A loop partition technique for reducing cache bank conflict in multithreaded architecture

C-C. Wu
C. Chen

Indexing terms: Loop partition, Multibank cache, Bank conflict, Multithreaded processor, Compile time

Abstract: Parallel multithreaded architectures take advantage of the ability to execute more than one thread simultaneously on a single chip at low synchronisation and communication costs and high hardware resource utilisation. However, a high bandwidth cache, such as a multibank cache, is especially critical to serve memory accesses issued at the same time from different threads. To prevent bank conflicts of multibank cache from seriously degrading system performance, a loop partition method is proposed to reduce or even eliminate bank conflicts. The partition allows each thread access to certain bank modules and prevents any two from accessing the same bank module. The method neither slows down the clock rate nor increases the array subscript expression complexity. The performance gains of the bank-conflict-free loop partition approach are shown in simulation results.

1 Introduction

In the near future, improvements in semiconductor technology will allow multiple high performance floating point units and several megabits of memory to reside on a single chip [1]. This trend is stimulating the design of multithreaded processors [2–6]. In multithreaded architectures, whenever a long latency operation take places in a running thread, the system immediately switches out the thread and selects one of the waiting threads for execution. Thus, long latencies can be hidden by executing threads, which improves utilisation of system resources [2].

There are two kinds of multithreaded processors: concurrent multithreaded processors (CMPs) and parallel multithreaded processors (PMPs) [4]. CMPs execute only one thread at any time. They rapidly switch to one of the waiting threads when the running thread encounters such situations as data absences or synchronisation failures, which cause long processor latencies. The success of a CMP design depends on providing a fast context switching mechanism to efficiently overlap long latencies. Architectures possessing this ability include HEP [2], Sparcle [4], Tera [5] and *T [6].

PMP architectures, on the other hand, are capable of executing more than one thread at the same time [4]. Although, they usually require more expensive hardware and greater design complexity, they can hide latencies at the instruction level rather than at the thread level. When an instruction from a thread cannot be issued, because of either a control or data dependence within the thread, an independent instruction from another thread is executed instead. Thus, the advantage of PMPs is that they enable greater hardware utilisation because the functional units in this processor are shared by all the parallel running threads. Some examples of PMP architecture are described in [1–3]. It is expected that this type of processor will become one of the most popular forms of single processor design because of its better resource utilisation. In addition, because multibank caches are often included in PMP architectures to provide high bandwidth memory subsystems [1], in this paper the PMP architecture will be studied with a multibank cache.

A good loop partition technique is very important for both multiprocessor systems and PMP architectures because it can exploit more parallelism from a program. Although there have been many partition techniques focused on exploring cache locality, the features of multibank caches have been ignored [9]. One multibank cache feature is that several memory accesses can be served simultaneously. However, if a bank module receives more than one access at the same time, there will be a bank conflict. Bank conflicts severely degrade system performance. Therefore, reducing cache–bank conflicts is very important in partitioning loops in PMP architectures. In this paper, we propose a loop partition method that improves system performance according to special considerations on multibank caches. The loop is partitioned on the basis of multibank cache data allocation characteristics, so as to reduce bank conflicts and thus enhance the system performance. It was found that some important properties help partition the loop without bank conflicts. According to simulation results, the ideal case of this method can, at best, speed system performance by a factor of 1.9 over other methods.

2 Basic configuration and issues

The PMBC (parallel multithreaded multibank cache) processor is a PMP architecture with a multibank cache, as shown in Fig. 1. This architecture has two distinguishing features: (i) several threads executed

IEE Proceedings online no. 19960007

Paper first received 28th February 1995 and in revised form 22nd September 1995

Department of Computer Science & Information Engineering, National Chiao Tung University, Hsinchu, Taiwan, Republic of China

simultaneously in the same chip can access the single shared cache at the same time, and (ii) the multibank cache allows more than one memory access to be served at the same time. A PMP comprises several logical processors (*LP*), each of which can execute one individual thread. An LP is composed of a thread slot, several functional units, and a register set. Each thread slot has its own program counter and decoder unit. Instructions are fetched from the instruction cache into the thread slot and then scheduled into the functional units after decoding. Data are fetched from or written into the corresponding register set. There are many register sets, which provide for fast context switching among running threads, blocked threads and ready threads. Synchronisation and communication between threads are performed through the shared memory.

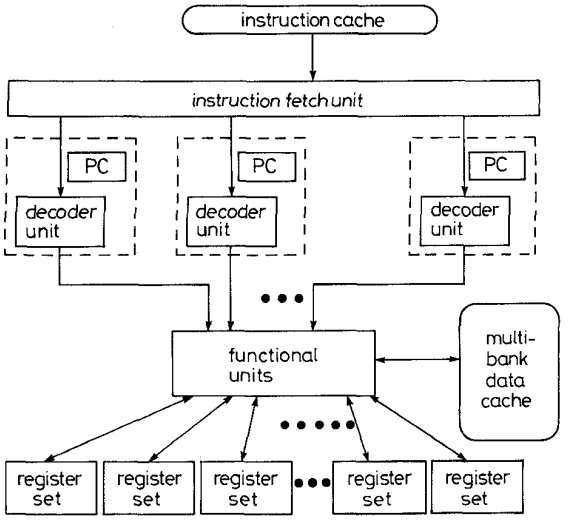

Fig. 1 *Overview of PMP architecture*

Naturally, since there will be many simultaneous memory accesses from multiple threads, the PMP needs a high bandwidth cache subsystem, otherwise the cache subsystem will become a bottleneck and severely degrade system performance. The multibank cache is a popular way of providing high bandwidth for memory accesses [1,3]. If the data requested by several threads are allocated to different bank modules, all the memory access requests can be granted simultaneously. On the other hand, a bank conflict will occur if more than one memory request for a single bank module arrives at the same time, or if the bank module is busy when the memory request arrives. Whenever a bank conflict occurs, one of the requests is arbitrarily granted to access the bank module. The other requests are blocked and must wait for the next arbitration. The memory access time increases in proportion to the time these blocked requests must wait. That is to say if the average memory access time is increased, the total program execution time will also be increased. If the bank conflict ratio can be reduced, or even eliminated, obviously, system performance would be improved. Since a large percentage of the execution time is spent in executing the loop body, the intention of this paper is to propose a loop partition approach that will reduce, or even eliminate bank conflicts.

The loop is partitioned into threads such that each thread can access specific bank modules but no two threads can access the same bank module. The method employs a software approach that does not raise the complexity of the array subscript expression. It is based on data allocation characteristics of the multibank cache, which has different properties and solution techniques from those either in multiprocessor systems or in multibank memory systems [7–15].

Many research projects on loop partitions used in multiprocessor systems have been proposed [7–11]. In multiprocessor systems, each processing element has its own cache, and copies of data are stored in several different caches. Therefore, there are cache coherence problems in multiprocessor systems. Basically, the purposes of these projects have been to reduce synchronisation overhead and communication traffic between the memory modules in the shared or distributed memory system. The principles of these loop partitions exploit the locality of the data caches, instead of the memory modules. Thus, it does not matter whether the memory is multibank or not. In a PMP architecture, however, although multiple simultaneous running threads shares a single cache, there is no cache coherence problem because there is only one copy of the data. Reducing bank conflicts is the special goal of loop partition for PMP architecture.

Many studies have also been done on multibank memories for vector machines [12–15]. In a vector machine, when a memory request from a single instruction stream is issued to the memory, an interleaving access occurs. This request includes the starting memory address, the access stride, and the access length. Ideally, an interleaving access will fetch one datum from each bank module unless there is bank conflict. The goal of these studies has been to reduce bank conflicts by restructuring data allocation. Such a reduction can be achieved using hardware or software approaches. Although the hardware approach increases the length of the critical path and affects the clock rate, the software approach results in a very complex array subscript expression and a long run time for subscript calculation. In PMP architectures, however, multiple simultaneous memory requests are issued from many streams to a shared cache. In addition, the addresses of these requests may have no regularity among them. Therefore, a loop partition method in terms of the data allocation features in the multibank cache is proposed. This method does not increase the subscript expression complexity or affect the clock rate.

3 Single dimension array, single loop structure

In this Section, we consider the following code executed in a PMBC model processor according to the subscript expression

for j =1 to M do

... $A[a^*j]$ ...

For convenience, we assume that the number of LPs in the PMBC processor is P, $P = 2^p$, and that the number of cache banks is B, $B = 2^b$. Data allocation adopts low order interleaving and the datum at address d will be allocated to bank r if r equals (d mod B). Suppose that the accessed array elements are allocated to exactly H bank modules, where $H \geq P$ and $H/P = D$. If we partition the code into P threads such that each thread references exactly D different bank modules, there will be no bank conflict. In the following lemma, we determine the sufficient condition for accessing exactly D bank modules.

Lemma 1: Assume that $B = m^*P$, where $B = 2^b$, $P = 2^p$, and both m and a are integers,

Let $R = \{r \mid r = a^*j \bmod B$ and j is an integer$\}$,

$R_k = \{r \mid r = a^*j \bmod B$ and $j = (k + 1) + s^*P$, $s \geq 0$ and s is an integer$\}$, k is a constant, $0 \leq k < P$,

$U = \{R_k \mid 0 \leq k < P\}$

If (i) a is odd, or

(ii) a is even and there exist integers a' and q, $a' > 0$, $q > 0$, such that $a = a'^*q$ and $a'^*q^*P = B$,

then U is a partition of R.

Proof: (1) Trivially, the union of all R_k equals R, where $0 \leq k < P$.

(2) Let $a \bmod B = a'$. If r belongs to both R_g and R_h, $0 \leq g, h < P$, then $r = a^*((g + 1) + s^*P) \bmod B = a^*((h + 1) + s'^*P) \bmod B$, where s and s' are nonnegative integers.

(i) a is odd. Let $s_1 = s \bmod m$, $s_1' = s' \bmod m$. If $a \bmod B = a'$, then $a'^*((g + 1) + s_1^*P) \bmod B = a'^*((h + 1) + s_1'^*P) \bmod B$. This implies $((g - h) + (s_1 - s_1')^*P) \bmod B = 0$. Since $1 - P \leq g - h \leq P - 1$ and, $1 - m \leq s_1 - s_1' \leq m - 1$, we have $(g - h) + (s_1 - s_1')^*P < B$, which implies $g = h$ and $s_1 = s_1'$. Hence, R_g and R_h are disjoint if $g \neq h$.

(ii) a is even, $a = a'^*q$, and $a'^*q^*P = B$. If $v_1 = s \bmod q$, $v_1' = s' \bmod q$, then $(a'^*(g + 1) + a'^*v_1^*P) \bmod B = (a'^*(h + 1) + a'^*v_1'^*P) \bmod B$. Because $a'^*(1 - P) \leq a'^*(g - h) \leq a'^*(P - 1)$ and $a'^*(1 - q)^*P \leq a'^*(v_1 - v_1')^*P \leq a'^*(q - 1)^*P$, we have $a'^*(1 - q^*P) \leq (a'^*(g - h) + a'^*(v_1 - v_1')^*P) \leq a'^*(q^*P - 1)$. Hence, $a' - B \leq a'^*(g - h) + a'^*(v_1 - v_1')^*P) \leq B - a'$. Comparing with the previous expression: $(a'^*(g - h) + a'^*(v_1 - v_1')^*P) \bmod B = 0$, we have $g = h$ and $v_1 = v_1'$ because $-B < a' - B < 0$ and $0 < B - a' < B$. That is, R_g and R_h are disjoint if $g \neq h$.

(3) According to the results of (1) and (2), U is a partition of R. <Q.E.D.>

In lemma 1, the meanings of both a and j are the same as those in the code segment mentioned above. The element r, of the sets R and R_k, represents which bank module will be allocated, where the subscript k of R_k indicates the identification of the logical processor. The following lemma describes an alternative way of calculating how many bank modules will be accessed when the coefficient a is even.

Lemma 2: Assume that both B and a' are even and that n is an integer. If $2 \leq a' \leq B/2$, $a'^*n = B$, j and j' are nonnegative integers, then (1) the set of the remainders of $(a'^*j \bmod B)$ is identical to the set of the remainders of $((B - a')^*j' \bmod B)$; (2) the remainders of $(a'^*j \bmod B)$ have exactly n different values.

Proof: (1) If $a'^*j \bmod B = (B - a')^*j \bmod B$, then $a'^*j - (B - a')^*j' \bmod B = 0$. Hence, we have $a'^*(j + j') \bmod B = 0$. Since $a' \bmod B \neq 0$, $(j + j') \bmod B = 0$. In addition, since $2 \leq j + j' \leq 2^*n$ and $2 \leq n \leq B/2$, we have $2 \leq j + j' \leq B$, which implies $j' = B - j$.

(2) Let $j = s_1 + s_2^*n$, where $0 \leq s_1 < n$ and s_2 is an integer. We find that $a'^*j \bmod B = (a'^*s_1 + a'^*s_2^*n) \bmod B = (a'^*s_1) \bmod B$. Since $0 \leq s_1 < n$, we have $0 \leq a'^*s_1 < a'^*n = B$. Hence, there are n different remainders for $(a'^*j \bmod B)$. <Q.E.D.>

The partitions of the remainders for a^*j and $(B - a)^*j$ are the same, which implies that the number of banks that will be accessed can be calculated by either a or $(B - a)$. When a is odd, a different property will be obtained as shown below.

Lemma 3: Assume that a is odd and that the processor is a PMBC model. Let $Z = \{A[a^*(C + w)] \mid 0 \leq w \leq B - 1$, C is an arbitrary constant and w is an integer$\}$, if $x \neq y$, both x and y belong to Z, then x and y will not be allocated to the same bank module.

Proof: If $A[a^*(C + s_1)]$ and $A[a^*(C + s_2)]$ are allocated to the same bank module, where $0 \leq s_1, s_2 \leq B - 1$, then $a^*(C + s_1) \bmod B = a^*(C + s_2) \bmod b$. Because a is odd and B is even, $a \bmod B \neq 0$. Hence, $C + s_1 \bmod B = C + s_2 \bmod B$, and we get $s_1 - s_2 \bmod B = 0$. Since $1 - B \leq s_1 - s_2 \leq B - 1$, we have $s_1 = s_2$. <Q.E.D.>

We know from lemma 3 that all the B bank modules will be accessed individually by consecutive B memory references. The following partition rule which precludes bank conflicts can be obtained directly from the above three lemmas.

Theorem 1: Let $A[a^*j]$ be a data array in the following loop structure:

for $j = 1$ to M do

... $A[a^*j]$ ...

Assume that $(a \bmod B) = e$ and $a' = \min(e, B - e)$. If (1) a is even, $a' > 0$, and $B \bmod (a'^*P) = 0$, or (2) a is odd, we can partition this loop into P threads for our PMBC model using the following rule, such that each thread can be executed concurrently without bank conflict for array A.

LP_k executes:

for $j = k + 1$ to M by P

... $A[a^*j]$ ...

where $0 \leq k \leq P - 1$, and LP_k is one of the P logical processors in PMBC.

Proof: (1) Assume that a is even. If $a' > 0$, $2 \leq a' \leq B/2$, since $a' = \min(e, B - e)$ and $(a \bmod B) = e$. Moreover, $B \bmod (a'^*P) = 0$ implies that there exists an integer q such that $a'^*q^*P = B$. According to lemma 2, (q^*P) bank modules will be accessed. In addition, based on the result of lemma 1, we can partition this loop into P threads such that each thread references exactly q bank modules.

(2) According to lemma 3, we know that B bank modules will be accessed. Moreover, in terms of the result of lemma 1, we can partition this loop into P threads such that each thread references exactly B/P bank modules. <Q.E.D.>

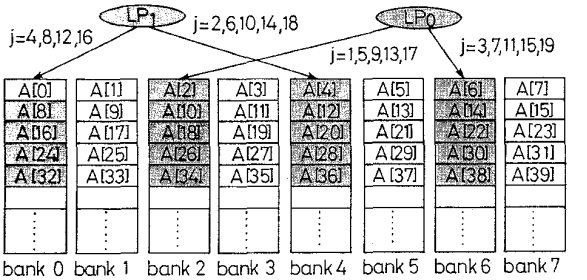

Fig.2 *An example for lemmas and theorem: access conditions*

To explain the above lemmas and theorem, an example is illustrated in Table 1 and Fig. 2 for the case of $a = 2$, $P = 2$, and $B = 8$. In Table 1, we find that the remainders can be divided into two partitions: R_0 and R_1. We also find that the partitions of the remainders for a^*j and $(B - a)^*j$ are the same; that is, the number of banks that will be accessed can be calculated by

either a or $(B - a)$. Fig. 2 shows that LP_0 always accesses banks 2 and 6, which combine to equal set R_0, and that the LP_1 always accesses banks 0 and 4.

Next, the partition rules developed in this section are applied to more complicated structures. It should be noted, however, that different preprocessing procedures must be performed for some structures. These procedures will be discussed in the following two Sections.

Table 1. Example for lemmas and theorem: remainders and partitions

$k = 0$					$k = 1$				
j	$a*j$	r	$(B-a)*j$	r	j	$a*j$	r	$(B-a)*j$	r
1	2	2	6	6	2	4	4	12	4
3	6	6	18	2	4	8	0	24	0
5	10	2	30	6	6	12	4	36	4
7	14	6	42	2	8	16	0	48	0
9	18	2	54	6	10	20	4	60	4
11	22	6	66	2	12	24	0	72	0
13	26	2	78	6	14	28	4	84	4
15	30	6	90	2	16	32	0	96	0
17	34	2	102	6	18	36	4	108	4

$a = 2$, $P = 2$, $B = 8$, $R = \{0, 2, 4, 6\}$, $R_0 = \{2, 6\}$, $R_1 = \{0, 4\}$

4 Multi-dimension array, single loop structure

Consider the example shown in Fig. 3. Because both $A[0][0]$ and $A[1][0]$ are allocated to the same bank (based on the allocation rule in our PMBC model), we cannot make the best use of the high bandwidth cache by directly applying theorem 1 to partition the loop. One solution to this problem is to restructure the array such that the loop index j is a variable in the subscript expression of the last array dimension.

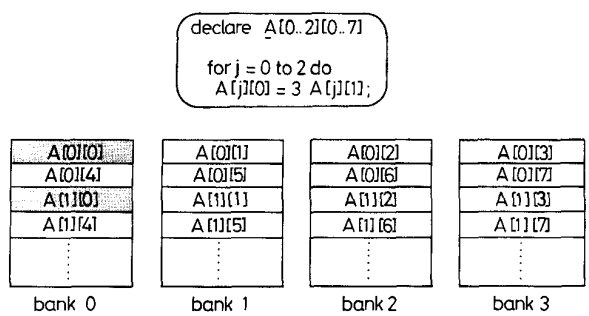

Fig. 3 *The original code segment and data allocation*

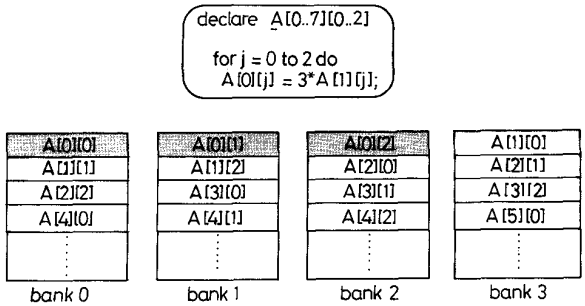

Fig. 4 *The amended code segment and data allocation after dimension interchange*

Given an array $A[i_1][i_2]...[i_{e-1}][i_e]$, if the loop index influences the subscript value of the fth dimension, and f does not equal e, we can redeclare the array by interchanging the dimensions f and e. We see that $A[0][0]$ and $A[1][0]$ can be allocated to different bank modules after the interchange as shown in Fig. 4.

Let us consider another problem. As shown in Fig. 4, since $A[0][0]$ and $A[1][0]$ are not mapped to the same bank module, the partition rule for single dimension array, single loop structures mentioned in Section 3 cannot be directly applied to this example. In order to use the results from the preceding Section, we must reduce the partition problem by mapping and restructuring the multidimension array, single loop structure into an appropriate single dimension array, single loop structure. The principle behind this reduction is that only the subscript of the last array dimension will be used to determine which bank module will be allocated. To meet this requirement, the number of array elements in the last dimension must be expanded to a multiple of the number of the bank modules. Suppose, for instance, that there is an e-dimension array $A[i_1][i_2]...[i_{e-1}][i_e]$. Assume that the range of the last dimension of array A is $0 \leq i_e < N$. Let $r_e = N \bmod B$, where B is the number of bank modules. If $r_e \neq 0$, let $N_1 = N + (B - r_e)$; otherwise, $N_1 = N$. We can redeclare the range of the last dimension of array A to be $0 < i_e < N_1$.

After the alignment, there may be some dummy elements at the end of the last dimension, but all the first elements of every dimension are mapped to bank 0. Thus, the bank module that a certain element belongs to depends only on the subscript of the last dimension. Fig. 5 illustrates the result of the array alignment of the code segment in Fig. 4.

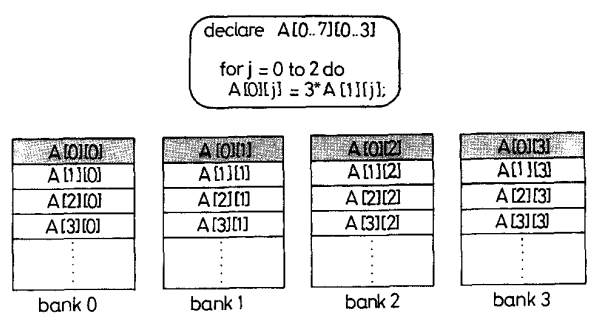

Fig. 5 *The amended code segment and data allocation after array alignment*

Trivially, the alignment procedure cannot be performed before the interchange procedure because the latter will destroy the result of the former. After the alignment and the interchange procedures are performed, loop partition of the multidimension array, single loop structure can be performed in terms of theorem 1.

5 Linear subscript expression, nested loop structure

According to the statistics reported in previous, related articles, a large percentage of expressions are linear. Therefore, it is reasonable to assume that the array subscript expressions are linear [8]. It also makes sense to assume that the array is single-dimensional because multidimension arrays can be handled just like single dimension arrays in terms of the approaches mentioned in the above section. Consider the following code segment

for $i = 1$ to N do

for $j = 1$ to M do

... $A[a_2{}^*i + a_1{}^*j + a_0]$ ...

In the following paragraphs, we will discuss how to partition the loop in terms of the values of a_0, a_1, and a_2.

Case 1: The constant a_0 contributes a constant offset for every array access that does not affect the distance between two adjacent accesses. Therefore, the constant a_0 can be ignored when the loop is partitioned.

Case 2: If $((a_1 \bmod B) \bmod P) = 0$ and $((a_2 \bmod B) \bmod P) \neq 0$, then the value of the loop index j decides which elements will be accessed. Hence, we can partition the outer loop according to the value of a_2. This partition method is the same as theorem 1 but replaces a with a_2. Alternatively, if the loop is interchangeable, we can partition the innermost loop after the interchange.

Case 3: If $((a_1 \bmod B) \bmod P) \neq 0$, then the value of loop index j will determine which elements will be accessed, and, therefore, we can partition the innermost loop according to theorem 1. However, in order to allow each logical processor to access specific bank modules, we must recalculate the initial value of index j whenever index i is increased by one. Let j_0 be the initial value of index j when the value of index i is i', and $((a_2 \bmod B) \bmod P) = a_2'$. When the value of i becomes $(i' + 1)$, the initial value of j will be:

$$(j_0 - a_2') \bmod P$$

If there are multiple arrays in the loop body, we can choose a major array that is accessed most frequently and then divide the loop according to the major array subscript expression.

6 Extension to MP systems

Although the partition method described was developed for the PMBC uniprocessor model, we can extend this method easily to PMBC-based MP systems. Each processing element in this kind of MP system has its own local cache, each of which possesses multiple bank modules; several simultaneously executing threads share this cache. We need to partition loops twice, such that each partition has two levels, as shown in Fig. 6. The higher level is the basic unit of process scheduling and will be allocated to a certain processing element which we call a 'cluster'. Therefore, when partitioning loops into clusters, one should be aware of cache locality. Next, when each cluster is divided into several threads, we let each thread access specific bank modules such that bank conflicts will be reduced.

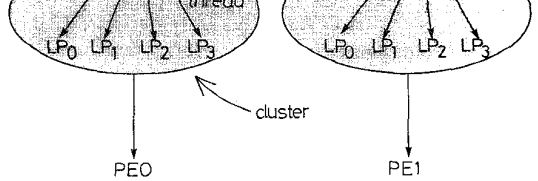

Fig. 6 *Two-level loop partition for PMBC-based MP systems*

Many studies have been aimed at partitioning loops so as to exploit the good cache locality in MP systems, but all such research has failed to treat the critical matter of whether or not the local cache is multibank. However, for the PMBC uniprocessor, the proposed loop partition method takes into account multibank caches. Hence, we can combine this present approach with one result from previous work to take advantage of both. First, we apply the current method to partition the innermost loop (a loop interchange may be needed to utilise the cache features of the PMBC architecture). Next, we partition the outer loop into clusters, according to one of the approaches developed in previous studies, e.g. loop tiling [16].

7 Simulation results

In this Section, some preliminary performance gains resulting from use of the proposed method are evaluated. It is assumed that applying this method can give a bank-conflict-free partition. In addition, an analysis of the impacts of the cache parameters is given, including the number of bank modules, the number of logical processors, and the hit ratio. A simulator has been constructed in the IBM RS/6000 workstation to simulate the effect of loop partition for multibank caches that uses a random procedure to generate test data instead of using real programs. In this way, performance evaluations can be done by controlling the following important parameters involved in the random procedure:

- $\#p$: the number of logical processors
- $\#b$: the number of bank modules
- $\%h$: the hit ratio

First, we give some reasonable assumptions: the percentage of memory accesses ($\%m$) equals 30%, the hit time (T_h) equals 1 cycle, the miss time (T_m) equals 16 cycles, and the hit ratio ($\%h$) equals 95% [1]. Next, we define the following two evaluation criteria:

(i)

$$\text{bank conflict ratio} = \frac{\begin{array}{l}\text{the number of memory accesses}\\ \text{that cannot be granted}\\ \text{access to the corresponding}\\ \text{bank module when issued}\end{array}}{\text{total number of memory accesses}} \quad (1)$$

(ii)

$$\text{average access time : } AAT = \frac{\sum_1^t W_i + \sum_1^h T_h + \sum_1^z T_m}{\text{total number of memory accesses}}$$

where W_i represents the waiting time of the ith memory access because of a bank conflict, h and z represent the number of memory access requests that hit and miss, respectively. Additionally, the total number of the memory accesses, t, equals $h + z$.

$$\text{average access-time speedup} = \frac{AAT_{\text{multibank cache}}}{AAT_{\text{bank-conflict-free cache}}} \quad (2)$$

where $AAT_{\text{multibank cache}}$ represents the average access time of the multibank cache without bank-conflict-free loop partition, and $AAT_{\text{bank-conflict-free cache}}$ represents the average access time of the multibank cache with loop partition when there is no bank conflict. In the multibank cache configuration, it is assumed that the probability of a certain memory request accessing any one of the bank modules is equal. However, in the bank-con-

flict-free cache configuration, the memory request from a thread accesses specific bank modules, and no two threads access the same bank module.

In the following subsection, these two criteria are used as a basis to examine the impact of each parameter. There is one notation that we have to explain first:

PMBC(P, B) represents a PMBC processor with $\#p = P$ and $\#b = B$

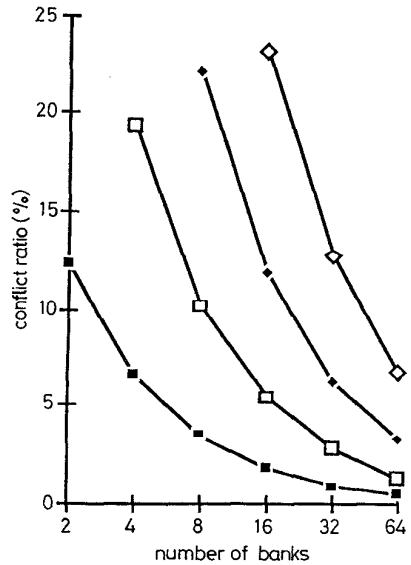

Fig. 7 *The bank conflict ratio for various configurations*
—■— 2-LP
—□— 4-LP
—◆— 8-LP
—◇— 16-LP

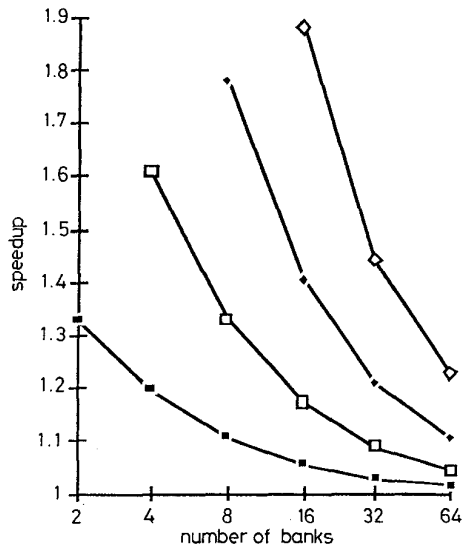

Fig. 8 *Average access time speedup for various configurations*
—■— 2-LP
—□— 4-LP
—◆— 8-LP
—◇— 16-LP

7.1 The number of logical processors versus the number of bank modules

The larger the $\#p$, the greater the likelihood that the bank conflict will occur. This is because more threads will issue memory requests to a specific bank module. On the other hand, when the $\#b$ is enlarged, the possibility that a certain memory request will access a specific bank module is $1/\#b$ according to our assumption. The bank conflict ratio decreases more and more slowly when $\#b$ or $\#p$ increases gradually, as shown in Fig. 7. Fig. 8 shows the speedup of the AAT for various configurations. Because the bank conflict ratio decreases similarly, the speedup is decreased when the $\#b$ increases. The decrease gradient is steeper for configurations with more logical processors.

7.2 The consequences of different hit ratios

The bank conflict ratio is proportional to the hit ratio, as shown in Fig. 9. On average, the hit ratio is greater than 90%, otherwise, the point of using caches is lost. The speedup increases when the hit ratio decreases from 100% to 90%, but after the hit ratio drops below 90%, the speedup decreases a little, as shown in Fig. 10. This is because the performance gain is obtained by removing the waiting time for accessing bank modules. We express it in the following formula:

$$\text{performance gain} = \frac{\sum_{1}^{t} W_i}{\%h * T_h + (100\% - \%h) * T_m} \quad (3)$$

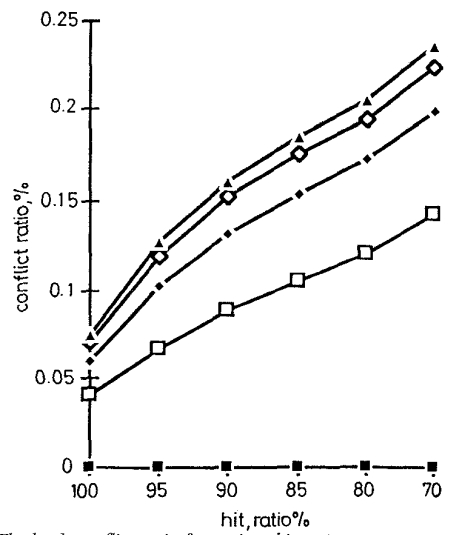

Fig. 9 *The bank conflict ratio for various hit ratios*
—■— PMBC(1,2)
—□— PMBC(2,4)
—◆— PMBC(4,8)
—◇— PMBC(8,16)
—▲— PMBC(16,32)

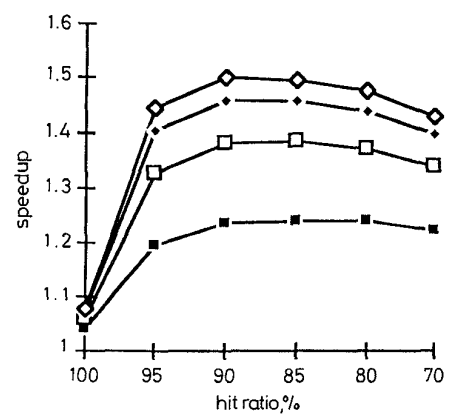

Fig. 10 *Average access time speedup for various hit ratios*
—■— PMBC(2,4)
—□— PMBC(4,8)
—◆— PMBC(8,16)
—◇— PMBC(16,32)

The sum of $\Sigma_1^t W_i$ is related to the bank conflict ratio and the miss time, T_m, but T_m is constant in this expression. Both T_h and T_m are constant in the expression of the denominator, so the value of the denominator depends only on $\%h$. When the $\%h$ is large enough,

the sum of $\Sigma_1^t W_i$ dominates the result of the expression. Otherwise $(100\% - \%h)^* T_m$ dominates the result of the expression, because T_m is much larger than T_h.

8 Concluding remarks

In this paper, a loop partition method has been proposed that can reduce or even eliminate bank conflict for parallel multithreaded processor with multibank cache configurations. The proposed approach can be applied to many numeric calculations, including, inner product, matrix multiplication, and Gauss–Jordan elimination. For instance, the transformation of matrix multiplication is as follows:

```
for m = 1 to 25 do
  for i = 1 to 25 do
    temp = Y(i,m);
    for j = 1 to n do
      X(i,j) = X(i,j) + temp*X(m,j);
⇒
for m = 1 to 25 do
  for i = 1 to 25 do
    temp = Y(i,m);
(LP_k) for j = k + 1 to n by P
    X(i,j) = X(i,j) + temp* X(m,j);
```

So far, we have obtained some important results using simulation results: (1) the performance can be enhanced, especially when the difference between the number of bank modules and the number of logical processors is increased; (2) if the number of logical processors, $\#p$, is larger than or equal to the number of bank modules, $\#b$, speedup will increase when $\#p$ and $\#b$ increase. The best speedup was approximately 1.9 in the simulation results; (3) when the hit ratio decreased, speedup first increased, then, once above 90%, decreased slowly.

In the future, more general cases will be further studied and analysed. Two important problems remain to be solved: (i) finding the heuristic function that selects the major array, or the major subscript expression; and (ii) partitioning loops when the number of logical processors is not a power of 2.

9 Acknowledgments

The authors would like to thank the reviewers for their helpful comments. This research was supported by National Science Council under the contract number: NSC84-2221-E009-001.

10 References

1 KECKLER, S.W., and DALLY, W.J.: 'Processor coupling: integrating compiler time and runtime scheduling for parallelism', Proceedings of the 19th Annual International Symposium on *Computer Architecture*, May 1992, pp. 202–213
2 IANNUCCI, R.A., GAO, G.R., HALSTEAD, R.H. Jr, and SMITH, B.: 'Multithreading: a summary of the state of the art' (Kluwer Academic, 1993)
3 HIRATA, H., KIMURA, K., NAGAMINE, S., and MOCHIZUKI, Y.: 'An elementary processor architecture with simultaneous instruction issuing from multiple threads', Proceedings of the 19th International Symposium on *Computer Architecture*, May 1992, pp. 136–145
4 AGARWAL, A., KUBIATOWICZ, J., KRANZ, D., LIM, B.-H., YEUNG, D., D'SOUZA, G., and PARKIN, M.: 'Sparcle: an evolutionary processor design for large-scale multiprocessors', *IEEE Micro*, 1993, **13**, (3), pp. 48–61
5 ALVERSON, G., ALVERSON, R., CALLAHAN, D., KOBLENZ, B., PORTERFIELD, A., and SMITH, B.: 'Exploiting heterogeneous parallelism on a multithreaded multiprocessor', Workshop on *Multithreaded Computers*, Proceedings of *Supercomputing'91*, Albuquerque, New Mexico, November 1991
6 NIKHIL, R.S., PAPADOPOULOS, G.M., and ARVIND : '*T: A multithreaded massively parallel architecture', Proceedings of the 19th International Symposium on *Computer Architecture*, 1992, pp. 156–167
7 HUANG, C.-H., and SADAYAPPAN, P.: 'Communications-free hyperplane partitioning of nested loops', *J. Parallel and Distributed Computing*, 1993, **19**, pp. 90–102
8 FANG, J.Z. and LU, M.: 'An iteration partition approach for cache or local memory thrashing on parallel processing', *IEEE Trans. Computers*, 1993, **42**, (5), pp. 529–546
9 BANERJEE, U., EIGENMANN, R., NICOLAU, A., and PADUA, D.A.: 'Automatic program parallelization', *Proc. IEEE*, 1993, **81**, (2), pp. 211–243
10 WOLF, M.E., and LAM, M.S.: 'A data locality optimizing algorithm', Proceedings of ACM SIGPLAN'91 Conference on *Programming Language Design and Implementation*, 1991, pp. 30–44
11 RAMANUJAM, J., and SADAYAPPAN, P.: 'Compile-time techniques for data distribution in distributed memory machines', *IEEE Trans. Parallel Distributed Systems*, 1991, **2**, (4), pp. 472–482
12 BAILEY, D.H.: 'Vector computer memory bank contention', *IEEE Trans.*, 1987, **C-36**, pp. 293–298
13 RAGHAVAN, R., and HAYES, J.P.: 'On randomly interleaved memories', Proceedings of Supercomputing'90, November 1990, pp. 49–58
14 HARPER, D.T.: 'Increased memory performance during vector access through the use of linear address transformations', *IEEE Trans. Computers*, 1992, **41**, pp. 227–230
15 SOHI, G.S.: 'High-bandwidth interleaved memories for vector processors-A simulation study', *IEEE Trans. Computers*, 1993, **42**, (1), pp. 34–44
16 HWANG, K.: 'Advance computer architecture: parallelism, scalability, programmability' (McGraw-Hill, 1993)