Reducing Multithreaded Frame Cache Miss Ratio by Prefetching and Working Frame Set Scheduling

Jongpil Choi*, Soonhoi Ha and Chushik Jhon
Department of Computer Engineering
Seoul National University, Korea

ABSTRACT In software-oriented multithreading execution model, the compiler identifies the remote accesses and performs fast context switches to hide remote access latency. In TAM model of execution, threads access the local memory through a data structure called "frame". This paper introduces a cache memory for frame structure and applies two techniques to reduce the cache miss ratio. One is a frame prefetching, which is based on the frame scheduling information, and the other is a changing frame execution sequences by the working frame set concept. Multithreading simulation is performed using benchmark programs and causes of cache misses are classified and analyzed. This paper shows the promising result that the frame prefetching based on scheduling information is very effective to reduce the cache miss ratio. But the effect of reordering the sequence of the frame execution is not so big than the prefetching.

1 Introduction

In a massively parallel computer, an unavoidable time delay incurs when accessing a remote memory location. To cope with this latency problem, latency hiding techniques such as coherent cache, prefetching, and relaxed memory models have been developed[1]. Multithreading is another attractive technique to reduce processor idling time and to raise processor utilization by fast context switching to a ready-to-run thread[1, 2]. A thread is defined as a sequence of instructions without suspension during execution.

In a software-oriented multithreading, the compiler identifies threads in high-level language programs and translates into object code according to multithreaded execution model[3, 4, 5]. Among various multithreaded execution models, this paper is based on the TAM(Threaded Abstract Machine) model[5] that uses compiler to identify locations of remote memory access, where thread switch occurs. In this model, a program is a collection of code blocks and a code block consists of thread code parts and message handler parts. A code block represents a function or a loop body. An invocation of a code block allocates a "frame" that is a structurally defined data storage to give an environment for local variables of a function or loop body execution. Thus, the dynamically allocated frames form a tree, rather than a stack, which is distributed over the processors, with frames as the basic unit of parallelism between processors.

To enhance the performance, program locality should be carefully considered and utilized by the memory hierarchy. Heap memory is reserved for the shared objects among code blocks and is accessed through split transaction. Since the compiler identifies the location of remote accesses, heap access could be tolerated by fast thread switching. Therefore, the coherent caches are not expected to be required for heap memory. On the other hand, locality appears in the frame structure because local variables are confined to a frame within a single processor. Since frame memory access is not tolerated, a frame is expected to be required to reduce the cache misses. This paper introduces a frame cache organization and proposes a frame cache prefetching method based on the frame scheduling information. Also, various frame scheduling methods are examined, having the scope of reducing cache miss ratio.

The most relevant research was performed by Kavi[6], in which an instruction cache, an operand cache and an I-structure cache are designed for ETS(Explicit Token Store) model of dataflow computation. The operand cache has two-level set

*is supported in part by Hyundai Electronics Industries, Korea. This project is funded in part by KOSEF(KOrea Science and Engineering Foundation) contract # 951-0910-126-2.

associativities. The first level sets are for superblocks assigned for each activated frame. The second level of associativity is used for accessing individual locations within a frame. They introduced cache memory for multithreading but data prefetching is not supposed.

Previous studies showed negative effects of multithreading on cache. Saavedra et al.[7] assumed that when N contexts exist, available cache size of one context becomes $1/N$. They postulated that the benefit of increasing the number of contexts is limited by cache miss effects. Also Weber and Gupta[8] showed negative effects of cache performance through simulation study assuming that a common cache is shared among many contexts.

Because instruction execution is sequential and branching within a loop is likely to be predicted, instruction prefetching is effective by sequential prefetching or history based prediction[9]. On the other hand, sequential data prefetching is not effective because data references are rarely sequential. Thus, data prefetching is usually based on the data locality and programmer's notification. It can be done by software or hardware. Hardware methods have a merit that compiler support or code change is not required. But more hardware resources are needed and unnecessary data prefetching increases data traffic about 2%[10]. In software methods, prefetching is done by static analysis of compiler and its conservative decision makes about 70% unnecessary prefetches. Also, prefetching instruction itself becomes execution time overhead about 28%[11]. In Klaiber[12], the compiler inserts prefetching instructions explicitly for data prefetching and have a fetch buffer for prefetched data storage to avoid cache access conflicts. Thus, reducing both unnecessary prefetching and execution time overhead is important in prefetching.

In the proposed scheme, prefetching decision based on frame scheduling information correctly fetches the next frame while conventional prediction schemes based on past execution history can not perfectly predict the next data. Moreover, since the prefetching is performed automatically and independently, no execution time overhead is incurred.

In section 2, simulation environments and benchmark program characteristics are shown. In section 3, the proposed prefetching scheme will be explained and experimental results are analyzed. In section 4, a new frame scheduling method is introduced and scheduling effects are discussed. In section 5, implementation issues will be illustrated, followed by conclusions in section 6.

2 Simulation Environments

Simulation is performed with the ATOM simulator[13], which is a multithreaded simulator based on the Berkeley TAM execution model. ATOM is an instruction level event driven simulator, which uses MIPS R2000/R3000 RISC microprocessor instruction sets. It uses system call primitives for multithreading execution model. ATOM is extended to incorporate a cache simulator for this experiment.

Benchmark programs are originally written in Id language and converted to intermediate language TL0. This TL0 code is translated to MIPS assembly code and executed on the ATOM simulator.

Programs for experiments are *mm, mmt* and *qs*. *mm* is a matrix multiplication program and accesses the heap memory for each element. *mmt* is also a matrix multiplication program but it divides the matrix into four by four sub-matrices. *qs* is a quick sort program and accesses the heap memory for each element.

The table shows the benchmark program characteristics and simulation input parameter values.

	mm	mmt	qs
input	50	50	100
instruction steps	11653K	3595K	672K
memory accesses	3733K	1535K	187K
num. of code blocks	6	9	9
avg. frame size(Bytes)	229	313	196
system call cost	1	1	1
network latency	1	1	1

Though the ATOM simulator supports multiple node, this experiment is concerned with the frame cache activity within a node.

3 Frame Cache Prefetching

In our model of computation, frames form an activation tree, distinguished from conventional stack structure. This tree structure enables the concurrent execution of frames for parallel processing[5]. Enabled frames are linked by a ready frame link pointer in a ready queue. When the current frame is suspended, then the next frame to be executed is picked up in order from this ready queue. Since the scheduling information is embedded in the frame structure with the ready queue, the next frame in the ready queue is prefetched into the frame cache by the prefetcher during the execution of current frame.

When the same cache is accessed simultaneously by the processor and the prefetcher, cache conflict occurs. Also some valid data of the current frame could be expelled by prefetching. It will increase cache misses and degrade performance. So, we divide the frame cache into two independently accessible regions, of which one region is accessed by the processor and the other region is accessed by the prefetcher. We call the latter a shadow cache. Figure 1 shows the shadow frame cache structure. During the current frame execution, thread codes access the current region while the prefetcher prefetches the next frame into the other region or shadow cache. The next frame sits in front of the scheduling queue. When frame switch occurs, roles of two cache regions are also switched. Thus, the processor and the prefetcher can access caches independently, which solves the cache conflict problem. Also, the shadowing cache does not expel the valid data of the current frame. In effect, the proposed structure is a special form of two-way set associativity. On the other hand, in case the same memory block is accessed simultaneously by the processor and the prefetcher, memory conflict occurs with one port memory. Memory conflict is solved by suspending the prefetcher during processor's memory access.

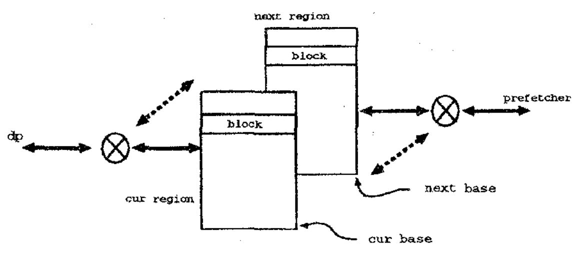

Figure 1: Shadowing frame cache

If all the next frame data is prefetched during the current frame execution, then the next frame execution will not experience any cache miss. Also when a frame is rescheduled, some resident data since the previous activation will reduce frame prefetching time and cache misses.

Suppose that the frame size is smaller than the cache size and no residual data from the previous execution remains in the cache. Then, the proposed scheme can be modeled with the following parameters.

Parameter	Description
$Q(n)$	total execution time of nth frame
$F_s(n)$	size of nth frame (number of blocks)
S	prefetching speed (number of blocks per unit time)
$T_f(n)$	prefetching time of nth frame, $T_f(n) = \frac{F_s(n)}{S}$
$I(n)$	busy processing time of nth frame without cache misses

$Q(n)$ is the sum of $I(n)$ and cache miss penalty. If $I(n) > T_f(n+1)$, the execution time of the current frame is so large that the next frame is successfully prefetched, and does not experience any cache miss. If $I(n) < T_f(n+1)$, the execution time of the current frame is shorter than the prefetching time of the next frame. In this case, during $Q(n)$ time, only $I(n) \cdot S$ blocks are prefetched and frame is switched. Cache hit ratio is simply assumed as $\frac{I(n)}{T_f(n+1)} = \frac{I(n)}{F_s(n)} \cdot S$[1].

F_s and I are determined by program characteristics. And parameter S depends on the hardware design. To increase the cache hit ratio, this formula indicates that the frame size should be smaller that the frame execution time per activation. In the TAM model, threads are fine-grained and the frame execution time is not big. If the grain size is large, then the frame execution time will increase. Since our experiments are based on the original TAM model, the experimental results are expected to be pessimistic in terms of the performance.

When managing two regions of caches, the cache coherence problem arises between two caches. Suppose that a certain frame is suspended and removed from the ready queue. Later that thread can be activated again and enqueued in the ready queue. When the frame reaches at the head of the ready queue, the frame is rescheduled and begins execution. If the cache region of the frame is different from that of the previous activation, all the cached data is lost and frame data should be reloaded. Besides the overhead of reloading, there is a fundamental problem of cache coherency if the previously modified data were not properly updated to the memory. To prevent this, we enforce that the working cache

[1]This approximation is the ratio of the frame size and the preloaded data size.

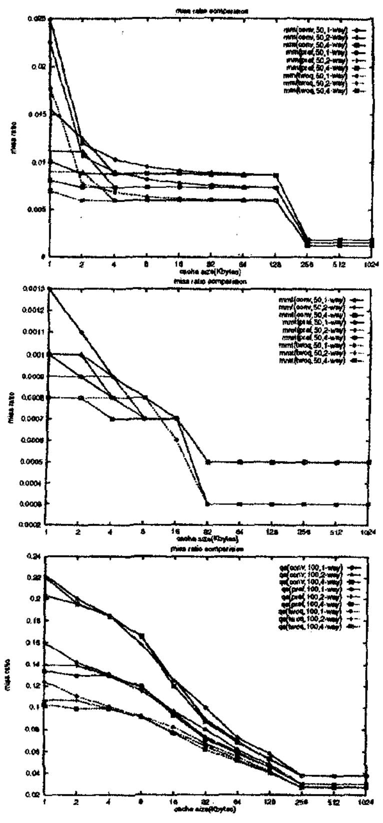
Figure 2: Various prefetching methods

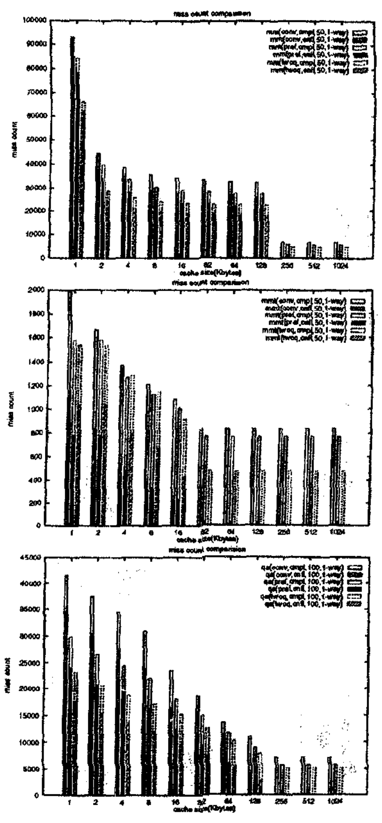
Figure 3: Compulsory and conflict misses

region of a frame is fixed through all activations. Therefore, a frame is assigned a cache region when the frame is first allocated. We classify a frame as odd or even so that odd frames are assigned to one region and even frames to the other region.

Because the ready queue keeps the sequence of the runnable frames, we may have more than two independent cache regions, and prefetch more than one frame. In this case, each region should have an independent access to the memory. Frame size and hardware complexity shows that having too many regions is not effective in performance enhancement.

Figure 2 shows the results for three caching techniques, *conv*, *pref* and *twoq*. Each method is executed with varying the associativity of each region of cache to 1, 2 and 4. For each case, cache miss ratio versus cache size is plotted.

conv represents a conventional method and the cache miss ratio is the highest. In *pref* method, a frame is classified into odd and even. While an even frame is executed, only an odd frame can be prefetched into the shadow cache. When an odd frame is in execution, only an even frame can be prefetched. In this method, if the class of the next frame is the same as that of the current frame, prefetching does not occur. This method shows the middle level of cache miss ratio. In *twoq* method, two ready queues are prepared for odd and even frames. When an even frame is currently in execution, the next frame to be prefetched is obtained from the odd ready queue. During execution of an odd frame, the even ready queue is searched for the next prefetched frame. This method shows the lowest cache miss ratio.

The average miss ratio of these methods are calculated for the direct map(associativity 1) cache and 11 different cache sizes. Cache miss ratio is reduced by about 17.4% in *pref* method and by about 31.3% in *twoq* method compared with *conv* method. As cache size increases, cache miss ratio reduces, but at some point, cache miss ratios do not change. This means that the cache size is large enough to load all the frame data, and increasing cache size does not increase the cache miss ratio, which admittedly implies that the benchmark programs are too small.

This figure also shows that when the cache size is

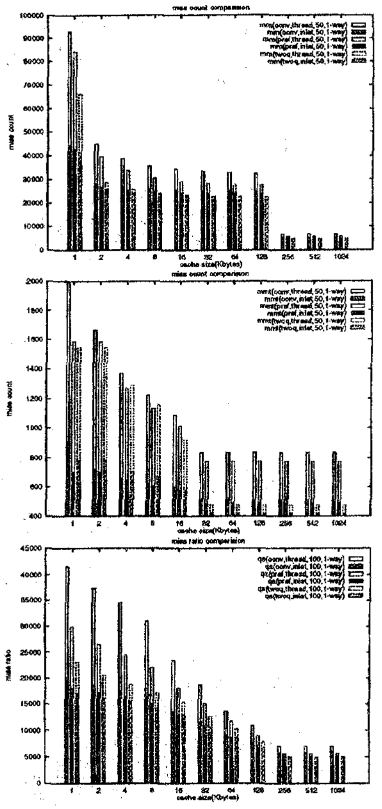

Figure 4: Location of cache misses

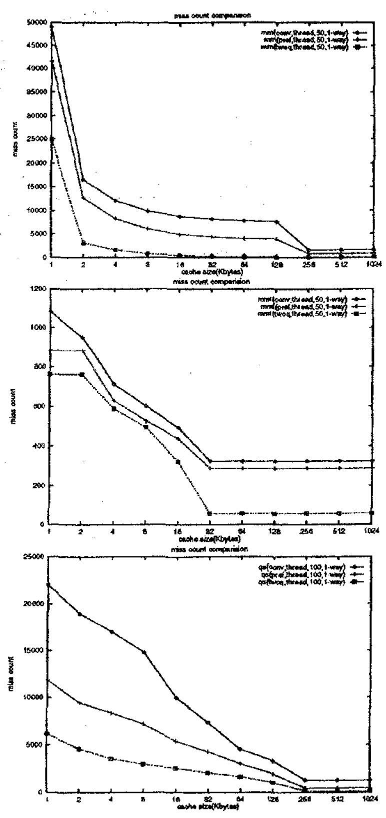

Figure 5: Thread part cache misses

small, increasing the cache associativity is helpful for reducing cache miss ratio. But when the cache size is large, increasing cache associativity is no more helpful than prefetching.

Cache misses are classified into compulsory misses and conflict misses. Figure 3 shows the cause of misses in bar chart. Higher empty bars are compulsory miss parts and lower patterned bars are conflict miss parts. As cache size increases, conflict miss reduces, and at some point(*mm* 256, *mmt* 32, *qs* 256), conflict disappears. Compulsory miss is independent to cache size, but depends on the prefetching methods. The benchmark programs contains small number of code blocks, do repetitive jobs with large input values. As a result, the large portion of cache misses is conflict miss. But decreasing the input value will show the obvious compulsory miss difference between prefetching methods.

In the previous section, it is mentioned that a code block consists of thread codes and message handling(*inlet* in TAM) codes. Figure 4 shows the place where cache misses occur. Higher empty bars are cache misses at thread codes and lower patterned bars are cache misses at message handling codes. Cache miss reduction at thread part is significant, which is our main objective. Figure 5 represents only thread parts of cache misses and shows this more clearly. Message handler intervenes the current frame execution. And data referenced within message handler do not preserve the principle of data locality. Therefore, we propose the dual processor node architecture to be illustrated in section 5.

4 Working Frame Set Scheduling

In the previous section, it is assumed that the frame execution sequence is the LIFO(Last-In-First-Out) style, which is the lately scheduled frame is executed firstly. Generally, the LIFO style requires smaller memory storage for frame region than the FIFO(First-In-First-Out) style. Also, it is expected that the LIFO style has more advantages in keeping valid cache blocks when the frame is reactivated.

One more frame scheduling method is that of using the cache information. The frame that is already loaded in the cache can have a priority in frame scheduling. LIFO and FIFO are well known scheduling methods and the third method is explained detailed and discussed. Figure 6 shows the frame mapping from local memory to frame cache. The frame invocation sequence is `f1-f2-f3-f4-f5-f1-f6`. `f5` is mapped to the same place of `f1`, and `f6` is mapped to that of `f4`. When the frame `f1` is activated secondly, `f1` experiences many cache misses since `f5` has expelled `f1`'s valid cache blocks before. In a new scheduling methods, `f1` is executed before `f5` to avoid these misses.

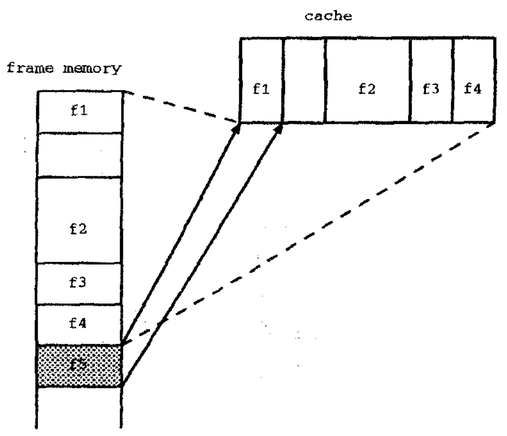

Figure 6: Local frame memory to cache mapping

We define the "working frame set" as a set of frames that have valid blocks in cache currently, and "pending list" as a list of scheduled frames which have no valid blocks in the cache. When (`f1-f2-f3-f4`) is a current *working frame set*, and `f5-f1-f6` frame scheduling requirements are generated while the current *working frame set* is in execution, coming frames are connected to the *pending list* and the *ready queue* as follows.

```
Working frame set:(f1-f2-f3-f4)
Pending list:(f5-f6)
Ready queue:(f1)
```

`f5` is connected to the *pending list*. `f1` is connected to the *ready queue* and executed before f5 since it has valid blocks in the cache as noted in the *working frame set*. `f6` is connected to the *pending list* since it is not a member of the *working frame set*. After the completion of `f1`, `f5` is fetched from pending list and be a member of the *working frame set* since the *ready queue* is empty. Now, members are changed as follows.

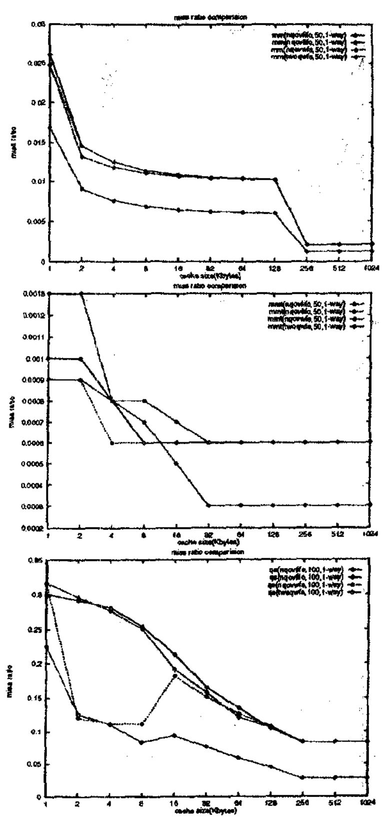

Figure 7: Various frame scheduling

```
Working frame set:(f5-f2-f3-f4)
Pending list:(f6)
Ready queue:(f5)
```

In LIFO or FIFO, the frame execution sequence is determined and fixed by the program behavior and the scheduling strategy. On the contrary, the *working frame set* method uses the run-time cache information to reduce miss ratio, and the sequence is not fixed.

Figure 7 shows the experimental result of LIFO, FIFO, WFS(Working Frame Set), and *twoqwfs* schedulings. The *twoqwfs* is a mix of WFS and *twoq* prefetching method. The scheduling overhead is ignored and only misses of real codes are measured. The miss ratio of the LIFO is smaller than FIFO, but the difference is not so big. The WFS method is slightly better than LIFO or FIFO in small cache sizes of *mm* and *mmt*, and shows a large difference in small cache sizes of *qs*. But when the cache size becomes larger, the difference between scheduling methods is not obvious. The *twoqwfs* method shows much better results of miss ratios. It is not by the effect of WFS scheduling, but by the effect of *twoq*

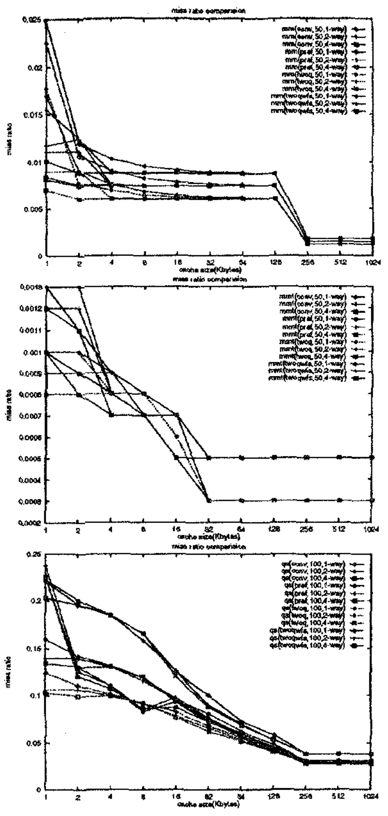

Figure 8: Prefetching and WFS scheduling

prefetching.

Figure 8 compares the *twoqwfs* with prefetching methods. In *mm* and *mmt*, twoqwfs has no difference from the prefetching method. In *qs*, *twoq* is better than the *twoqwfs*. The *twoqwfs* is better than the *twoq* only in 8KBytes cache size and becomes similar to the *pref* in large cache sizes.

The WFS method shows sensitive miss ratio changes in the variance of cache sizes, and it is also sensitive to program characteristics. When the cache size, program behaviors are matched with WFS scheduling method, the prefetching with WFS scheduling is better than the prefetching only methods, but in other cases, the prefetching with WFS scheduling is not better than the prefetching only method.

5 Implementation Considerations

Figure 9 shows the proposed node architecture. dp is the data processor which executes thread codes. network control manages message input and output, frame cache prefetcher performs prefetching operation of a next frame, and memory control controls accesses to the local memory. Local memory is partitioned functionally for thread codes, message handling codes, run-time system codes, frame area, and heap area. sp is the system processor that has three purposes: executing message handler codes in user program, executing runtime system codes, managing network control and frame cache prefetcher. There are two mail boxes P mail box and N mail box. P mail box is used for the communication of dp and sp, and N mail box is used for network interface. ds box of P mail box is used to notice the system calls to sp, and sd box is used for thread scheduling.

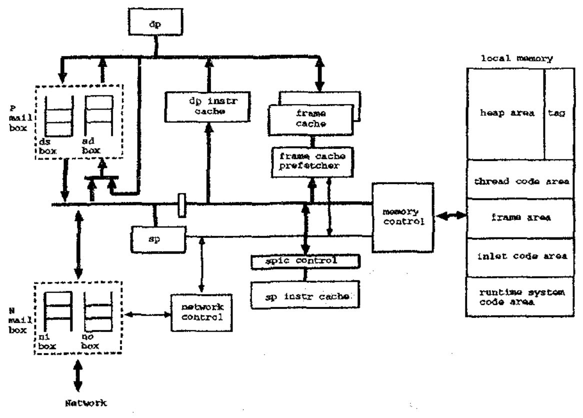

Figure 9: Processor node architecture

In the proposed architecture, message handlers executed in sp do not allocate the cache block in case of cache miss not to expel the valid data. We assume that the execution time of a program is determined by execution time of dp. In case sp becomes the bottleneck, prefetching is no helpful to improve the performance.

6 Conclusions

Multithreading appears as an attractive solution to hide long memory latency of massively parallel computers. The frame structure is generally used in software-oriented multithreading to store local variables and becomes an unit of parallel scheduling.

In this paper, a cache memory for frame data is introduced. To reduce cache miss ratio, a new prefetching method based on frame scheduling information is proposed and a new frame scheduling method based on dynamic cache information is examined. Frame cache is divided into two independently accessible regions, *current region* and *next region*. The *current region* is accessed by the currently activated frame and the *next region* is accessed by the prefetcher while the current frame is in execution. When the current frame is suspended, the roles of two regions are switched.

Simulation experiments based on multithreaded execution model are performed. Experimental results show that cache miss ratio is reduced by about 20-30% compared with conventional cache scheme and the effect of changing frame scheduling method is not so big than the prefetching. Based on the experiments, a new node architecture is proposed, in which the data processor will be more benefited by the proposed prefetching method. We are currently evaluating the effectiveness of this two-processor node architecture.

References

[1] A. Gupta, J. Hennessy, K. Gharachorloo etc. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proc. 18th Annual Int'l Symp. on Comp. Arch.*, pp.254-263, 1991

[2] R. Boothe. Evaluation of Multithreading and Caching in Large Shared Memory Parallel Computers. *PhD thesis UC Berkeley, UCB/CSD-93-766*, Jul. 1993

[3] G. Papadopoulos, D. Culler. Monsoon: an Explicit Token-Store Architecture. In *Proc. 17th Annual Int'l Symp. on Comp. Arch.*, pp.82-91, 1990

[4] R. S. Nikhil and Arvind. Can dataflow subsume Von Neumann computing? In *Proc. 16th Annual Int'l Symp. on Comp. Arch.*, pp.262-272, 1989

[5] D. E. Culler, S. C. Goldstein, K. E. Schauser, T. Eicken. TAM - A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, Jun. 1993

[6] K. M. Kavi, A. R. Hurson, P. Patadia, E. Abraham, P. Shanmugam. Design of Cache Memories for Multi-Threaded Dataflow Architecture. In *Proc. 22th Annual Int'l Symp. on Comp. Arch.*, pp.253-264, 1995

[7] R. Saavedra-Barrera, D. Culler, T. Eicken. Analysis of Multithreaded Architectures for Parallel Computing. *6th ACM Symp. on Parallel Algorithms and Architectures*

[8] W. Weber and A. Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proc. 16th Annual Int'l Symp. on Comp. Arch.*, pp.273-280, 1989

[9] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3) pp.473-530, Sep. 1982

[10] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. 17th Annual Int'l Symp. on Comp. Arch.*, pp.364-375, 1990

[11] D. Callahan, K. Kennedy, A. Porterfield. Software prefetching. *The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.40-52. 1991

[12] A. Klaiber, H. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proc. 18th Annual Int'l Symp. on Comp. Arch.*, pp.43-53, 1991

[13] Jinsoo Kim, Soonhoi Ha, Chushik Jhon. Building a Simulation environment for Multithreaded Parallel Architectures. *Korean Information Science Society Proceedings of Parallel Processing System*, Vol.6,No.1, May 1995