Reducing Cache Conflicts by Multi-Level Cache Partitioning and Array Elements Mapping

Chih-Yung Chang
Department of Computer and Information Science
Aletheia University
32 Chen-Li St., Tamsui, Tapiei, Taiwan
changcy@email.au.edu.tw

Jang-Ping Sheu and Hsi-Chiuen Chen
Department of Computer Science and Information Engineering
National Central University
Chung-Li 32054, Taiwan
{sheu,camus} @axp1.csie.ncu.edu.tw

Abstract

This paper presents an algorithm to reduce cache conflicts and improve cache localities. The proposed algorithm analyzes locality reference space for each reference pattern, partitions the multi-level cache into several parts with different size, and then maps array data onto the scheduled cache positions such that cache conflicts can be eliminated. To reduce the memory overhead for mapping arrays onto partitioned cache, a greedy method for rearranging array variables in declared statement is also developed. Besides, we combine the loop tiling and the proposed schemes for exploiting both temporal and spatial reuse opportunities. To demonstrate that our approach is effective at reducing number of cache conflicts and exploiting cache localities, we use Atom as a tool to develop a simulator for simulation of the behavior of direct-mapping cache. Experimental results show that applying our cache partitioning scheme can largely reduce the cache conflicts and thus save program execution time in both one-level cache and multi-level cache hierarchies.

1 Introduction

The growing speed gap between memory and processor makes the memory system design become a sophisticated memory hierarchy. For uniformity of reference, both cache and main memory are divided into equal-sized units, called block in main memory and cache line in the cache [1]. The placement policy determines the mapping function from the main memory address to the cache location. There are basically three placement policies: direct, fully associative, and set associative. The direct mapping scheme maps blocks of main memory to cache in a round robin manner. The direct mapping scheme has the advantage that no associative comparison is needed and, hence, the cost is reduced[1]. However, a disadvantage of direct-mapping cache is that the cache hit ratio drops sharply if two or more blocks, used alternately, happen to map onto the same cache line in the cache.

Cache conflicts during the execution of loop nests in scientific program cause data to be swapped out from the cache, leading to cache misses and degrading performance. A lot of efforts focus their attention on the insertion of some padding arrays [4][5]. However, for the following two cases of loops programs, adding a small amount, say, a cache line size, of padding arrays can not avoid the occurrence of cache conflicts. Firstly, when the coefficients of index variables of reference pattern are not equal, accessing elements of two different arrays may cause cache conflict, even though a cache line size of padding array has been inserted between these two arrays. Secondly, for those loops programs that there exist localities in a long iteration distance, padding arrays scheme can prevent conflicts but can not exploit the reuse opportunities even though loop tiling technique is applied.

Manjikian et al.[4] proposed a cache partitioning algorithm to partition cache into equal-sized regions and to allocate each region for each array variable. During loop execution, the accesses to elements with spatial locality or temporal locality will have a good cache performance since el-

0-7695-0568-6/00 $10.00 © 2000 IEEE

ements of the same array have their own region and will not be replaced by another array. However, if we treat cache as a system resource, it is reasonable that cache size should be scheduled according to the working set of arrays accessed during loop execution. Another critical problem of the equal-sized cache partitioning technique is that there is a large amount of memory fragmentation. To implement the cache partitioning scheme under direct mapping cache policy, a large amount of padding arrays should be introduced to achieve the goal of cache partitioning.

In this paper, we propose a cache partitioning technique to reduce the number of cache conflict. The proposed partitioning scheme partitions cache into several regions, possibly with different sizes, and maps the array elements of main memory onto the allocated region without any hardware support. For a multi-level cache memory system, we propose an algorithm to evaluate the size of padding array for each user-declared array such that all the user-declared arrays can be scheduled on the partitioned cache regions in each cache level. To minimize the memory overhead caused by applying padding array scheme, we develop a greedy method to redeclare the arrays in a different order such that the size of padding array can be reduced. For the case that the size of arrays in real applications is much larger than partitioned cache region and data will be reused after a long iteration distance, we employ a method for combining the well-known loop tiling technique [5][7] and our cache partitioning and mapping scheme.

2 Basic Concepts

In most modern computer systems, both cache and main memory are divided into equal-sized units, called block in main memory and cache line in the cache. During program execution, making access to an element will imply that a section of data is transferred from main memory to cache. Data brought into cache should be reused as much as possible before they are replaced. In most loop applications, exploiting locality of reference is the key to achieving high levels of performance.

Conflict misses may occur when too many data items map to the same set of cache locations, causing cache lines to be flushed from cache before they are reused. For example, consider the following loop program.

Example 1:

float $A[128][128], B[128][128], C[128][128]$;
for (I=0; I <128; I++)
  for (J=0; J <128; J++)
    S_1:$C[I][J] = A[I][J] + B[I][J]$; $(L1)$

Assume that the row-major main memory uses 8 bytes to store a floating point number and each cache line has capacity of 4 floating point elements (32 bytes). Thus, the memory space for storing array A is 128*128*8=128 KB. At the cases that the size of a direct-mapping cache is 16K, 32K, 64K, or 128K bytes, cache conflicts will occur at each reference during execution of loop $L1$. This is because that the cache size is a factor of the size of arrays A, B, and C. The direct mapped cache will map array elements $A[I][J]$, $B[I][J]$, and $C[I][J]$ onto the same cache line, for specific values of I and J.

Manjikian et al.[4] proposed a partitioning algorithm to divide cache into several equal-sized regions and apply *padding technique* [4][5] to map array data onto the partitioned regions. Applied to loop $L1$, their cache partitioning technique will partition cache into three equal-sized regions for arrays A, B, and C. Partitioning cache into several regions and allocating one region to each array can avoid cache conflicts in execution of Example 1. However, treating cache as a resource of system, the compiler should partition cache into regions according to the working set of each array. Since the working set of reference patterns are possibly not equal, partitioning cache into several regions with different size is necessary. The following example illustrates this situation.

Example 2:

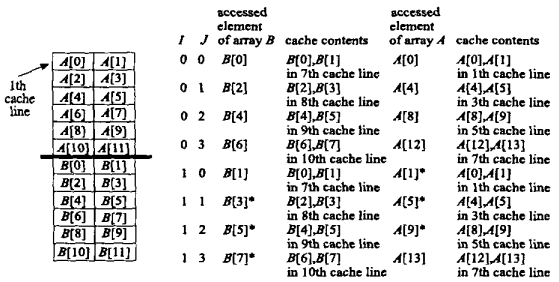
S_1:float $A[504], B[504]$;
for (I=0; I <100; I++)
  for (J=0; J <100; J++)
    S_2:$A[I + 4 * J] = B[I + 2 * J]$; $(L2)$

For simplicity, we assume that there is only one level cache with size of 12 cache lines; each line has capacity of 2 array elements. Since the cache size is a factor of size of array A, the corresponding elements of arrays A and B are mapped to the same cache line. Iterations (I, J)=(0, 0) and (I, J)=(1, 0) will respectively access elements $B[0]$ and $B[1]$ which are located in the same cache line. The reference of element $B[0]$ in iteration (I, J)=(0, 0) will cause the movement of elements $B[0]$ and $B[1]$ from memory to cache. To exploit the spatial locality, we apply the loop tiling technique to loop $L2$. If we tile the innermost loop J with a block size of 4 iterations, loop $L2$ is then modified as the following loop $L2'$.

S_1: float $A[504], B[504]$;
for (J'=0; J' <100; $J' = J' + 4$)
  for (I=0; I <100; I + +)
    for (J=J'; J < $J' + 4$; J++)
      S_2:$A[I + 4 * J] = B[I + 2 * J]$; $(L2')$

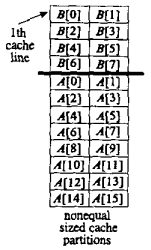
Applied the equal-sized cache partitioning technique proposed in [4], compiler will insert a padding array $P[12]$ between arrays A and B and redeclare S_1 as

float $A[500], P[12], B[500]$



I	J	accessed element of array B	cache contents	accessed element of array A	cache contents
0	0	B[0]	B[0],B[1] in 7th cache line	A[0]	A[0],A[1] in 1th cache line
0	1	B[2]	B[2],B[3] in 8th cache line	A[4]	A[4],A[5] in 3th cache line
0	2	B[4]	B[4],B[5] in 9th cache line	A[8]	A[8],A[9] in 5th cache line
0	3	B[6]	B[6],B[7] in 10th cache line	A[12]	A[12],A[13] in 7th cache line
1	0	B[1]	B[0],B[1] in 7th cache line	A[1]*	A[0],A[1] in 1th cache line
1	1	B[3]*	B[2],B[3] in 8th cache line	A[5]*	A[4],A[5] in 3th cache line
1	2	B[5]*	B[4],B[5] in 9th cache line	A[9]*	A[8],A[9] in 5th cache line
1	3	B[7]*	B[6],B[7] in 10th cache line	A[13]	A[12],A[13] in 7th cache line

Figure 1. The equal-sized cache partitioning and mapping of arrays A and B for Loop $L2'$.



I	J	accessed element of array B	cache contents	accessed element of array A	cache contents
0	0	B[0]	B[0],B[1] in 1th cache line	A[0]	A[0],A[1] in 5th cache line
0	1	B[2]	B[2],B[3] in 2th cache line	A[4]	A[4],A[5] in 7th cache line
0	2	B[4]	B[4],B[5] in 3th cache line	A[8]	A[8],A[9] in 9th cache line
0	3	B[6]	B[6],B[7] in 4th cache line	A[12]	A[12],A[13] in 11th cache line
1	0	B[1]*	B[0],B[1] in 7th cache line	A[1]*	A[0],A[1] in 5th cache line
1	1	B[3]*	B[2],B[3] in 2th cache line	A[5]*	A[4],A[5] in 7th cache line
1	2	B[5]*	B[4],B[5] in 3th cache line	A[9]*	A[8],A[9] in 9th cache line
1	3	B[7]*	B[6],B[7] in 4th cache line	A[13]*	A[12],A[13] in 11th cache line

Figure 2. The nonequal-sized cache partitioning and mapping of arrays A and B for loop $L2$.

so that arrays A and B can be mapped onto the equal-sized cache regions. Fig. 1 shows the equal-sized cache partition. The cache is partitioned into two parts and allocate arrays A and B to each part. Each partitioned cache consists of 6 cache lines and has capacity of storing 12 array elements. In Fig. 1, the first and the second columns display the running iteration. The third and the fourth columns of Fig. 1 respectively show the accessed element and the cache contents for a specific iteration.

For example, at the execution of (I, J)=(0, 0), statement S_2 will access array element $B[0]$. The cache memory management system will transfer elements $B[0]$ and $B[1]$ from main memory to the 7th cache line of cache. However, at the execution of (I, J)=(0, 3), the reference of array element $A[12]$ will cause elements $A[12]$ and $A[13]$ are moved from main memory to the 7th cache line. The array element $B[1]$, which is expected to be reused in iteration (I, J) = (1, 0), will be replaced by $A[13]$. The main reason is that the space of accessed elements of array A is larger than one of array B during execution of the innermost loop. As we can see, the space of accessed elements of array A is $A[0 : 12]$ ($A[0 : 12]$ is accessed in execution of $I = 0$, $0 \leq J \leq 3$) whereas the space of accessed elements of array B is $B[0 : 6]$. Note that in the third and the fifth columns of Fig. 1, elements with a '*' symbol indicate that the reference of this element can be obtained in cache before they are replaced. That is, the reuse opportunity of spatial locality is exploited. During the execution of innermost loop, only 6 reuse opportunities are exploited by applying equal-sized cache partitioning technique.

In our approach, we firstly identify the array variables that are mapped to the same cache line. According to the size of space accessed by these array variables, we partition cache into several regions with different size. Each partitioned region is assigned to an array variable to avoid the situation that the preloading elements are replaced before they are accessed. According to the partitioned cache space for each array variable, we will determine the tiling size and apply the loop tiling technique to exploit the reuse opportunities. For example, consider the loop $L2$ of Example 2. Since the cache size is a factor of size of arrays A and B, the corresponding elements of arrays A and B will be mapped to the same cache line. We partition the cache into two regions for arrays A and B. The space accessed by array A for the innermost loop J is $A[4 * J], 0 \leq J \leq 99$, which is equal to the working set $A[0 : 396]$. By the similar computation, the space accessed by array B for the innermost loop J is $B[0 : 198]$. The ratio of the reference space of arrays A and B is 2:1. Thus, we partition the cache into two regions according to the ratio of reference space of arrays A and B. That is, allocate (2/3 * 12)=8 cache lines to array A and allocate (1/3 * 12)=4 cache lines to array B. To map arrays A and B onto allocated cache regions under direct mapping policy, insertion of padding array and redeclaration of statement S_1 are needed. Two possible declarations are considered as follows:

$$S_1 : \text{float } A[504], P[16], B[504] \text{ or}$$

$$S_1 : \text{float } B[504], P[8], A[504].$$

To reduce the size of memory fragmentation caused by applying padding array technique, the second declaration is selected. The partitioned cache and the mapping of arrays A and B are shown in Fig. 2.

To exploit the reuse opportunities of execution of outer loop, we further apply the well known tiling technique to cooperate with our cache partitioning technique to exploit the reuse opportunities of cache data in a partitioned region. Since the size of region allocated to array B is four cache lines which contain eight floating elements, we set the tiling size to 4 to guarantee that all the cache data will not be replaced by elements of array A during execution of the innermost loop. This makes it possible that the reuse opportunities existed in the execution of successive outer loop iterations are exploited. By applying our cache partitioning technique to Example 2, we have the following program.

S_1: float $B[504]$, $P[8]$, $A[504]$;
for (J'=0; J' <100; $J' = J' + 4$)

for (I=0; I <100; I + +)
for (J=J'; J < J' + 4; J++)
S_2:$A[I + 4 * J] = B[I + 2 * J]$; ($L2''$)

Compared to the equal-sized cache partition, two advantages can be found in the nonequal-sized partitioning scheme. First, we save two cache lines of memory fragmentation. That is, the size of padding arrays has been reduced from $P[12]$ to $P[8]$. Second, all the preloading elements can be accessed before they are replaced. This situation can be found in Fig. 2. For example, in the execution of (I, J)=(0, 0), the reference of $B[0]$ will cause elements $B[0]$ and $B[1]$ moves from main memory to the first cache line. The preloading element $B[1]$ will be accessed in iteration (I, J)=(1, 0). This is because that we allocate a larger cache region to array A such that the accessed elements of array A will not replace the preloading element of array B during execution of iterations of innermost loop. As shown in Fig. 2, there are 8 preloading elements (with a '*' symbol) accessed before they are replaced. Compared to the equal-sized partitioning technique, two more elements are exploited their spatial locality.

3 Preliminaries and The Algorithms

In this section, cache partitioning and array mapping algorithm for one level and multi-level cache hierarchies is proposed.

Array variables are said in a *dependent set* if their first elements are mapped to the same cache line and their reference patterns access to the same cache line during the execution of the innermost loop or tiling loop. Variables not belonging to any dependent set are said in an *independent set*. Without loss of generality and for simplicity, we assume all arrays A_i are one dimensional. For those real applications that multi-dimensional arrays are accessed, our algorithm only considers to exploit localities of array references in the rightmost dimension. Thus, the reference pattern $A_i[f]$ in loops program can be expressed by

$$A_i[f] = A[a_{i1}I_1 + a_{i2}I_2 + \ldots + a_{in}I_n],$$

where a_{ij} is the coefficient of loop index variable I_j of array A_i.

To illustrate our cache partitioning algorithm in detail, consider the following loops program.

Example 3:
float A[128000], B[128000];
float C[140800], D[128000], E[128000];
for (I=0; I < 25600; I++)
for (J=0; J < 25600; J++)
S_1: $A[I + J] = B[J]$; ($L3$)
...
for (I=0; I < 128000; I++) {

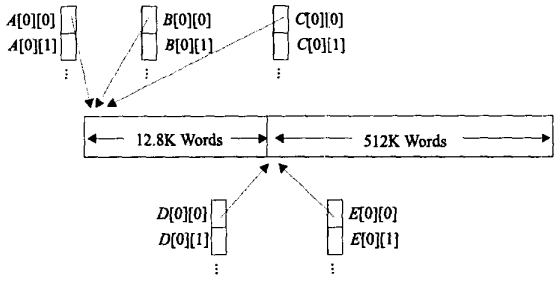


Figure 3. Cache mapping of arrays A, B, C, D, and E of loop $L3$.

S_2: $D[I] = C[I] + D[I]$; ($L4$)
S_3: $E[I] = C[I] + E[I]$; }

Assume that the cache size is 64K words. There are two reference patterns $A[I + J]$ and $B[J]$ in loop $L3$. Since the size of array variables A, B, D, and E are the multiples of cache size, as shown in Fig. 3, array elements $A[0]$, $B[0]$, and $C[0]$ are mapped to the same cache line and array elements $D[0]$ and $E[0]$ are mapped to the same cache line. In Example 3, we have two dependent sets $\{A, B\}$ and $\{D, E\}$ respectively belonging to loops $L3$ and $L4$. Similarly, the independent set of Example 3 is $\{C\}$.

For the sake of clarity, we further define a few terms which are used throughout the study.

- C^i : the ith level cache.
- C_s^i : the size of the ith level cache.
- A_i : the array variables in the dependent set.
- B_j : the array variables in the independent set.
- S_{A_i} : the size of array A_i measured by number of words.
- $C_{A_i}^j$: the size of region allocated to array A_i in the jth level cache.
- P_{A_i} : the needed size of padding array to allocate a region for array A_i.
- R_{A_i} : the reference space of array A_i.
- $|R_{A_i}|$: the size of reference space R_{A_i}.
- N : the number of cache levels in a cache system.
- m : the number of arrays in the independent set.
- n : the depth of loop nests.
- k : the number of arrays in a dependent set.
- I_i : the index variable of the ith depth loop counting from outermost loop, $1 \leq i \leq n$.
- a_{ij} : coefficient of index I_j in reference pattern A_i.
- r : the number of inner loops that temporal and spatial reuses should be exploited during the execution of index variables $I_{n-r+1}, I_{n-r+2}, \ldots, I_n$.
- d : the number of dependent sets.
- G_i : the ith dependent set.

P_{G_i} : the size of padding arrays needed for aligning arrays in set G_i.

We will use these terms to illustrate our methods for one level cache and multi-level caches in latter subsections.

3.1 One Level Cache Hierarchy

In this subsection, we will describe how we partition one level cache into k regions and map the k arrays onto these regions, where k is the number of dependent arrays for a specific loop body.

Assume that r is the number of inner loops that are determined to exploit the spatial and temporal localities during the execution. We define the *locality reference space* of array A_i by the set of elements that are accessed during execution of r inner loops. The locality reference space is determined by the loop index variables $I_{n-r+1}, \ldots, I_{n-1}, I_n$. The locality reference space R_{A_i} of array A_i can be derived by

$$R_{A_i} = \{A_i(x) | x = a_{i(n-r+1)} I_{n-r+1} + \ldots + a_{in} I_n,$$

$$l_j \leq I_j \leq u_j,\ n-r+1 \leq j \leq n\}.$$

We denote the size of locality reference space of array A_i by $|R_{A_i}|$. The partitioned region for array A_i is

$$|R_{A_i}| / \sum_{x=1}^{k} |R_{A_x}|,$$

where k denotes the number of arrays in the dependent set. In the case that all a_{ij} are positive integers for $n-r+1 \leq j \leq n$, the block sizes $u_p - l_p + 1$ of index variables I_p are equal for all $n-r+1 \leq p \leq n$. Cache partition region for array A_i can be simplified by

$$\sum_{x=n-r+1}^{n} a_{ix} / \sum_{i=1}^{k} \sum_{x=n-r+1}^{n} a_{ix}.$$

Consider loop $L3$. The locality reference space of array A is $A[0:330]$ which can be derived by

$$A[I+J], 0 \leq I \leq 165, 0 \leq J \leq 165.$$

Similarly, the locality reference space of array B is $B[0:165]$. Therefore, the ratio of locality reference space of arrays A and B is 2:1. Assume the size of one level cache is C_s^1. We will divide cache into 3 partitions and assign two partitions as a region for array A and assign one partition as another region for array B. Applied loop tiling and our cache partitioning techniques, the loop $L3$ is transformmed as shown in the following loop $L5$.

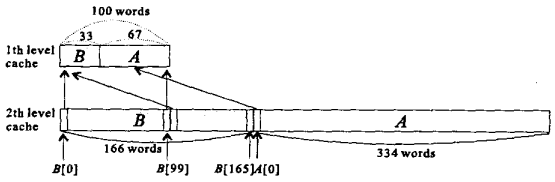


Figure 4. Partitioning of the second level cache can not map the right element onto right position of the first level region.

```
int B[128000],P_B[C_1^s/3], A[128000];
for (I' = 0; I' < 25600, I' = I' + 166)
    for (J' = 0; J' < 25600, J' = J' + 166)
        for (I = I'; I < 166 + I'; I++)
            for (J = J'; J < 166 + J', J++)
                A[I + J] = B[J];                    (L5)
```

3.2 Multi-Level Cache Hierarchy

In this subsection, memory architecture organized as multi-level caches is considered. Consider the case that the memory hierarchy consists of two-level caches. Partitioning of the second level cache into several regions and allocating them to array variables will not guarantee that data mapped from region of the second level cache to the first level cache will be placed in the region that we scheduled for the first level cache. To illustrate, consider loop $L5$ again. Let the cache memory hierarchy consist of two level cache, as the cache hierarchy design in most modern computer. Assume that the first level cache is with size C_s^1=100 words and the second level cache is with size C_s^2=500 words. The ratio of locality reference space of arrays A and B is 2:1. Our cache partitioning method will partition the first level cache into two regions. The first region is with size C_A^1 =67 words and is allocated to array A and the second region is with the size C_B^1=33 words and is allocated to array B.

If the cache system is organized as a two-level cache, we will also partition the second level cache into two regions. The first region is with size C_A^2=334 for array A and the second region is with size C_B^2=166. The partitioning of two regions of both the first and the second level caches is as shown in Fig. 4. According to the ratio of reference space of arrays A and B, we determine to partition cache into two regions, one-third of cache is allocated for array B and the remainder is allocated to array A. However, using this ratio to partition the second level cache will have a problem that the first cache line of the region allocated for array A in the second level cache is not direct mapped to the first cache line of the cache region allocated for array A in the first level cache. As shown in Fig. 4, the first element of array A is scheduled in the 166th cache word of the second level

cache. This element will be direct mapped to the 66th cache word in the first level cache. However, as our scheduling, the first element of array A in the first level cache should be mapped to the 34th cache word. To make the partition of the second level cache satisfy the scheduling we made in the first level cache, we should reduce the size of region for array B in the second level cache. Thus, cache size allocated for array B in the second level cache should be

$$\lfloor C_B^2/C_s^1 \rfloor \times C_s^1 + C_B^1 = \lfloor 166/100 \rfloor \times 100 + 33 = 133.$$

This implies that the padding array for the two level cache is $P[133]$, not $P[166]$.

Let there be k dependent array variables $A_1 \ldots A_k$ in a loop and their locality reference spaces are $R_{A_1} \ldots R_{A_k}$, respectively. We will partition the first level cache into k regions where the ith-region is with size

$$C_{A_i}^1 = \lfloor (|R_{A_i}|/\sum_{i=1}^{k} |R_{A_i}|) \times C_s^1 \rfloor,$$

and is allocated for array A_i. Similarly, for the second level cache, we will allocate cache size $C_{A_i}^2$ to array A_i where

$$C_{A_i}^2 = \lfloor (|R_{A_i}|/\sum_{i=1}^{k} |R_{A_i}|) \times C_s^2 \rfloor.$$

However, to guarantee that the first element in region of $C_{A_i}^2$ maps to the starting location of $C_{A_i}^1$, we should adjust the value of $C_{A_i}^2$ by $C_{A_i}^2 = \lfloor C_{A_i}^2/C_s^1 \rfloor \times C_s^1 + C_{A_i}^1$. The size of padding array needed to allocate $C_{A_i}^2$ to array A_i is thus $P_{A_i} = C_{A_i}^2 - (S_{A_i} \bmod C_s^1)$.

In the next subsection, we propose the multi-level cache partitioning and array elements mapping algorithm.

3.3 The algorithm

In this subsection, we combine techniques proposed in the previous subsections and propose our partitioning algorithm.

Algorithm:Multi-Level Cache Partitioning and Array Elements Mapping Algorithm.
Input:The number of cache level N, size C_s^i of the ith level cache, for $1 \le i \le N$, and a loop nests program L.
Output: A restructured program with redeclaration of array variables
Step 1:Let temporal variables $P'_{A_i} = 0$, for $1 \le i \le k$.
/* processing k arrays */
Step 2:for $(i = 1; i \le k; i++)$ {
/* processing N levels of cache */
for $(j = 1; j \le N; j++)$ {
(2.1) Compute the size of region allocated to A_i in the jth level cache by:
$C_{A_i}^j = C_s^j \times |R_{A_i}|/\sum_{i=1}^{k} |R_{A_i}|$;
/* multi-level caches consideration */
(2.2) if ($j > 1$)
$C_{A_i}^j = \lfloor C_{A_i}^j/C_s^{j-1} \rfloor \times C_s^{j-1} + C_{A_i}^{j-1}$;
} /* end of for j */
(2.3)Compute the size of padding array P_{A_i} by:
$P_{A_i} = C_{A_i}^j - (S_{A_i} \bmod C_s^j), 1 \le i \le k-1$;
if $(P_{A_i} < 0)$
$P_{A_i} = P_{A_i} + C_s^j$;
}
Step 3:/* reducing the size of padding array by moving */
/* array with maximal padding size to the last position */
Let $C_{A_{max}}^N = Max(C_{A_i}^N), 1 \le i \le k$.
Move A_{max} to the last position. That is, modify the declaration statement $A_1, \ldots, A_{max}, \ldots, A_k$ by $A_1, \ldots, A_k, A_{max}$.
Step 4: Repeatedly perform steps 2 and 3 until all dependent group G_i have been processed, $1 \le i \le d$.
Step 5: /* Reducing the size of padding array by moving */
/* the independent arrays to the most profit position. */
/* Process the d dependent groups */
Let $M = \bigcup_{j=1}^{j=m} B_j$, $P = \bigcup_{i=1}^{i=k-1} P_{G_i}$;
while $M <> \emptyset$ {
Let B_{best} and $P_{G_{best}}$ be the pair of best values that satisfy:
$P_{G_{best}}$ - ($|B_{best}|$ mod C_s^N) $\le P_{G_i}$ - ($|B_j|$ mod C_s^N),
$\forall B_j \in M, \forall P_{G_i} \in P$.
$M = M - B_{best}$, $P = P - P_{G_{best}}$;
Partition B_{best} into several subarrays and insert these subarrays into $P_{G_{best}}$ such that the subarrays can be instead of P_{A_j}, for $A_j \in P_{G_{best}}$.
Step 6: /* Cooperate with the loop tiling technique */
According to the size of region of first level cache allocated to A_1, determine the tiling size and apply loop tiling technique to L.

Step 1 of the algorithm initializes the temporal variables for calculating the size of padding array P_{A_i}. Step 2.1 computes $C_{A_i}^j$ which is the size of each region allocated to array A_i in the jth level of cache. To guarantee that the partitioning of the lower level cache can match the partitioning of the higher level cache, we need additional computation in Step 2.2. Step 2.3 computes the size P_{A_i} of padding array for preserving region for A_i and aligning the first element of array A_{i+1} to the starting location of the next cache region.

Inserting the padding array will cause main memory overhead. Steps 3 and 5 are designed with the purpose of reducing the size of padding array. In step 3, we first find the array A_{max} that needs maximal size of padding array for allocating region for A_{max}. The larger the size of region allocated to A_{max}, the larger the size of padding array is needed. The array A_{max} will be interchanged with the last array variable in the declaration statement. This step guarantees that the size of padding array for A_{max} can be saved. In step 5, we further reduce the size of padding array by adjusting the position of arrays belonging to independent set. We use greedy method to select an independent array B_{best} that is with the most profit of reducing size of padding array for dependent set G_i. The independent array B_{best} is then partitioned and acted as several padding arrays to save memory overhead. To fix the size of region allocated to each dependent array, we should not change the position of arrays in a dependent set. To achieve, we now

treat a set of dependent array variables in P_{G_i} as a super array and change the position of the independent array variables among several super arrays.

Applying steps 3 and 5, we can achieve two goals that the size of padding arrays can be largely reduced and the cache conflicts can be avoided. As soon as the partitioned cache region allocated for array A_1 in the first level cache is known, we can easily evaluate the size of a tile and applying the loop blocking technique to enhance our cache partitioning algorithm. Localities existed in the execution of r inner loops thus can be exploited.

4 Performance Study

To measure the performance improvement, we use Atom as a tool for developing simulator. The simulator is designed for simulating the environment of multilevel direct-mapping cache. Comparisons of cache miss rates are made for the following different cache partitioning schemes.

(1) No padding array is applied. That is, the original cache direct mapping is applied, referred as Org in tables.
(2) Apply the equal-sized partitioning which is proposed in [4], referred as EP in tables.
(3) Apply the proposed cache partitioning and array element mapping technique, referred as CP in tables.
(4) Combine the proposed cache partitioning algorithm and loop tiling technique, referred as $Comb$ in tables.

Some factors such as cache size, the number of cache level, the number of iterations of innermost loop, and the array size are considered to be fixed or changed to observe the cache behavior. Applications such as matrix multiplication, subroutines of BLAS2 (Basic Linear Algebra Subprograms), and several numerical computation programs such as Fourier Least-Squares Approximation(FLSA), Jacobi Method for Solution of Linear Equations (Jacobi), Barycentric Form of Lagrange Interpolation (BFLI), Accumulating a Sum (ASum), Solving Linear Equation by Gaussian Elimination (SLEGE), Computing the Uniform Norm of Matrix A and A Inverse (Uniform norm), Gauss-Seidel Iterations (CSNCI), and Computing The Value of A Filtered Discrete Fourier Transform (FDFT) are selected as the source of loop programs. The improvements in cache conflicts for these applications have a similar behavior. To illustrate and analyze the cache behavior, we use Matrix Multiplication as a representation to describe the comparisons of different approaches (1), (3), and (4). The arrays declared for Matrix Multiplication are set to be a multiple of the size of the lowest-level cache.

Table 1 shows the cache miss rates of Org, CP, and $Comb$ by varying the number of the iterations of the innermost loop. The analysis is based on the environment of two-level cache. We fix the size of array to 2 Mbytes. The

Table 1. Comparisons of Org, CP, and $Comb$. We fix the size of the array and the size of two level cache, and vary the number of iterations of the innermost loop.

		Miss rates(%)				
Iterations		65536	32768	16384	8192	4096
Org	C^1	31.28	31.27	31.27	31.29	25.14
	C^2	25.06	25.07	25.08	25.10	25.14
CP	C^1	31.28	31.27	31.27	31.22	12.71
	C^2	12.68	0.319	0.326	0.34	0.364
$Comb$	C^1	15.19	15.19	15.19	7.686	0.521
	C^2	0.316	0.319	0.326	0.340	0.364

Table 2. Comparisons of Org, CP, and $Comb$. We fix the size of the array and the number of iterations of the innermost loop, and change the size of two level cache.

		Miss rates(%)			
C_s^1		64 KB	32 KB	16 KB	8 KB
C_s^2		1 MB	512 KB	256 KB	128 KB
Org	C^1	25.14	31.33	31.33	31.47
	C^2	25.14	25.14	25.14	25.14
CP	C^1	12.71	23.46	23.95	24.54
	C^2	0.364	4.69	8.23	15.96
$Comb$	C^1	0.521	7.635	15.32	15.45
	C^2	0.364	0.364	0.364	0.368

sizes of the first level cache and the second level cache are set by 64 Kbytes and 1 Mbytes, respectively. We have a large amount of improvement in miss rate by applying CP and $Comb$ techniques. When the number of iterations of the innermost loop becomes larger, CP improves little but $Comb$ have a significant improvement in cache miss rate of the first level cache. This is because that a large set of iterations accesses a large amount of elements which will replace the neighboring region and cause cache conflicts. However, combining loop tiling technique and the proposed cache partitioning technique can avoid this situation. On the second level cache, since each region has a larger size, CP and $Comb$ both have a significant improvement.

Table 2 shows the cache miss rates of Org, CP, and $Comb$ by varying the size of cache. The analysis is based on the environment of two-level cache. The size of array is set by 2 Mbytes and the number of iterations of the innermost loop is set by 4096. We have a great performance improvement by applying CP and $Comb$ in the condition that there is a large cache size. However, only $Comb$ technique has a stable cache miss rate (about 0.36) when cache size of both two levels becomes smaller. This is because that the proposed technique can tile the inner loops according to the size of each region. For the second level cache, since

Table 3. Comparisons of EP, CP, and $Comb$ for two-level cache. We fix the size of the array and the number of iterations of the innermost loop, and change the ratio of reference spaces.

ratio of reference space	Miss rates(%)					
	C^1			C^2		
	EP	CP	$Comb$	EP	CP	$Comb$
1:1	12.95	12.95	3.64	3.15	3.15	0.43
1:2	21.65	17.46	4.96	5.39	3.95	1.15
1:3	28.71	21.24	6.23	6.76	4.52	1.97
1:4	39.45	28.56	10.44	10.21	5.81	2.47
2:3	36.58	25.69	8.95	8.92	4.86	2.21
3:4	44.59	31.76	12.56	10.25	7.15	1.16

each region is with a larger size, accessed data will not replace the neighboring region. This makes the improvement significant.

In comparing the effects on cache miss rate under different cache partitioning techniques of (2), (3), and (4) applied, we vary the ratio of locality reference space of reference patterns ranging from 1:1 to 1:4. The size of arrays is set by 8 MBytes. The cache is considered as a two-level cache in which the first and the second level caches are with size 256 KBytes and 1 MBytes, respectively. The cache miss rate of our simulation is shown in Table 3.

In the condition that the ratio of reference spaces is 1:1, applying our technique will partition cache into equal regions. Thus, EP and CP have the same cache miss rate. The $Comb$ has smaller cache miss rate since loop tiling prevents the referenced elements to replace the neighboring region. In the cases that the size of reference space is not equal, CP and $Comb$ have a smaller cache miss rate. Noted that when the difference of size of reference space is getting large, cache miss rate raises. This is because that larger stride of memory access will have a poor cache performance since many elements move to cache without reference but occupy the cache space. In these cases, the $Comb$ and CP have smaller cache miss rates than EP.

The proposed technique can be considered as a generalized method of partitioning technique proposed in [4]. Another advantage is that, cooperated with the tiling technique, we can easily determine the block size according to the partitioned region. Localities existed in inner-nested loops, thus, can be exploited.

5 Conclusions

In this paper, we propose an algorithm to partition the multi-level cache into regions and to schedule the regions for arrays such that both temporal and spatial localities can be exploited and cache conflicts can be avoided. We treat cache as a system resource and schedule it to arrays according to the ratio of locality reference spaces. For multi-level caches, the partition of lower level cache should be careful so that cache region scheduling of a lower level cache can be consistent with the cache region scheduling of a higher level cache.

Besides, for reducing the memory overhead caused by applying padding array technique, we develop a greedy method to reposition both the dependent and independent arrays such that size of padding array can be reduced. To evaluate the performance of the proposed cache partitioning algorithm, we use Atom as a tool to develop a multi-level cache environment and simulate the cache behavior under direct mapping scheme. Performance study shows that the proposed cache partitioning scheme can largely improve the cache miss rate caused by cache conflict and exploit the reuse opportunities.

Acknowledgement:

This work was supported by National Science Council of the Republic of China under grant NSC 89-2213-E-156-001.

References

[1] K. Hwang and F. A. Briggs. Computer Architecture and Parallel Processing. McGRAW-Hill, Inc. 1984.

[2] M. Lam, E. E. Rothberg, and M E. Wolf. The cache performance of blocked algorithms. In *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 63-74, Apr. 1991.

[3] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, Vol. 27, No. 10, pp. 15-26, 1994.

[4] N. Manjikian and T. S. Abdelrahman. Reduction of cache conflicts in loop nests. Technical Report CSRI-318, Computer Systems Research Institute, University of Toronto, March 1995.

[5] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers*, Vol. 48, No. 2, Feb. 1999.

[6] O. Temam, C. Fricker, and W. Jalby. Impact of cache interferences on usual numerical dense loop nests. *Proceedings of the IEEE*, Vol. 81, No. 8, pp. 1103-1115, 1993.

[7] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 30-44, June 1991.