The Journal of Supercomputing, 22, 197–219, 2002

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

CHIH-YUNG CHANG changcy@email.au.edu.tw

Department of Computer and Information Science, Aletheia University, 32 Chen-Li St., Tamsui, Tapiei, Taiwan

JANG-PING SHEU AND HSI-CHIUEN CHEN {sheu, camus}@axp1.csie.ncu.edu.tw

Department of Computer Science and Information Engineering, National Central University, Chung-Li 32054, Taiwan

Abstract. This article 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 sizes, and then maps array data onto the scheduled cache positions to eliminate cache conflicts. A greedy method for rearranging array variables in declared statement is also developed, to reduce the memory overhead for mapping arrays onto a partitioned cache. Besides, loop tiling and the proposed schemes are combined to exploit opportunities for both temporal and spatial reuse. Atom is used as a tool to develop a simulation of the behavior of the direct-mapping cache to demonstrate that our approach is effective at reducing number of cache conflicts and exploiting cache localities. Experimental results reveal that applying the cache partitioning scheme can greatly reduce the cache conflicts and thus save program execution time in both single-level cache and multi-level cache hierarchies.

Keywords: cache conflict, array padding, cache partitioning, multi-level cache, direct mapping, loop tiling

1. Introduction

The growing speed gap between memory and processor demands that the memory systems are designed to follow a sophisticated memory hierarchy. Current high-performance computers rely on an efficient cache organization that consists of many levels of cache to reduce the speed discrepancy between processor and memory. 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 [4]. The placement policy specifies the mapping function from the main memory address to the cache location. Basically, three placement policies existed: direct, fully associative, and set associative. The direct mapping scheme maps blocks of main memory to a cache in a round robin fashion. That is, block i of the memory maps onto cache line i mod l where l is the number of cache lines in the cache. The direct mapping scheme has the advantage that no associative comparison is required and, hence, the cost of the process is reduced [4]. However, a disadvantage of the direct-mapping cache is that the cache hit ratio drops sharply when two or more blocks, accessed alternately, happen to map onto the same cache line in the cache.

Cache conflicts while executing nested loops in a scientific program cause data to be swapped out from the cache, causing cache misses and degrading performance. A high-performance compiler must pay great attention to preventing such conflicts. Much work has focused on inserting some padded arrays [8, 10, 12, 14, 18]. Padding arrays are inserted into arrays accessed in turn and mapped onto the same cache line under direct mapping policy, to ensure that such arrays map to different cache lines such that conflicts can be reduced. However, in the following two cases of loop programs, adding a small amount, say, the size of a cache line, of padding arrays can not prevent cache conflicts. Firstly, when the coefficients of the index variables of the reference pattern are unequal, accessing elements of a pair of two different arrays may induce cache conflict, even though a cache line amount of padding array has been inserted between these two arrays. Secondly, for loop programs that involve localities over many iterations, a padding array scheme can prevent conflicts but can not exploit the opportunities for reuse even if loop tiling is applied.

Manjikian et al. [10] presented an algorithm to partition a cache into equal-sized regions and to allocate each to a single array variable. During loop execution, access to elements with spatial locality or temporal locality yields good cache performance since each array has a region and will not be replaced by another array. However, treating the 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 it causes much memory fragmentation. To achieve cache partitioning, many padding arrays must be introduced to establish cache partitioning under a cache policy of direct mapping.

This work presents a cache partitioning technique to reduce the number of cache conflicts. The proposed scheme partitions the cache into many regions, generally with different sizes, and maps the array elements of the main memory onto the allocated region without requiring hardware support. The *locality reference space* is derived for each variable, and is defined by the set of elements that are accessed during the execution of some inner loops. The algorithm partitions the cache into many regions according to the locality reference space of reference patterns. It then automatically inserts padding arrays to map the array elements onto scheduled regions. The array variables are rearranged and the loop programs are reorganized to greatly reduce the memory fragmentation caused by applying padding scheme.

An algorithm is proposed for a multi-level cache memory system, to determine the size of the 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, a greedy method is developed to redeclare the arrays in a different order to minimize the memory overhead caused by applying the padding array scheme. The well-known loop tiling technique [3, 5, 12, 16, 17] and the cache partitioning and mapping scheme presented here are combined for the case in which the arrays in real applications are much larger than the partitioned cache region and in which data is reused after many iterations. Atom is employed as a tool to develop a simulation of the behavior of the direct-mapped cache to demonstrate that the approach presented here effectively reduces the number of cache conflicts and exploits cache

localities. Experimental results show that applying the cache partitioning scheme can greatly reduce cache conflicts and thus save program execution time in both single-level cache and multi-level cache hierarchies.

The rest of this study is organized as follows. Section 2 discusses the background and basic concept, with examples. Section 3 shows the loop model, cache model, preliminaries, and algorithms for cache partitioning and array element mapping. Section 4 describes related work in cache conflict prevention. Section 5 addresses the performance study of the partitioning algorithm. Conclusions are finally given in Section 6.

2. Basic concepts

This section uses examples to illustrate the basic concepts and the key idea behind the proposed techniques. Section 3 will presents the preliminaries and the general algorithm of cache partitioning and array element mapping.

The cache is a small high speed buffer memory which holds data that is likely to be accessed in the near future. In most modern computer systems, both cache and main memory are divided into equal-sized units, called blocks in the main memory and cache lines in the cache. During program execution, accessing an element involves transferring a section of data from the main memory to the cache. Clearly accessing these extra elements is desirable as no further main memory access will be needed for them. Data brought into cache should be reused as often as possible before they are replaced.

Most executions of loop programs present opportunities to reuse data. The regularity of array references and operations collected in loops typically gives temporal and spatial localities to the execution of loop nests. The reuse of data that remains in the cache is called temporal locality, and the use of the nearby data in a cache line is called spatial locality. Exploiting locality is essential to achieving high performance in most loop applications.

Conflict misses may occur when too many data items map to the same set of cache locations, causing cache lines to be flushed from the cache before they are reused. Conflict misses have been found to be a significant source of poor performance in scientific programs, particularly within loop nests. Consider, for example, 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 employs 8 bytes to store a floating point number and each cache line has a capacity of four floating point elements (32 bytes). Thus, the memory space for storing array A is 128*128*8 = 128 KB. Cache conflicts occur at each reference during execution of loop $L1$ when the size of the direct-mapping cache is 16K, 32K, 64K, or 128K bytes, because that the cache size is a

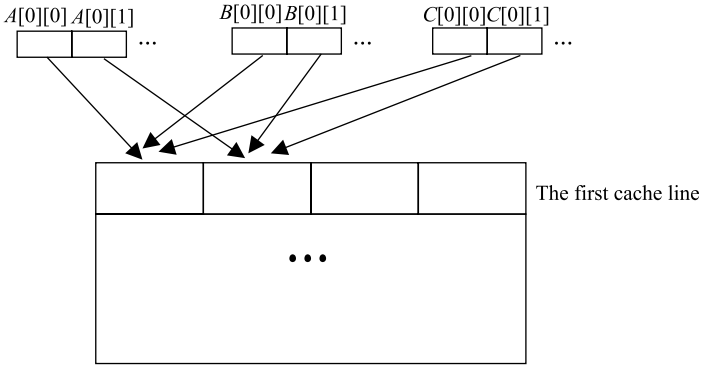


Figure 1. Cache conflict occurs when starting elements of arrays A, B, and C are mapped to a single cache line and are accessed during a single iteration.

factor of the size of arrays A, B, and C. The directly mapped cache maps array elements $A[I][J]$, $B[I][J]$, and $C[I][J]$ onto a single cache line, for specific values of I and J, as shown in Figure 1.

For instance, when executing index variables $I = 0$ and $J = 0$, statement S_1 firstly accesses element $A[0][0]$. The four elements $A[0][0]$, $A[0][1]$, $A[0][2]$, $A[0][3]$ are transferred from main memory to a cache line, as shown in Figure 2(a). The extra elements, $A[0][1]$, $A[0][2]$, and $A[0][3]$ are automatically transferred and are expected to be accessed during the execution of iterations $(I, J) = (0, 1), (0, 2)$, and $(0, 3)$, respectively.

However, the reference of array element $B[0][0]$ causes four elements $B[0][0]$, $B[0][1]$, $B[0][2]$, and $B[0][3]$ to move from the main memory to the same cache line to which the element $A[0][0]$ was mapped. As shown in Figure 2(b), elements $A[0][1]$, $A[0][2]$, $A[0][3]$ are replaced before they are used. Similarly, statement S_1 accesses element $A[0][1]$ while executing index variables $I = 0$ and $J = 1$. The four elements, $A[0][0]$, $A[0][1]$, $A[0][2]$, and $A[0][3]$ are again transferred from the main memory to a cache line. The array elements $B[0][1]$, $B[0][2]$, and $B[0][3]$ are replaced before they are used. The same array elements are repeatedly swapped between the cache and the main memory. The repeated swapping follows mainly from the fact that, for specific values I and J, array elements $A[I][J]$, $B[I][J]$, and $C[I][J]$, are mapped onto the same cache line and are accessed while executing a single loop iteration.

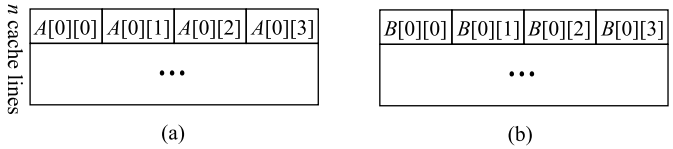


Figure 2. The contents of the first cache line in execution of loop $L1$. (a) Reference of array element $A[0][0]$. (b) Reference of array element $B[0][0]$.

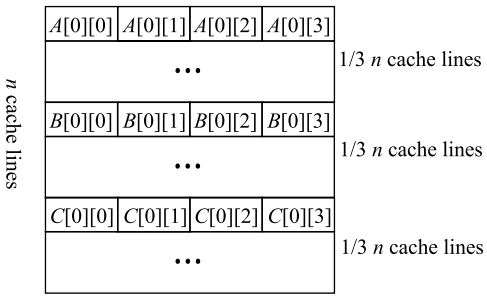


Figure 3. The mapping of arrays A, B, and C onto partitioned cache by applying equal-sized partitioning technique.

Manjikian et al. [10] offered a partitioning algorithm to divide the cache into many equal-sized regions and apply the *padding technique* [8][10][12][14][18] to map array data onto the partitioned regions. Assume that a direct mapped cache includes n cache lines. Manjikian's cache partitioning approach partitions the cache into three equal-sized regions for arrays A, B, and C. Figure 3 shows the mapping of arrays A, B, and C onto the cache, given by applying the equal-sized partitioning scheme to Example 1.

Partitioning cache into many regions and allocating one region to one array can prevent cache conflicts in Example 1. However, treating the cache as a resource of the system, the compiler should partition cache into regions according to the working set of each array. The cache must be partitioned into many regions with different sizes because the working set of reference patterns are possibly unequal. 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:$A[I + 4 * J] = B[I + 2 * J];$ $\qquad (L2)$

For simplicity, single-level cache with 12 cache lines is assumed; each line has a capacity for two array elements. The corresponding elements of arrays A and B are mapped to a single cache line, since the cache size is a factor of size of array A. Iterations $(I, J) = (0, 0)$ and $(I, J) = (1, 0)$ will respectively access elements $B[0]$ and $B[1]$ which are located in a single cache line. The reference of element $B[0]$ in iteration $(I, J) = (0, 0)$ causes elements $B[0]$ and $B[1]$ to move from the memory to the cache. Loop tiling technique is applied to loop $L2$ to exploit the spatial locality. Loop $L2$ is then modified as the following loop $L2'$, if the innermost loop, J, is tiled with blocks each with a size of four iterations.

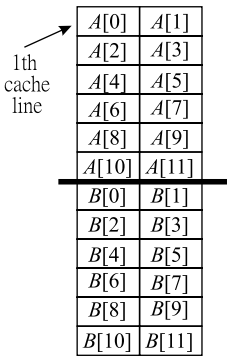
S_1:float $A[504]$, $B[504]$;
for $(J' = 0; J' < 100; J' = J' + 4)$
  for $(I = 0; I < 100; I{+}{+})$
  for $(J = J'; J < J' + 4; J{+}{+})$
    S_2:$A[I + 4 * J] = B[I + 2 * J];$ $\qquad (L2')$

Applying the equal-sized cache partitioning technique proposed in [10], the compiler inserts a padding array $P[12]$ between arrays A and B and redeclares S_1 as

float $A[504]$, $P[12]$, $B[504]$

such that arrays A and B can be mapped onto the equal-sized cache regions. Figure 4 shows the equal-sized cache partition. The cache is partitioned into two parts and arrays A and B are allocated one to each part. Each partitioned cache is comprised of six cache lines and has a capacity for 12 array elements. In Figure 4, the first and the second columns display the running iteration. The third and the fourth columns show the accessed element and the cache contents, respectively, for a particular iteration.

For example, statement S_2 accesses array element $B[0]$ during the execution of $(I, J) = (0, 0)$. The cache memory management system transfers elements $B[0]$ and $B[1]$ from the main memory to the 7th line of the cache. However, the reference of array element $A[12]$ causes elements $A[12]$ and $A[13]$ to be moved from the main memory to the 7th cache line during the execution of $(I, J) = (0, 3)$. The array element $B[1]$, which is expected to be reused in iteration $(I, J) = (1, 0)$, is replaced by $A[13]$, primarily because the space of accessed elements in array A is larger than that in array B, during the execution of the innermost loop. The space of accessed elements in array A is $A[0 : 12]$ ($A[0 : 12]$ is accessed during the execution of $I = 0$, $0 \leq J \leq 3$) whereas the space of accessed elements of array B is $B[0 : 6]$. Notably, in the third and fifth columns of Figure 4, a '*' symbol indicates that the reference of the element can be obtained from the cache before the element is replaced. That is, the reuse opportunity due to spatial locality is exploited. During the execution of the innermost loop, only six reuse opportunities are exploited by applying equal-sized cache partitioning technique.



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
0	2	$B[4]$	$B[4]$,$B[5]$ in 9th cache line	$A[8]$	$A[8]$,$A[9]$ in 5th cache
0	3	$B[6]$	$B[6]$,$B[7]$ in 10th cache line	$A[12]$	$A[12]$,$A[13]$ in 7th cache
1	0	$B[1]$	$B[0]$,$B[1]$ in 7th cache line	$A[1]$*	$A[0]$,$A[1]$ in 1th cache
1	1	$B[3]$*	$B[2]$,$B[3]$ in 8th cache line	$A[5]$*	$A[4]$,$A[5]$ in 3th cache
1	2	$B[5]$*	$B[4]$,$B[5]$ in 9th cache line	$A[9]$*	$A[8]$,$A[9]$ in 5th cache
1	3	$B[7]$*	$B[6]$,$B[7]$ in 10th cache line	$A[13]$	$A[12]$,$A[13]$ in 7th cache

Figure 4. The equal-sized cache partitioning and mapping of arrays A and B for Loop $L2'$.

The replacement of the cache region of array B by elements of array A can be prevented in two ways. First, the size of the tiling block can be reduced. Second, a larger cache region can be scheduled for array A. Array A does not occupy the cache region of array B if the first method is employed to tile loop $L2$ with a smaller block size, say 3 iterations, during the execution of one tiling block. However, three cache lines of the region scheduled for array B are not used. Partitioning the cache into regions of different sizes is a feasible means of fully exploiting the utilization of cache resources and prevent missing cache, due to the replacement of array A. Partitioning the cache into regions, possibly of different sizes, according to the space accessed by each array variable, is thus encouraged.

The array variables that are mapped to the same cache line are firstly identified. The cache is partitioned into many regions with various sizes, according to the space accessed by these array variables. Each partitioned region is assigned to an array variable to prevent replacing the preloading elements before they are accessed. The tiling size is determined according to the cache space partitioned for each array variable and the loop tiling is applied to exploit the reuse opportunities. For example, consider loop $L2$ in Example 2. Corresponding elements of arrays A and B will be mapped to the same cache line since the cache size is a factor of the size of arrays A and B. The cache is partitioned into two regions for arrays A and B. The space accessed by array A for the innermost loop J is

$$A[4 * J], 0 \leq J \leq 99,$$

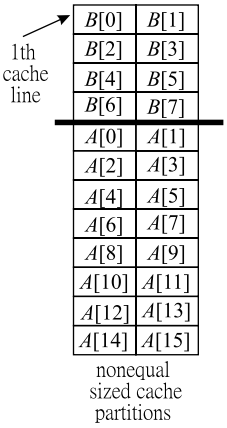
equal to the working set $A[0 : 396]$. Similarly, the space accessed by array B for the innermost loop, J, is $B[0 : 198]$. The ratio of the reference space of array A to that of array B is 2:1. Thus, the cache is partitioned into two regions according to the ratio of the reference spaces. That is, $(2/3 * 12) = 8$ cache lines are allocated to array A and $(1/3 * 12) = 4$ cache lines are allocated to array B. A padding array must be inserted and statement S_1 should be redeclared to map arrays A and B onto allocated cache regions under direct mapping policy. Two possible declarations are as follows.

S_1 : float $A[504], P[16], B[504]$ or

S_1 : float $B[504], P[8], A[504]$.

The second declaration is chosen as it reduces the size of memory fragmentation caused by applying padding array technique. Figure 5 depicts the partitioned cache and the mapping of arrays A and B.

The well-known tiling technique is further employed to cooperate with the cache partitioning technique to exploit the reuse opportunities of executing the outer loop. The tiling size is set to four to guarantee that no cache data is replaced by elements of array A during the execution of the innermost loop since the size of the region allocated to array B is four cache lines containing eight floating elements. The reuse opportunities in executing successive outer loop iterations can thus be



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
0	2	$B[4]$	$B[4]$,$B[5]$ in 3th cache line	$A[8]$	$A[8]$,$A[9]$ in 9th cache
0	3	$B[6]$	$B[6]$,$B[7]$ in 4th cache line	$A[12]$	$A[12]$,$A[13]$ in 11th cache
1	0	$B[1]$*	$B[0]$,$B[1]$ in 7th cache line	$A[1]$*	$A[0]$,$A[1]$ in 5th cache
1	1	$B[3]$*	$B[2]$,$B[3]$ in 2th cache line	$A[5]$*	$A[4]$,$A[5]$ in 7th cache
1	2	$B[5]$*	$B[4]$,$B[5]$ in 3th cache line	$A[9]$*	$A[8]$,$A[9]$ in 9th cache
1	3	$B[7]$*	$B[6]$,$B[7]$ in 4th cache line	$A[13]$*	$A[12]$,$A[13]$ in 11th cache

Figure 5. The nonequal-sized cache partitioning and mapping of arrays A and B for loop $L2$.

exploited. Applying the cache partitioning technique to Example 2 yields 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:$A[I + 4 * J] = B[I + 2 * J];$ $\quad (L2'')$

The unequal-sized partitioning scheme has two advantages over the equal-sized cache partitioning. First, two cache lines of memory fragmentation are saved. That is, the size of padding arrays is reduced from $P[12]$ to $P[8]$. Second, all the preloading elements can be accessed before they are replaced. Figure 5 presents this situation. For example, during the execution of $(I, J) = (0, 0)$, the reference of $B[0]$ causes elements $B[0]$ and $B[1]$ to move from the main memory to the first cache line. The preloading element $B[1]$ is accessed in iteration $(I, J) = (1, 0)$, because a larger cache region is allocated to array A such that the accessed elements of array A do not replace the preloading element of array B during execution of iterations of the innermost loop. As shown in Figure 5, eight preloading elements (with a '*' symbol) are accessed before they are replaced. The spatial locality of two more elements are exploited than in equal-sized partitioning.

This section uses an artificial example to depict the fundamental idea of cache partitioning to reduce the number of cache conflicts and exploit the cache localities. The following section presents preliminaries and the multi-level cache partitioning algorithm.

3. Preliminaries and the algorithms

This section proposes a cache partitioning and array mapping algorithm for single-level and multi-level cache hierarchies. First, the cache is partitioned into many regions according to the accessed space of each array reference pattern. Then the order of array variables is resorted, padding arrays are inserted in their proper positions, and array variables are redeclared such that array elements can be mapped to the partitioned regions and the memory overhead can be minimized. Finally, cache partitioning and loop blocking (or tiling) are combined to exploit localities in loop execution.

Assume that the data mapping from main memory to cache uses direct-mapping as a cache placement strategy. The strategy is fast and simple but tends to causes cache conflicts. The loop program inputs to the partitioning algorithm can be nested loops. The reference patterns within the body of a loop can be multi-dimensional array variables with an affine function. The program model, L, considered here is expressed as follows.

Program Model:

declaration of array variables $A_i, 1 \le i \le k$ and array variables $B_j, 1 \le j \le m$
for $(I_1 = l_1; I_1 \le u_1; I_1{+}{+})$
  for $(I_2 = l_2; I_2 \le u_2; I_2{+}{+})$
  ...
    for $(I_n = l_n; I_n \le u_n; I_n{+}{+})$
      { statements $A_i[f(I_1, I_2, \ldots, I_n)]$, for $1 \le i \le k$; } $\qquad (L)$

where f is the affine function of q-dimensional array A_i and could be defined by

$$f = (a^1_{i1} I_1 + \cdots + a^1_{in} I_n, a^2_{i1} I_1 + \cdots + a^2_{in} I_n, \ldots, a^q_{i1} I_1 + \cdots + a^q_{in} I_n).$$

Array variables are said to be in a *dependent set* if their first elements are mapped to the same cache line and their reference patterns access the same cache line while executing the innermost loop or tiling loop. Variables not belonging to any dependent set are said to be in an *independent set*. In program model L, k arrays are assumed to be in dependent sets and m arrays are assumed to be in independent sets. This paper considers an n-nested loop program, as shown in loop L. Most applications fall in this loop model. For a row-major memory system, the spatial and temporal localities are evident in successive accessing of the rightmost dimension of an array variable. Without loss of generality, and for simplicity, all arrays A_i are assumed to be one-dimensional. For real applications that access multi-dimensional arrays, the algorithm presented here only exploits localities of array references in the rightmost dimension. Therefore, the reference pattern $A_i[f]$ in program model L can be simplified as

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

The number of array variables in a loop body is assumed to be $m + k$, where k and m denote the number of arrays in dependent sets and in independent sets, respectively. Notably, the body of the loop may contain many statements.

The following artificial example is used to illustrate the algorithm in detail. In the next section, the proposed cache partitioning technique is applied to evaluate real programs. Consider the following loop 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++)
        S1: A[I + J] = B[J];          (L3)
...
for (I = 0; I < 128000; I++) {
        S2: D[I] = C[I] + D[I];       (L4)
        S3: E[I] = C[I] + E[I]; }
```

Assume that the cache size is 64000 words. Two reference patterns $A[I + J]$ and $B[J]$ exist in loop $L3$. The terms A_1 and A_2 in program model L, denote arrays A and B, respectively, and coefficients a_{11}, a_{12}, a_{21}, and a_{22} take the values 1, 1, 0, and 1, respectively. 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, since the size of array variables A, B, D, and E are multiples of cache size, as presented in Figure 6. In Example 3, two dependent sets $\{A, B\}$ and $\{D, E\}$ respectively belong to loops $L3$ and $L4$. Similarly, the independent set of Example 3 is $\{C\}$. Thus, for loop $L3$ of Example 3, the number of arrays in the dependent set is $k = 2$. For loop $L4$, $k = 2$ and $m = 1$.

A few terms, used throughout this study are defined for clarity.

C^i: the ith level cache.
C_s^i: the size of the ith level cache measured by number of words.
A_i: the array variables in the dependent set.

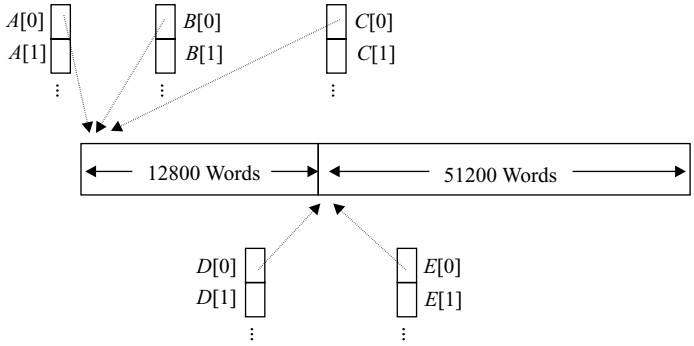


Figure 6. Cache mapping of arrays A, B, C, D, and E for loop $L3$.

B_j: the array variables in the independent set.
S_{A_i}: the size of array A_i measured by number of words.
$C^j_{A_i}$: the size of region allocated to array A_i in the jth level cache.
P_{A_i}: the required 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, $1 \le i \le n$.
a_{ij}: coefficient of index variable 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 required for aligning arrays in set G_i.

Note that we have

$$P_{G_i} = \sum_{x=1}^{|G_i|} |P_{A_x}|, \quad \forall A_x \in P_{G_i}.$$

These terms are used in discussing single-level cache and multi-level caches below.

3.1. *Single-level cache hierarchy*

This section describes how a single-level cache is partitioned into k regions and k arrays are mapped 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 required to exploit the spatial and temporal localities during execution. The *locality reference space* of array A_i is defined by the set of elements that are accessed during the execution of r inner loops. The locality reference space is determined by the loop index variables $I_{n-r+1}, \ldots, I_{n-1}, I_n$. In the program model, L, the locality reference space R_{A_i} of array A_i can be derived since the coefficients of r inner loop index variables $I_{n-r+1}, \ldots, I_{n-1}, I_n$ are $a_{i(n-r+1)}, \ldots, a_{i(n-1)}, a_{in}$, respectively:

$$R_{A_i} = \{A_i(x) | x = a_{i(n-r+1)} I_{n-r+1} + \cdots + a_{in} I_n, \\ l_j \le I_j \le u_j, n - r + 1 \le j \le n\}.$$

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

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

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

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

The following loop program is transformed by applying tiling technique to loop $L3$.

```
int A[128000], B[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)
```

In this example, loops I and J are tiled with a block size 166 * 166. Assume that the cache level $N = 1$. The number of dependent arrays is $k = 2$ and the depth of the loops is $n = 4$. The index variables I_3 and I_4 in loop model L respectively represent I and J in $L5$. Array variables A_1 and A_2 respectively stand for variables A and B. The coefficients of array variables A and B are $a_{11} = 0$, $a_{12} = 0$, $a_{13} = 1$, $a_{14} = 1$, $a_{21} = 0$, $a_{22} = 0$, $a_{23} = 0$, and $a_{24} = 1$. Assume that the localities are exploited during the execution of loops I and J. Accordingly, the locality reference space of array A is $A[0:330]$ which can be derived as

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

Similarly, the locality reference space of array B is $B[0{:}165]$. Hence, the ratio of the locality reference space of array A to that of B is 2:1. Assume the size of the single-level cache is C_s^1. The cache is divided into 3 partitions. Array A is assigned to a region of two partitions, and array B is assigned to a region of the remaining partition. Thus, the sizes of the cache regions for A and B are $C_A^1 = 2C_1^s/3$ and $C_B^1 = C_1^s/3$, respectively. The first statement of loop $L5$ is rewritten to reduce memory fragmentation:

int $B[128000]$, $P_B[C_1^s/3]$, $A[128000]$.

Loop $L5'$ is the transformed loop.

```
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*

This section considers multi-level cache architecture. The size of the partitioned cache region of the ith-level cache maybe a multiple of the size of the $(i-1)$th-level cache which has faster access. Consider a memory hierarchy with two-level caches. Partitioning the second level cache into several regions and allocating them to array variables does not guarantee that data mapped from the region of the second level cache to that of the first level cache will be placed in the region scheduled for the first level cache. This section proposes criteria for partitioning the lower level cache such that data swapped from a lower level cache to a higher level cache is guaranteed to be mapped to the correct region.

Consider loop $L5$. Let the cache memory hierarchy be composed of two-level cache, as is the cache hierarchical design in the most modern computer. Assume that the size of the first level cache is $C_s^1 = 100$ words and that of the second level cache is $C_s^2 = 500$ words. As presented in loop $L5$, the sizes of arrays A and B are a multiples of 500 words. Figure 7(a) shows the address mapping of arrays A and B. The two-level cache is partitioned for the execution of loop $L5$.

The ratio of the reference space of pattern $A[I+J]$ to that of $B[J]$ can be easily determined by considering the coefficients, since both the index variables I and J are varying from 0 to 165. That is, the ratio of locality reference space of arrays A to that of array B is 2:1. The cache partitioning method partitions the first level cache into two regions. The first region has size $C_A^1 = 67$ words, and is allocated to array A; the second region has size $C_B^1 = 33$ words and is allocated to array B. If cache memory system contains only a single-level cache, the positions of arrays A and B in the declaration statement are exchanged and a padding array $P_B[33]$ is inserted between arrays B and A. Figure 7(b) gives the address mapping of arrays A and B.

If the cache system is organized as a two-level cache, the second level cache is partitioned into two regions. The first region has size $C_A^2 = 334$ for array A and the second region has size $C_B^2 = 166$. Figure 8 depicts the partitioning of two regions of

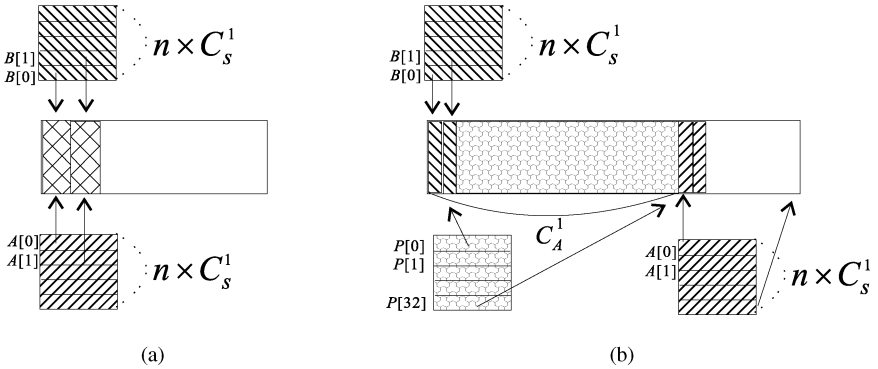


Figure 7. Single-level cache partition for address mapping of arrays A and B. (a) Address mapping of arrays A and B before executing the cache partitioning. (b) Address mapping of arrays A and B after executing the cache partitioning.

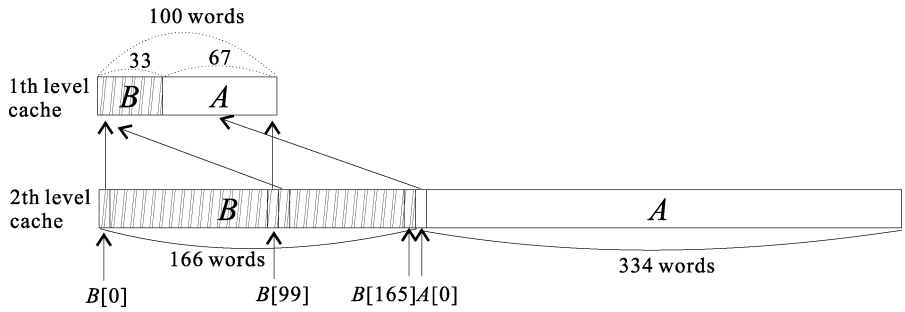


Figure 8. Partitioning of the second level cache can not map the right element onto the correct position of the first level region.

both the first and the second level caches. The cache is partitioned into two regions, according to the ratio of reference space of arrays A and B, one-third of the cache is allocated to array B and the remainder is allocated to array A. However, using this ratio to partition the second level cache raises the problem that the first cache line of the region allocated to array A in the second level cache, is not directly mapped onto the first cache line of the cache region allocated to array A in the first level cache. As Figure 8 shows, the first element of array A is scheduled in the 166th cache word of the second level cache. This element is directly mapped to the 66th cache word in the first level cache. However, according to the scheduling described here, the first element of array A in the first level cache is mapped to the 34th cache word. The size of the region allocated to array B in the second level cache should be reduced to the partition of the second level cache to satisfy the scheduling of the first level cache. Hence, the cache size allocated to 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 result implies that the padding array for the two-level cache is $P[133]$ rather than $P[166]$.

When the cache memory hierarchy is organized as a multi-level cache, the partition for the lower level cache must address not only the ratio of locality reference spaces of dependent array variables, but also the mapping from the lower level cache to the higher level cache. The partitioning of the ith-level cache should also guarantee that, for a specific array variable A_k, the direct mapping of the starting address of A_k in the region of the ith-level cache to the starting address of A_k in the region of the $(i-1)$th-level cache, should be consistent with the partition for the $(i-1)$th-level cache. Let a loop contains k dependent array variables $A_1 \ldots A_k$, whose locality reference spaces are $R_{A_1} \ldots R_{A_k}$, respectively. The first level cache is partitioned into k regions where the ith-region has size

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

and is allocated to array A_i. Similarly, for the second level cache, cache size $C^2_{A_i}$ is allocated to array A_i, where

$$C^2_{A_i} = \left\lfloor \left(|R_{A_i}| \Big/ \sum_{i=1}^{k} |R_{A_i}| \right) \times C^2_s \right\rfloor.$$

However, the value of $C^2_{A_i}$ should be adjusted to ensure that the first element of the region of $C^2_{A_i}$ maps to the starting location of $C^1_{A_i}$:

$$C^2_{A_i} = \lfloor C^2_{A_i} / C^1_s \rfloor \times C^1_s + C^1_{A_i}.$$

The size of the padding array required to allocate $C^2_{A_i}$ to array A_i is thus,

$$P_{A_i} = C^2_{A_i} - (S_{A_i} \bmod C^1_s).$$

These two expressions are substituted into the previous example, for verification. A_1 and A_2 are taken to be arrays B and A, respectively. The sizes of the first level and the second level caches are $C^1_s = 100$ and $C^2_s = 500$, respectively. The locality reference spaces of arrays B and A are,

$$R^s_B = \{J | 0 \le J \le u_J - 1\}, \qquad R^s_A = \{I + J | 0 \le J \le u_J - 1, 0 \le I \le u_I - 1\},$$

where $u_J = u_I$ specifies the upper bound of loop index variables I and J in loop $L5$. Therefore, the size of the first level cache allocated to array B is

$$C^1_B = \lfloor (|R_B| / (|R_A| + |R_B|)) \times C^1_s \rfloor = \lfloor 1/3 * 100 \rfloor = 33.$$

Similarly, the size of the first level cache allocated to array A is $|C^1_A| = 67$. For the second level cache, the size allocated to array B is

$$C^2_B = \lfloor C^2_B / C^1_s \rfloor \times C^1_s + C^1_B = \lfloor 166/100 \rfloor \times 100 + 33 = 133.$$

Similarly, the size of the second level cache allocated to array A is 367. Therefore, the size of the padding array is

$$P_B = C^2_B - (S_B \bmod C^1_s) = 133 - 0 = 133.$$

The array statement is then redeclared as,

float B[128000], P[133], A[128000]

rather than the original statement,

float A[128000], B[128000].

The following subsection proposes the multi-level cache partitioning and array element mapping algorithm.

3.3. The algorithm

This section combines techniques proposed in the previous subsections and proposes the new partitioning algorithm.

Algorithm: Multi-Level Cache Partitioning and Array Element 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:
$C^j_{A_i} = C^j_s \times |R_{A_i}| / \sum_{i=1}^{k} |R_{A_i}|$;
/* multi-level caches consideration */
(2.2) if $(j > 1)$
$C^j_{A_i} = \lfloor C^j_{A_i} / C^{j-1}_s \rfloor \times C^{j-1}_s + C^{j-1}_{A_i}$;
} /* end of for j */
(2.3) Compute the size of padding array P_{A_i}:
$P_{A_i} = C^j_{A_i} - (S_{A_i} \bmod C^j_s), 1 \le i \le k-1$;
if $(P_{A_i} < 0)$
$P_{A_i} = P_{A_i} + C^j_s$;
}
Step 3 : /* reducing the size of padding array by moving */
/* array with maximal padding size to the last position */
Let $C^N_{A_{max}} = Max(C^N_{A_i}), 1 \le i \le k$.
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 $M \langle\rangle \varnothing$
{ Let B_{best} and $P_{G_{best}}$ be the pair of best values that satisfy:
$P_{G_{best}} - (|B_{best}| \bmod C^N_s) \le P_{G_i} - (|B_j| \bmod C^N_s), \forall B_j \in M$,
$\forall P_{G_i} \in P$.
$M = M - B_{best}$, $P = P - P_{G_{best}}$;

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 to exploit the reuse opportunities */
According to the size of region of the first level cache allocated to A_1, determine the tiling size and apply the loop tiling technique to L.

Step 1 of the algorithm initializes the temporal variables to calculate the size of padding array, P_{A_i}. Step 2.1 determines $C^j_{A_i}$ which is the size of each region allocated to array A_i in the jth level of the cache. The size $C^j_{A_i}$ is obtained by the ratio of the locality reference space of A_i to the summation of locality reference spaces of all dependent arrays. A larger cache space is allocated to A_i if the locality reference space of array A_i is larger than one of array A_j in the inner r loops. The cache size is treated as a system resource and the resource is partitioned into many regions according the size of the working set of array elements accessed in the inner r loops. The region of A_i can thus be derived:

$$C^j_{A_i} = C^j_s \times \left(|R_{A_i}| \Big/ \sum_{j=1}^{k} |R_{A_j}| \right).$$

Additional computation is required in Step 2.2 to guarantee that the partitioning of the lower level cache can match that of the higher level cache. Step 2.3 determines the size P_{A_i} of the padding array to preserve the region for A_i and align the first element of array A_{i+1} to the starting location of the subsequent cache region.

Inserting the padding array increases the main memory overhead. Steps 3 and 5 are designed to reduce the size of the padding array. Step 3 first finds the array A_{max} that requires the largest padding array to allocate region to A_{max}. A larger region allocated to A_{max} demands a larger padding array. The array A_{max} is exchanged with the last array variable in the declaration statement. This step ensures that the size of padding array for A_{max} can be reduced. Step 5 further reduces the size of the padding array by adjusting the position of arrays belonging to independent sets. The greedy method is used to select an independent array B_{best} that most profitably reduce the size of the padding array for the dependent set, G_i. The independent array B_{best} is then partitioned and acts as many padding arrays to reduce memory overhead. The position of arrays in a dependent set should be maintained to fix the size of region allocated to each dependent array. A set of dependent array variables in P_{G_i} is then treated as a super array and the positions of the independent array variables are changed among several super arrays.

Applying Steps 3 and 5 significantly reduces the size of padding arrays and prevents cache conflicts. As soon as the partitioned cache region allocated for array A_1 in the first level cache is known, the size of a tile can be easily evaluated and loop blocking technique can be applied to enhance the cache partitioning algorithm. Localities in the execution of r inner loops can thus be exploited.

4. Related work

Array padding [1] is a data-reorganization technique that increases the size of the array dimension aligned with the storage order, and most effectively reduces the occurrence of self-conflicts when the array dimensions are powers of two. Array padding is employed to align the array to the starting address of its region, and thereby implement the proposed cache partitioning technique. Since the size of each partitioned region is not equal, the padding arrays for each dependent array variable are unequally sized. The use of array padding causes much memory fragmentation. The method in Steps 3 and 5 of the algorithm are proposed to reduce memory overhead.

Temam et al. [15] examine conflicts arising from array references in loop nests, typical of scientific applications. The authors analyze specific instances of self- and cross-conflicts, and suggest the use of padding and careful placement of arrays in the memory. However, no detailed methodology is offered for resolving conflicts. Bacon et al. [2] discuss a method to determine the amount of padding required to avoid cache and TLB mapping conflicts among individual array references in the innermost loop of a loop nest. However, their approach is inappropriate for locality-enhancing loop transformations, as it does not address data reuse in outer loops, and therefore cannot prevent cache conflicts for reusable data. The cache partitioning algorithm described here can work with loop tiling such that localities in inner loops can be exploited. Criteria are offered to guarantee valid mapping from a lower level cache to a higher level cache and thus prevent cache conflict in a multi-level cache system.

Lebeck and Wood [7] present a case study of improving performance through a variety of techniques, including data transformations such as padding and memory alignment. However, these transformations are discussed in the context of programmer tuning of application performance. There is no discussion of how such transformations may be incorporated into a compiler. Systematic method is proposed here, to apply padding to cache partitioning. The combination of the proposed method and the tiling technique is also discussed. The nonequal-sized cache partitioning technique provides a greedy algorithm to adjust the position of independent arrays and thereby reduces the memory fragmentation caused by implementing the padding array.

Manjikian et al. [10] present a cache partitioning algorithm to partition the cache into equal-sized regions to prevent cache conflicts. However, the cache is treated as a resource. The cache size is scheduled according to the locality reference space. Compared to their equal-sized partition, memory fragmentation is greatly reduced and the locality of the execution of multiple inner loops is exploited, providing the cooperation between the technique presented here and the loop tiling technique. For programs with nonunit stride access patterns, unequally sized partitioning markedly reduces cache conflicts.

5. Performance study

Atom is used as a tool to develop a simulator to measure the performance improvement. The environment of the multilevel direct-mapping cache is simulated. Cache miss rates are compared for the following cache partitioning schemes.

(1) No padding array is applied. That is, the original cache direct mapping is applied. The tables refer to this scheme as *Original*.
(2) Apply the equal-sized partitioning, proposed in [10]. The tables refer to this scheme as *EP*.
(3) Apply the proposed cache partitioning and array element mapping technique. The tables refer to this scheme as *CP*.
(4) Combine the proposed cache partitioning algorithm and the loop tiling technique. The tables refer to this scheme as *Combination*.

Some factors such as cache size, the number of the cache level, the number of iterations of the innermost loop, and the array size are considered to be fixed, or are 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 chosen as the source of loop programs. The improvements in cache conflicts for these applications are similar. Matrix Multiplication is used as a representation to compare different approaches (1), (3), and (4), to illustrate and analyze the cache behavior. The ratio of locality reference spaces is varied and the miss rate of two-level cache is measured to compare cache conflicts by applying cache partitioning techniques (2), (3), and (4).

Matrix Multiplication, which is a basic operation of many scientific programs, is used to illustrate the experimental results by applying the proposed cache partitioning technique. The arrays declared for Matrix Multiplication are set to multiples of the size of the lowest-level cache.

Table 1 gives the cache miss rates of *Original*, *CP*, and *Combination* for execution by varying the number of iterations in the innermost loop. The analysis is based on the environment of the single-level cache. The size of the array is set to 2 Mbytes

Table 1. Comparing cache miss rates for *Original*, *CP*, and *Combination*. The sizes of array and cache are fixed, and the number of iterations of the innermost loop is varied

	Miss rates (%)				
Iterations of the innermost loop	65536	32768	16384	8192	4096
Original	25.06	25.07	25.08	25.10	25.14
CP	12.68	0.31	0.32	0.34	0.360
Combination	0.32	0.31	0.32	0.34	0.36

and the size of the cache is set to 1 Mbytes. *CP* and *Combination* techniques exhibit significant performance improvements over the *Original* technique, when the number of iterations of the innermost loop is small. These improvements are due to avoiding many cache conflicts when arrays are mapped onto the scheduled regions. The opportunities for reuse of data can thus be exploited as much as possible. The improvement of *CP* decreases whereas that of *Combination* remains significant when the number of iterations of the innermost loop increases. The result follows mainly from the fact that the number of accessed elements exceeds the size of the scheduled cache region. This situation causes the array elements to replace elements in the neighboring region, resulting cache conflicts.

The effect of cache size on cache performance for a single-level cache is determined by fixing other factors and varying the cache size from 128 K-bytes to 1 M-byte. Table 2 gives the cache miss rates of *Original*, *CP*, and *Combination* to execute matrix multiplication with various sizes of cache. The size of each array is set to 2 Mbytes, and the number of iterations of the innermost loop is set to 65536. As shown in Table 2, the performance improvements are significant when *CP* and *Combination* are applied and the cache size is enlarged, because the larger region maintains more array elements, and thus reduces cache conflicts. Under such condition, data are more likely to be reused in the cache region.

Table 3 presents the cache miss rates of *Original*, *CP*, and *Combination*, obtained by varying the number of the iterations of the innermost loop. The analysis is based on the environment of the two-level cache. The size of the array is set to 2 Mbytes. The sizes of the first level cache and the second level cache are set to 64 Kbytes and 1 Mbytes, respectively. The miss rate is greatly reduced by applying *CP* and *Combination* techniques.

When the number of iterations of the innermost loop increases, *CP* slightly improves but *Combination* significantly improves the cache miss rate of the first level cache. This result follows from the fact that a large set of iterations accesses many elements, replacing the neighboring region and causing cache conflicts. However, combining loop tiling and the proposed cache partitioning can prevent such a situation. *CP* and *Combination* both show significant improvements for the second level cache.

Table 4 presents the cache miss rates of *Original*, *CP*, and *Combination*, found by varying the size of the cache. The analysis is based on the environment of the two-level cache. The size of the array is set to 2 Mbytes and the number of iterations of the innermost loop is set to 4096. Applying *CP* and *Combination* greatly enhances performance for a large cache. However, only *Combination* technique gives a stable cache miss rate (approximately .36) when the cache of both two levels shrinks,

Table 2. Comparisons of *Original*, *CP*, and *Combination*. The size of the array and the number of iterations of the innermost loop are fixed, and the size of the cache is varied

	Miss rates (%)			
The size of cache	1 M-bytes	512 K-bytes	256 K-bytes	128 K-bytes
Original	25.06	31.25	31.25	31.26
CP	12.68	31.24	31.25	31.26
Combination	0.325	7.731	15.17	15.17

Table 3. Comparisons of *Original*, *CP*, and *Combination*. The size of the array and the size of the two-level cache are fixed, and the number of iterations of the innermost loop is varied

		Miss rates (%)				
Iterations of the innermost loop		65536	32768	16384	8192	4096
Original	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
Combination	C^1	15.19	15.19	15.19	7.686	0.521
	C^2	0.316	0.319	0.326	0.340	0.364

because the proposed technique can tile the inner loops according to the size of each region. For the second level cache, accessed data will not replace the neighboring region since each region of the second level cache is larger than that of the first level cache. The improvement is therefore significant.

The ratio of the locality reference space of reference patterns is varied ranging from 1:1 to 1:4, to compare the effects on cache miss rate of various cache partitioning techniques, (2), (3), and (4). The ratio of the sizes of the two reference spaces is set at 1:1, 1:2, 1:3, 1:4, 2:3, and 3:4. The size of the arrays is set at 8 MBytes. The cache is taken as a two-level cache in which the first and the second level caches have size, 256 KBytes and 1 MBytes, respectively. Table 5 presents the cache miss rate of the simulation.

Applying the technique presented here partitions the cache into equal regions when the ratio of reference spaces is 1:1. Thus, *EP* and *CP* have the same cache miss rate. *Combine* has a lower cache miss rate since loop tiling prevents the referenced elements from replacing the neighboring region. When the sizes of reference spaces are unequal, *CP* and *Combine* show a smaller cache miss rate than *EP*. Notably, the cache miss rate raises when the difference between the sizes of the reference spaces becomes large. This result follows from the fact that a larger stride of memory access yields poor cache performance since many elements move to the cache without a reference and occupy the cache space. In such cases, *Combine* and *CP* exhibit lower cache miss rates than *EP*.

Table 4. Comparisons of *Original*, *CP*, and *Combination*. The size of the array and the number of iterations of the innermost loop are fixed, the size of the two-level cache is varied

		Miss rates (%)			
Size of the first level cache		64 K-bytes	32 K-bytes	16 K-bytes	8 K-bytes
Size of the second level cache		1 M-bytes	512 K-bytes	256 K-bytes	128 K-bytes
Original	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
Combination	C^1	0.521	7.635	15.32	15.45
	C^2	0.364	0.364	0.364	0.368

Table 5. Comparisons of *EP*, *CP*, and *Combination* for two-level cache. The size of the array and the number of iterations of the innermost loop are fixed, the ratio of reference spaces is varied

Ratio of reference space	Miss rates (%)					
	First level cache			Second level cache		
	EP	*CP*	*Combine*	*EP*	*CP*	*Combine*
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

The proposed cache partitioning technique partitions a given cache into several regions according to size of the locality reference space of each dependent array. Applying the padding array scheme aligns each array to be mapped onto one region. The proposed technique can be considered to be a generalization of the partitioning technique proposed in [10]. As discussed in Section 2, the equal-sized partitioning suffers from either poor cache utilization or a high cache miss rate, even though loop tiling is applied. A greedy algorithm is proposed to reduce memory fragmentation. Partitioning criteria are considered for a multi-level cache system, such that the partitioning of the lower level cache is consistent with that of the higher level cache. Another advantage is that, combining the tiling technique allows the block size to be easily determined according to the partitioned region. Localities in inner-nested loops can thus be exploited.

6. Conclusions

This study proposes an algorithm to partition a multi-level cache into regions, and to schedule those regions for arrays such that both temporal and spatial localities can be exploited and cache conflicts can be avoided. Each cache is treated as a system resource and is scheduled into arrays according to the sizes of locality reference spaces. For multi-level caches, the lower level cache should be carefully partitioned so that its region scheduling is consistent with the cache region scheduling of a higher level cache.

A greedy method is developed to reposition both the dependent and independent arrays, reducing the size of the padding array, and also, therefore, the memory overhead caused by applying the padding array technique. Atom is employed as a tool to develop a multi-level cache environment and simulate cache behavior under the direct mapping scheme, and thereby evaluate the performance of the proposed cache partitioning algorithm. An investigation of performance shows that the proposed cache partitioning scheme can greatly reduce the cache miss rate caused by cache conflict, while exploiting the reuse opportunities.

Acknowledgments

The authors would like to thank the National Science Council of the Republic of China for financially supporting this research under Contract No. NSC-89-2213-E-008-013 and NSC-89-2213-E-156-001.

References

1. D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. Technical report UCB/CSD-93-781. Computer Science Division, University of California, Berkeley, 1993.
2. D. F. Bacon, J. H. Chow, D. C. R. Ju, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *CASCON'94*, pp. 270–282. Toronto, Canada, 1994.
3. F. Chen, T. W. O'Neil, and E. Sha. Machine architecture optimizing overall loop schedules using prefetching and partitioning. *IEEE Transactions on Parallel and Distributed Systems*, 11(6):604–614, 2000.
4. K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, Inc. 1984.
5. M. Kandemir, J. Ramanujam, and A. Choudhary. Improving cache locality by a combination of loop and data transformations. *IEEE Transactions on Computers*, 48(2):159–167, 1999.
6. M. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance of blocked algorithms. In *Proceedings of the Fourth International Conference Architectural Support for Programming Languages and Operating Systems*, pp. 63–74, 1991.
7. A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, 1994.
8. J. H. Lee, M. Y. Lee, S. U. Choi, and M. S. Park. Reducing cache conflicts in data cache prefetching. *Computer Architecture News*, 22(4):71–77, 1994.
9. L. S. Liu, C. W. Ho, and J. P. Sheu. On the parallelism of nested for-loops using index shift method. In *Proceedings of the International Conference on Parallel Processing*, vol. II, pp. 119–123, 1990.
10. 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.
11. T. Mowry. Tolerating latency through software-controlled data prefetching. Ph.D. dissertation. Dept. of Electrical Engineering, Standford University, 1994.
12. 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*, 48(2):142–149, 1999.
13. S. Przybylski, M. Horowitz, and J. L. Hennessy. Performance tradeoffs in cache design. In *Proceedings of the 15th Symposium Computer Architecture*, pp. 290–298, 1988.
14. G. Rivera and C. W. Tesig. Data transformations for eliminating conflict misses. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.
15. O. Temam, C. Fricker, and W. Jalby. Impact of cache interferences on usual numerical dense loop nests. *Proceedings of the IEEE*, 81(8):1103–1115, 1993.
16. M. J. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference Parallel Processing for Scientific Computing*, pp. 357–361, 1987.
17. 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, 1991.
18. D. C. Wong, E. W. Davis, and J. O. Young. A software approach to avoiding spatial cache collisions in parallel processor systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(6):601–608, 1998.