing the vanishing states and associated transitions in Figure 15. The transitions are labelled by the corresponding firing delays.

What we obtained in Figure 15b is the state transition diagram that can describe the behavior of the queueing network in Figure 9b. The digit b_1 represents the number of threads in the queue. The digits b_3, b_4, b_5, b_7, b_8 and b_9 represent the number of threads at servers 1 to 6 in Figure 9. The digits b_2 and b_5 can be ignored since they are nonzero only in vanishing states. Therefore, the GSPN model in Figure 14 is equivalent to the queueing model shown in Figure 9.



T5 or T6 or T7, and T1
T8 or T9 or T10
$b_1\ 0\ 1\ 0\ 0\ 0\ b_7\ b_8\ b_9$
T2
T8 or T9 or T10
$b_1\ 0\ 0\ 1\ 0\ 0\ b_7\ b_8\ b_9$
T3
T8 or T9 or T10
$b_1\ 0\ 0\ 0\ 1\ 0\ b_7\ b_8\ b_9$
T4
$b_1\ 1\ 0\ 0\ 0\ 1\ b_7\ b_8\ b_9$
if $b_1 \neq 0$
if $b_1 = 0$
T5 or T6 or T7
T8 or T9 or T10
$0\ 1\ 0\ 0\ 0\ 0\ b_7\ b_8\ b_9$
T8 or T9 or T10
$b_1\ 1\ 0\ 0\ 0\ 0\ b_7\ b_8\ b_9$
T1

(a) State transition diagram



C
L' or L_r or L_l
$b_1\ 0\ 1\ 0\ 0\ 0\ b_7\ b_8\ b_9$
R
L' or L_r or L_l
$b_1\ 0\ 0\ 1\ 0\ 0\ b_7\ b_8\ b_9$
fqV
L' or L_r or L_l
$b_1\ 0\ 0\ 0\ 1\ 0\ b_7\ b_8\ b_9$
if $b_1 \neq 0$
if $b_1 = 0$
C
$0\ 1\ 0\ 0\ 0\ 0\ b_7\ b_8\ b_9$
L' or L_r or L_l

(b) Equivalent diagram after reduction

Figure 15: The state transition diagram for the Petri Net model.

5.3 Numerical Solution Using GreatSPN Package

The GreatSPN package (Version1.5) is a software simulator developed by Chiola, et al at Università di Torino [6, 5], for solving GSPN models with a given set of system parameters. It consists of a graphical editor and a analyzer for timed and stochastic Petri Nets. With the above GSPN model (Figure 14), we can apply the GreatSPN package to obtain the numerical solution. The values of all parameters used in the software experiments are the same as in Table 1.

Figure 16 compares the processor efficiency curve obtained from the queueing model

(Eq. 16) with the one obtained from the GreatSPN package. The two curves match very nicely. This proves the accuracy of the approximate solution, because the plot from the GreatSPN package is based on numerical solution for the same set of parameter values.

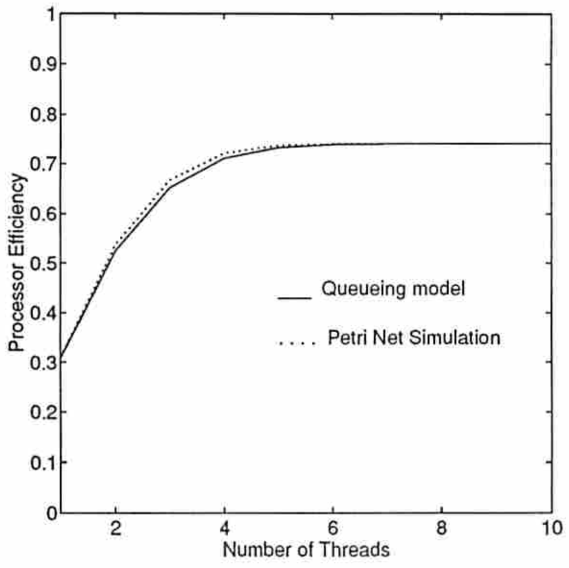


Figure 16: Comparision of the results obtained by two different methods.

6 Conclusions

The main contribution of this paper is to develop a queueing network model for multithreaded multiprocessors, supported by software-controlled data prefetching. This model is demonstrated useful to reveal the effects of multithreading and prefetching on the performance of scalable multiprocessors. To apply prefetching effectively, the number of threads handled by each processor should be maintained just a few, say $N \leq 6$.

The overhead of prefetching is usually low compared with the extent of latency hiding achieved. The adverse effect of prefetching on the increase of memory latency is also small, when the system is lightly loaded. When the system is overloaded with heavy traffic, prefetching is ineffective due to uncontrollable increase of the memory latency.

Our queueing model and the associated Petri net can be modified to analyze the effects of other latency tolerating mechanisms as well. For example, Chong and Hwang [7] have proposed the use of the generalized stochastic Petri net for modeling the performance of scalable multiprocessors, supported with various memory consistency models. We believe that the queueing model can be also extended to study the effects of hardware supported coherent caches, which is suggested for follow-on research challenges.

References

[1] A. Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[2] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Trabsaction on Computer Systems*, 7(2):184–215, May 1989.

[3] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proc. International Symposium on Computer Architecture*, pages 104–114, 1990.

[4] J. Archibald and J. L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transaction on Computer Systems*, 4(4):273–298, November 1986.

[5] G. Chiola. *GreatSPN 1.5 Software Architecture.* Dipartmento di Informatica, Universit'a di Torino, Torino, Italy, 1987.

[6] G. Chiola. A griphical petri net tool for performance analysis. In *Proc. of the 3rd International Workshop on Modeling Techniques and Tools for Computer Performance Evaluation*, pages 323–333, Paris, March 1987.

[7] Y. Chong and K. Hwang. Performance analysis of four memory consistency models for multithreaded multiprocessors. Technical Report 93-17, Dept. EE - Systems, USC, Los Angeles, CA, May 1993.

[8] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Proc. 4th Int'l Conf. Arch. Support for Prog. Lang. and OS.*, May 1992.

[9] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Ann. Symp. Computer Architecture*, June 1990.

[10] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proc. International Conference of Supercomputing*, pages 354–369, 1990.

[11] A. Gupta, J. Hennessy, K. Charachorloo, T. Mowry, and W. D. Weber. Computative evaluation of latency reducing and tolerating techniques. In *Proc. of Int'l Symp. Computer Architecture*, pages 254–263, Toronto, Canada, May 1991.

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

[13] E. R. Lassettre and A. L. Scherr. Modelling the performance of the OS/360 time-sharing option (TSO). In W. Freiberger, editor, *Statistical Computer Performance Evaluation*, pages 57–72. Academic Press, New York and London, 1972.

[14] S. S. Lavenberg. *Computer Performance Modeling Handbook.* Academic Press, Englewood Cliff, NJ, 1983.

[15] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The stanford Dash multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[16] K. Li and P. Hudak. Memory coherence in shared-memory systems. *ACM Trans. Computer Systems*, pages 321–359, November 1989.

[17] M. A. Marsan, G. Balbo, and G. Conte. *Performance Models of Multiprocessor Systems.* The MIT Press, Cambridge, MA, 1988.

[18] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, pages 87–106, 1991.

[19] R. H. Saavedra-Barrera and D. E. Culler. An analytical solution for a markov chain modeling multuthreaded execution. Technical Report UCB/CSD-91/623, Computer Science Division, UC Berkeley, March 1991.

[20] R. H. Saavedra-Barrera, d. E. Culler, and T. von Eicken. Analysis of multithreaded architectures for parallel computing. In *Proceedings of 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1990.

[21] W.D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proc. of International Symposium on Computer Architecture*, pages 273–280, 1989.