Cache-Partitioned Tiling for Data Reuse Across Loop Nests

Naraig Manjikian and Tarek Abdelrahman

Technical Report CSRI-307
October 1994

*** PLEASE NOTE: this report has now been superceded by report CSRI-318 ***

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Cache-Partitioned Tiling for Data Reuse Across Loop Nests

Naraig Manjikian and Tarek S. Abdelrahman
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada

Abstract

This paper presents cache-partitioned tiling, a systematic and integrated approach for global optimization of cache locality across multiple loop nests which reference multiple arrays. The approach is based on the idea of cache partitioning, in which the cache capacity is divided into a number of equal-sized sections. A data layout in memory is derived to eliminate cache conflicts by ensuring that arrays map into different partitions of the cache. A tiling transformation is then used to interleave iterations from different loop nests which reference the same data. Tiling enables reusable data from different arrays to remain cached until no longer needed, eliminating capacity misses that would result from executing individual loop nests to completion. The advantages of cache-partitioned tiling in relation to previous techniques are: a global view towards locality enhancement across loop nests; a systematic data-management approach which is integrated with the code restructuring for eliminating conflicts and retaining reusable data in the cache; and a guaranteed reduction in the number of cache misses to the minimum possible. The techniques for implementing cache-partitioned tiling are presented in sufficient detail to permit automation in a compiler. An analytical model based on data reuse and memory speed estimates performance improvements of 50% or more. Results of cache-partitioned tiling experiments conducted on one processor of the KSR1 multiprocessor system using real applications agree with the estimates obtained with the analytical model.

Index Terms — Data layout, Data reuse, Locality enhancement, Loop transformations, Optimizing compilers, Reduction of cache misses

1 Introduction

Modern computer systems rely on high-speed caches to bridge the increasing speed gap between processors and main memory. Enhancing cache locality is critical for achieving high levels of performance. This is particularly true in scientific applications where large arrays are accessed in nested loops. Considerable research effort has been devoted to exploring techniques that improve cache locality in such applications. Examples include loop blocking [6], loop fusion and loop distribution [2], and array padding and memory alignment [2].

The overall objective of these techniques is to reduce the number of cache misses by retaining reusable data in the cache. There are two primary means of achieving this; reducing the time between data reuses and reducing the possibility of conflicts in the cache. Reducing the time between data reuses decreases the probability of the data being displaced from the cache when the total amount of data being used exceeds the cache capacity. Reducing conflicts prevents data from being displaced unnecessarily from the cache.

Nonetheless, the above techniques suffer from serious limitations. First, these techniques are restricted to a single loop nest in their scope, rendering them useful only in local, rather than global, locality optimizations. Real applications contain multiple loop nests, and optimizing locality for each loop nest individually may not result in the best overall performance. Second, these techniques are typically applied in an ad hoc manner which prevents a systematic assessment of their benefit. As a result, a compiler cannot make an informed decision on whether or not to apply a particular technique.

In light of the above shortcomings, we propose *cache-partitioned tiling* as a systematic approach for *global* optimization of cache locality in programs consisting of multiple loop nests which reference a number of arrays. The central component of our approach is the idea of *cache partitioning*, in which we divide the cache capacity into equal-sized sections. Partitioning is performed entirely in software; no additional hardware is required. We derive a layout for arrays in main memory to eliminate cache conflicts by ensuring that the arrays map into different partitions of the cache. Data reuse across all loop nests in a program is taken into consideration to derive and enforce layout constraints on arrays which must reside in the cache simultaneously. We then employ a tiling transformation to overcome limited cache capacity by interleaving iterations from different loop nests which reference the same data. This transformation enables portions of arrays which are reused across loop nests to remain cached until they are no longer needed. Cache-partitioned tiling is described in sufficient detail to permit its inclusion in a compiler.

Cache-partitioned tiling differs from previous work in three respects. First, it is an *integrated* approach which considers both the computation and the data layout in order to *globally* enhance cache locality across programs. Second, it *systematically* reduces the number of cache misses to the minimum possible. Finally, it is possible to estimate the performance improvement resulting from our approach with analytical modelling.

The remainder of this paper is organized as follows. In Section 2, we present background material and contrast our approach with other similar techniques. In Section 3, we present an overview of cache-partitioned tiling. In Section 4 we provide the details for cache-partitioned data layout, and in Section 5, we provide the details for cache-partitioned tiling. In Section 6, we describe an analytical performance model which gives an estimate of the speedup which can be obtained with cache-partitioned tiling. In Section 7, we present experimental results from applying our proposed techniques to a number of real applications. Finally, in Section 8, we give concluding remarks and future directions for our work.

2 Background

Cache-partitioned tiling improves performance by reducing the number of cache misses incurred in a program. There are three types of cache misses. *Compulsory* misses result from initial references to data by the processor. *Conflict* misses occur after valid data in the cache is replaced with new data which maps to the same cache location. *Capacity* misses result from the displacement of data from the cache when the size of the working data set exceeds the size of the cache. Since the data set sizes in scientific applications often exceed the cache size, reducing the number of capacity misses significantly improves performance; eliminating conflict misses further improves performance.

Techniques that aim to reduce the number of cache misses in a program can be classified into two main classes. Techniques in the first class rely on code restructuring to reorder the execution of loop iterations so as to reduce the time between data reuses. Examples include loop blocking [6] and loop fusion [2]. Techniques in the second class reorganize program data to reduce the potential for conflicts in the cache. Examples include array padding and memory alignment [2]. In the remainder of this section, we contrast cache-partitioned tiling with these two classes of techniques.

Loop blocking [6, 9, 3] is a code restructuring technique that is commonly employed in array-based applications. The iteration space of a loop is divided into a number of blocks, and each block of iterations is completely executed before the next block. This approach attempts to ensure that the data referenced within a block remains cached and is completely reused before being displaced by data referenced in the next block. Blocking has been applied only to individual loop nests in an attempt to exploit reuse of a single array. In contrast, cache-partitioned tiling globally enhances locality for multiple arrays across multiple loop nests, taking into consideration data usage. Furthermore, the performance of blocking is extremely sensitive to the choice of block size [6]. Increasing the block size beyond some threshold for given cache and array sizes leads to a significant loss in performance due to conflict misses. The reduction of the block size below this threshold may result in underutilization of the cache capacity, and hence reduced performance. On the other hand, the performance of cache-partitioned tiling is not sensitive to cache and array sizes; conflict misses are systematically eliminated with a data layout which fully utilizes the cache capacity.

Loop fusion and loop distribution are complementary loop transformations which can be used to enhance locality [2, 5]. Loop fusion combines statements from adjacent loops which reference the same data into a single loop body. In this manner, references to the same data are moved closer in time, thereby increasing the likelihood of achieving locality in the cache, as well as in registers. Conversely, loop distribution moves statements from a single loop body into separate loops. Consequently, less data is referenced in each loop and the possibility of conflicts in the cache is decreased. However, the two loop transformations suffer from the lack of guidelines for determining the extent to which each can improve performance. For example, successively fusing statements from many loops into a single loop can result in one very large loop body. This considerably increases register pressure and can also increase the likelihood of conflicts in the cache. On the other hand, repeated application of loop distribution increases the time between data reuses, which reduces the likelihood of achieving locality. Cache-partitioned tiling *guarantees* a performance improvement by retaining reusable data in the cache until it is no longer needed, and the extent of this improvement can be estimated accurately. Furthermore, loop fusion or distribution may be prevented by data dependences. On the other hand, cache-partitioned tiling is applicable even in the presence of data dependences which prevent fusion or distribution.

Array padding [2] is a data-reorganization technique which increases the number of array elements in the dimension aligned with the storage order to avoid cache conflicts when accessing the array along the other dimensions. The drawback to this approach is that it is applied in an *ad hoc*

```
      array a[n,n], b[n,n]

L1 :  do j = 1, n
        do i = 1, n
          a[i,j] = f(b[i,j])
        enddo
      enddo
L2 :  do j = 1, n
        do i = 1, n
          b[i,j] = f(a[i,j])
        enddo
      enddo
```

(a) Original loop nests

```
      array a[n,n], b[n,n]

      do jj = 1, n, s
L1 :    do j = jj, min(jj + s - 1, n)
          do i = 1, n
            a[i,j] = f(b[i,j])
          enddo
        enddo
L2 :    do j = jj, min(jj + s - 1, n)
          do i = 1, n
            b[i,j] = f(a[i,j])
          enddo
        enddo
      enddo
```

(b) Tiled loop nests

Figure 1: A simple loop nest sequence

manner to individual arrays without regard to overall performance implications across the many loops that reference these arrays. Cache-partitioned tiling is different in that it enforces a data layout that removes conflicts by considering data usage across all loops.

Memory alignment [2] is another data-reorganization technique which aligns data in memory to cache line boundaries to reduce the number of cache lines which must be loaded into the cache. While this may result in a significant decrease in cache traffic for small data sets, the impact for large data sets is not significant due to the fine granularity of cache lines. In contrast, the layout in cache-partitioned tiling aligns data in memory to *cache partition* boundaries. Coupled with the tiling transformation, this alignment reduces the number of cache misses, and hence the data traffic, to the minimum possible.

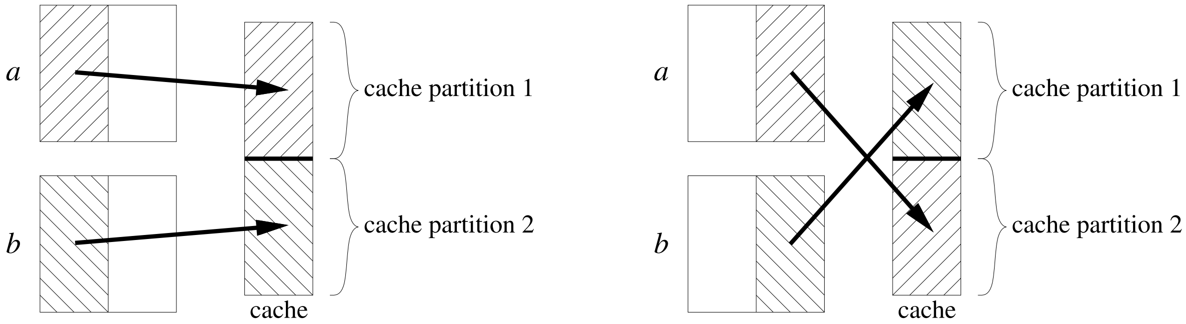
In conclusion, the strengths of our cache-partitioned techniques in relation to previous techniques are: a *global* view towards locality enhancement across loop nests, a *systematic* data-management approach which is *integrated* with the code restructuring in order to eliminate conflicts and retain reusable data in the cache, and a *guaranteed* reduction in the number of cache misses to the minimum possible.

3 Overview

In this section, we introduce key aspects of our proposed locality-enhancing techniques using the hypothetical loop nest sequence shown in Figure 1(a) as an example. We focus on the execution of the two loop nests L_1 and L_2, which both reference the arrays a and b. As L_1 executes, cache misses are incurred in order to load the arrays a and b into the cache. We assume that the size of each array exceeds the cache capacity. Hence, executing L_1 to completion displaces data needed by L_2 from the cache. As a result, additional cache misses are incurred by L_2 in reloading a and b into the cache. Consequently, cache locality is lost, even though L_2 reuses data which is referenced in L_1.

To restore cache locality, iterations from L_1 and L_2 which reference the same data should be executed closer in time. *Tiling across the loop nests* as shown in Figure 1(b) achieves the desired result. Tiling introduces a controlling jj loop shared by L_1 and L_2 and modifies the bounds of the original j loops. In each iteration of the controlling loop, a *tile* of j-loop iterations is executed from each of the loop nests L_1 and L_2. The tiling *interleaves* iterations from L_1 with iterations from L_2 which reuse the same data. To enable portions of arrays a and b to remain cached for reuse between tiles in the same iteration of the controlling loop, the step size s, which we refer to as the *tile size*, must be chosen such that the combined portions of data do not exceed the cache capacity. Tiles in subsequent iterations of the controlling loop may load new data and displace the cached data since it is no longer needed.

Although an appropriate tile size *enables* data to remain cached, it is not sufficient to *guarantee* cacheability. If portions of different arrays map into overlapping regions of the cache, then reusable data is displaced even if the cache capacity is sufficient for the data. Hence, an appropriate layout for the arrays in memory is also required to prevent such conflicts. To retain portions of arrays in cache without conflicts, the cache is logically *partitioned* into equal-sized contiguous sections, and the layout of the arrays in memory is adjusted such that a contiguous portion from each array maps to a separate partition. It is only necessary to adjust the starting address of each array; once the starting addresses have been properly adjusted, portions of data from different arrays will always map to separate partitions in successive iterations of the controlling loop. We refer to this approach as cache-partitioned data layout. Once the size of cache partitions is known, the tile size s is given simply by the number of j-loop iterations which fill the partitions without overflowing. We refer to the entire procedure consisting of cache-partitioned data layout and tiling across loop



(a) Mapping of array data into cache partitions for first iteration of controlling loop

(a) Mapping of array data into cache partitions for second iteration of controlling loop

Figure 2: Mapping of array data with cache-partitioned tiling

nests as cache-partitioned tiling.

The relationship between the data layout and tiling in our cache-partitioned approach is illustrated in Figure 2 for our hypothetical loop nest sequence. The cache is divided into two partitions, the starting addresses for arrays a and b are chosen to map them into separate partitions, and the tile step size s is chosen such that each partition is completely filled when executing tiles. In Figure 2(a), the mappings for data referenced by tiles in the first iteration of the controlling loop are shown, where the first portion of array a maps into cache partition 1 and the first portion of array b maps into cache partition 2. When tiles in the next iteration of the controlling loop are executed, new data is loaded into the cache, displacing the old data. Figure 2(b) illustrates how the next portion of array a now maps into cache partition 2 as a result of the cyclic function which maps addresses into the cache, whereas the next portion of b now maps into cache partition 1.

In order to apply cache-partitioned tiling, we require the loop nests to traverse the array data space in a *compatible* manner. That is, the direction of data access for a given array at a given nesting level must be the same across all loop nests. This definition of compatibility is required to permit loop nests to share a common controlling loop after tiling. Transformations such as loop interchange [2] may be employed to enforce compatibility. The loop bounds at each nesting level are restricted to affine expressions of loop indices from enclosing loops. This is shown in Figure 3 for two-dimensional loop nests and arrays.

array $A_1[N,N], A_2[N,N], \ldots, A_m[N,N]$

$$
\begin{array}{ll}
L_1: & \text{do } j_1 = \ell_1^j, u_1^j \\
 & \quad \text{do } i_1 = \ell_1^i(j_1), u_1^i(j_1) \\
 & \qquad A_1[f_1^1(i_1), g_1^1(j_1)], \ldots, A_m[f_1^m(i_1), g_1^m(j_1)] \\
 & \quad \text{enddo} \\
 & \text{enddo} \\
L_2: & \text{do } j_2 = \ell_2^j, u_2^j \\
 & \quad \text{do } i_2 = \ell_2^i(j_2), u_1^i(j_2) \\
 & \qquad A_1[f_2^1(i_2), g_2^1(j_2)], \ldots, A_m[f_2^m(i_2), g_2^m(j_2)] \\
 & \quad \text{enddo} \\
 & \text{enddo} \\
 & \quad \vdots \\
L_n: & \text{do } j_n = \ell_n^j, u_n^j \\
 & \quad \text{do } i_n = \ell_n^i(j_n), u_n^i(j_n) \\
 & \qquad A_1[f_n^1(i_n), g_n^1(j_n)], \ldots, A_m[f_n^m(i_n), g_n^m(j_n)] \\
 & \quad \text{enddo} \\
 & \text{enddo}
\end{array}
$$

Figure 3: Two-dimensional example of program model for cache-partitioned tiling

4 Cache-Partitioned Data Layout

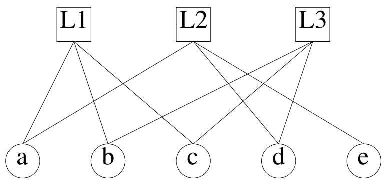
Cache-partitioned data layout is used to ensure a conflict-free mapping of arrays while fully utilizing the cache capacity. There are three main steps involved in cache partitioning for a given program: (1) identification of array use and reuse across the loop nests of the program; (2) generation of cache partitions and assignment of arrays to individual partitions; and (3) derivation and enforcement of array data layout constraints. The following sections describe these steps in detail.

4.1 The Array Reuse Graph

The *array reuse* graph (ARG) is used to determine the use and reuse of arrays in a program. This graph is similar in nature to the interference graph introduced in [1, 4] for communication analysis. It is a bipartite graph $G(L, A, E)$, which consists of n_ℓ vertices $L[G]$ that represent loop nests, n_a vertices $A[G]$ that represent arrays, and a set of undirected edges $E[G]$. An edge (ℓ, a) corresponds to a reference to the array a in the loop nest ℓ. The set of vertices $L[G]$ are subject to a total ordering, denoted by $\prec_{L[G]}$, which corresponds to the program execution order of the loop nest

```
L1: do j=1,n
      do i=1,n
        c[i,j] = b[i,j] + a[i,j]
      enddo
    enddo
L2: do j=1,n
      do i=1,n
        d[i,j] = a[i,j] + e[i,j]
      enddo
    enddo
L3: do j=i,n
      do i=1,n
        b[i,j] = c[i,j] + d[i,j]
      enddo
    enddo
```

(a) A simple program



(b) ARG for program

U(1) = {a,b,c}
U(2) = {a,e,d}
U(3) = {b,c,d}

R(1) = ∅
R(2) = {a}
R(3) = {b,c,d}

(c) Use and reuse sets

Figure 4: An example array reuse graph

sequence. A connected component within the ARG for a program denotes loop nests which reuse a set of arrays, and each connected component may be treated individually. In the remainder of this section, we assume that an ARG contains only one connected component for simplicity. An example showing the ARG for a simple program is given in Figure 4, ant it consists of a single connected component.

We define a number of subsets of arrays obtained from an ARG. The *use* set of a loop nest ℓ, denoted by $U(\ell)$, contains all arrays that are referenced in ℓ. That is,

$$U(\ell) = \{a \in A[G] : (\ell, a) \in E[G]\}.$$

The *reuse* set of a loop nest ℓ, denoted by $R(\ell)$, contains arrays in $U(\ell)$ that also appear in the use sets of *preceding* loop nests. That is,

$$R(\ell) = \bigcup_{\ell_i : \ell_i \prec \ell} (U(\ell_i) \cap U(\ell)).$$

The use and reuse sets for the example of Figure 4 can be easily determined by inspection. The use sets are:

$$U(1) = \{a, b, c\}, U(2) = \{a, e, d\}, U(3) = \{b, c, d\}.$$

The reuse sets are:

$$R(1) = \emptyset, \quad R(2) = U(1) \cap U(2) = \{a\},$$

$$R(3) = (U(1) \cap U(3)) \ \cup \ (U(2) \cap U(3)) = \{b, c\} \cup \{d\} = \{b, c, d\}.$$

4.2 Assignment of Arrays to Cache Partitions

The use and reuse sets derived from the ARG are used to determine the required number of *logical* cache partitions. The set $U_p(\ell)$ is defined to include all preceding uses of arrays as seen from a loop nest ℓ. That is,

$$U_p(\ell) = \bigcup_{\ell_i : \ell_i \prec \ell} U(\ell_i).$$

Similarly, the set $R_f(\ell)$ is defined to include all future uses of arrays as seen from a loop nest ℓ. Hence,

$$R_f(\ell) = \bigcup_{\ell_i : \ell \prec \ell_i} R(\ell_i).$$

The arrays in $U_p(\ell) \cap R_f(\ell)$ must remain in the cache in anticipation of future reuse at the time that loop nest ℓ is executed. The *cache coresidency* set $C(\ell)$ for a loop nest ℓ is defined to include those arrays as well as arrays in its own use set. That is,

$$C(\ell) = U(\ell) \ \cup \ (U_p(\ell) \cap R_f(\ell)). \tag{1}$$

The lifetime $LT(a)$ of an array $a \in A[G]$ is given by the set of loop nests that have a as a member of their coresidency set. Hence,

$$LT(a) = [\ell_b, \ell_e] \ni \begin{cases} a \in C(\ell), & \ell_b \preceq \ell \preceq \ell_e, \\ a \notin C(\ell), & \ell \prec \ell_b, \ell_e \prec \ell. \end{cases} \tag{2}$$

The notation $[\ell_b, \ell_e]$ denotes an interval set which includes all loop nests between ℓ_b and ℓ_e in the total ordering $\prec_{L[G]}$. The magnitude of the lifetime $|LT(a)|$ is the number of loop nests in the interval set. The required number of partitions, n_p, is given by the largest coresidency set:

$$n_p = \max_{\ell \in L[G]} |C(\ell)|. \tag{3}$$

Equivalently, the required number of partitions is given by the largest number of overlapping array lifetimes in any loop nest. That is,

$$n_p = \max_{\ell \in L[G]} |\{a \in A[G] : \ell \in LT(a)\}|.$$

Thus, the number of cache partitions is always less than or equal to the number of arrays; $n_p \leq n_a$.

The coresidency sets for the ARG shown in Figure 4 can be obtained using Equation 1; $C(1) = \{a, b, c\}, C(2) = \{a, b, c, d, e\}, C(3) = \{b, c, d\}$. The lifetimes of the 5 arrays used in the program are obtained using Equation 2, and are shown graphically in Figure 5. It can be seen easily

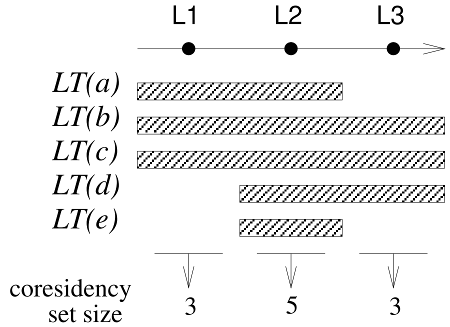
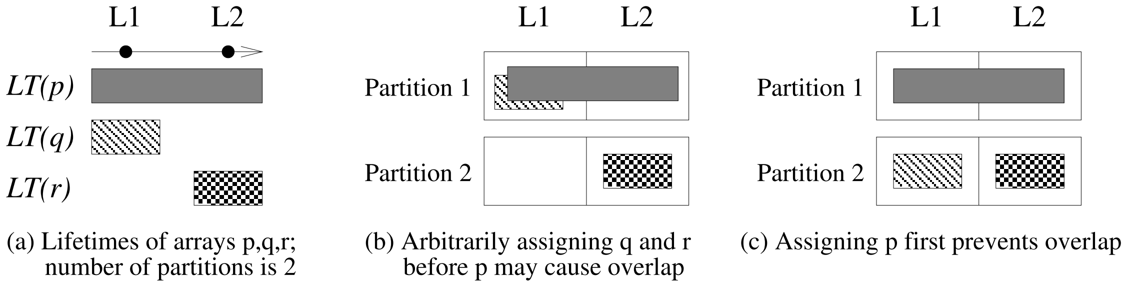


Figure 5: Required number of partitions



(a) Lifetimes of arrays p,q,r; number of partitions is 2
(b) Arbitrarily assigning q and r before p may cause overlap
(c) Assigning p first prevents overlap

Figure 6: Assignment according to longest lifetime

from the figure that the number of partitions is 5. This can also be obtained using Equation 3; $n_p = |C(2)| = 5$.

Once the number of partitions is determined, each array must be assigned to exactly one of the n_p logical partitions. In the trivial case in which the lifetimes of all arrays overlap, i.e., when $n_p = n_a$, each array is assigned a unique partition. In this case, the assignment can be arbitrary, as there is a one-to-one correspondence between arrays and partitions. However, when the number of partitions is less than the number of arrays, i.e., $n_p < n_a$, more than one array must be assigned to the same partition. In this case, arrays assigned to the same partition must not have overlapping lifetimes. The assignment may not be performed arbitrarily because there is no longer a one-to-one correspondence between arrays and partitions. This is illustrated in Figure 6, in which three

```
ASSIGNPARTITIONS::
    P = {0, 1, ..., n_p − 1}
    foreach p ∈ P do
        OCC(p) = ∅
    endfor
    S = A[G]
    do
        select a ∈ S where |LT(a)| = max_{s∈S} |LT(s)|
        done = false
        foreach p ∈ P while not done do
            if OCC(p) ∩ LT(a) = ∅ then
                OCC(p) = OCC(p) ∪ LT(a)
                ASSIGN(a) = p
                S = S \ {a}
                done = true
            endif
        endfor
    while S ≠ ∅
```

Figure 7: Partition assignment algorithm

arrays must be assigned to two partitions. If arrays q and r are assigned partitions arbitrarily as in Figure 6(b), it is not possible to assign p a partition without overlapping. However, if p is assigned first as in Figure 6(c), the result is a legal assignment. Array p has the longest lifetime; selecting it first ensures that it is assigned to a partition which can accommodate it.

An algorithm for assigning arrays to logical partitions is shown in Figure 7. In the outermost loop, arrays with the longest lifetimes are selected first to guarantee a legal assignment. Overlapping lifetimes are prevented by maintaining an occupancy set $OCC(p)$ for each partition. The innermost loop selects the first partition whose occupancy set does not intersect with the array lifetime. The array selected in the outermost loop is then assigned to this partition, and the occupancy set of the partition is updated with the array lifetime. The output of the algorithm on termination is a mapping $ASSIGN(a)$ which assigns each array $a \in A[G]$ to a logical partition in the set $\{0, 1, \ldots, n_p - 1\}$. The complexity of the algorithm is $O(n_p \cdot n_a)$.

4.3 Data Layout Constraints

Once the arrays have been assigned to logical cache partitions, the logical partitions must be mapped to *physical* cache partitions, and the arrays must be placed in *main memory* such that each array maps only into its physical partition of the cache. Since the memory-to-cache mapping is determined by the hardware and is based on a bit field in the memory address, the starting address of each array *in main memory* must be adjusted by inserting appropriately-sized gaps between arrays. The introduction of such gaps implies placing restrictions on the data layout of the arrays in memory, and we will refer to these restrictions as the data layout *constraints.* These constraints will be derived from the cache size, the number of partitions, and the partition assignment. The compiler or run-time system will enforce these constraints by determining the exact sizes of gaps, taking into consideration the array data sizes. This requires that memory allocated to a program be contiguous.

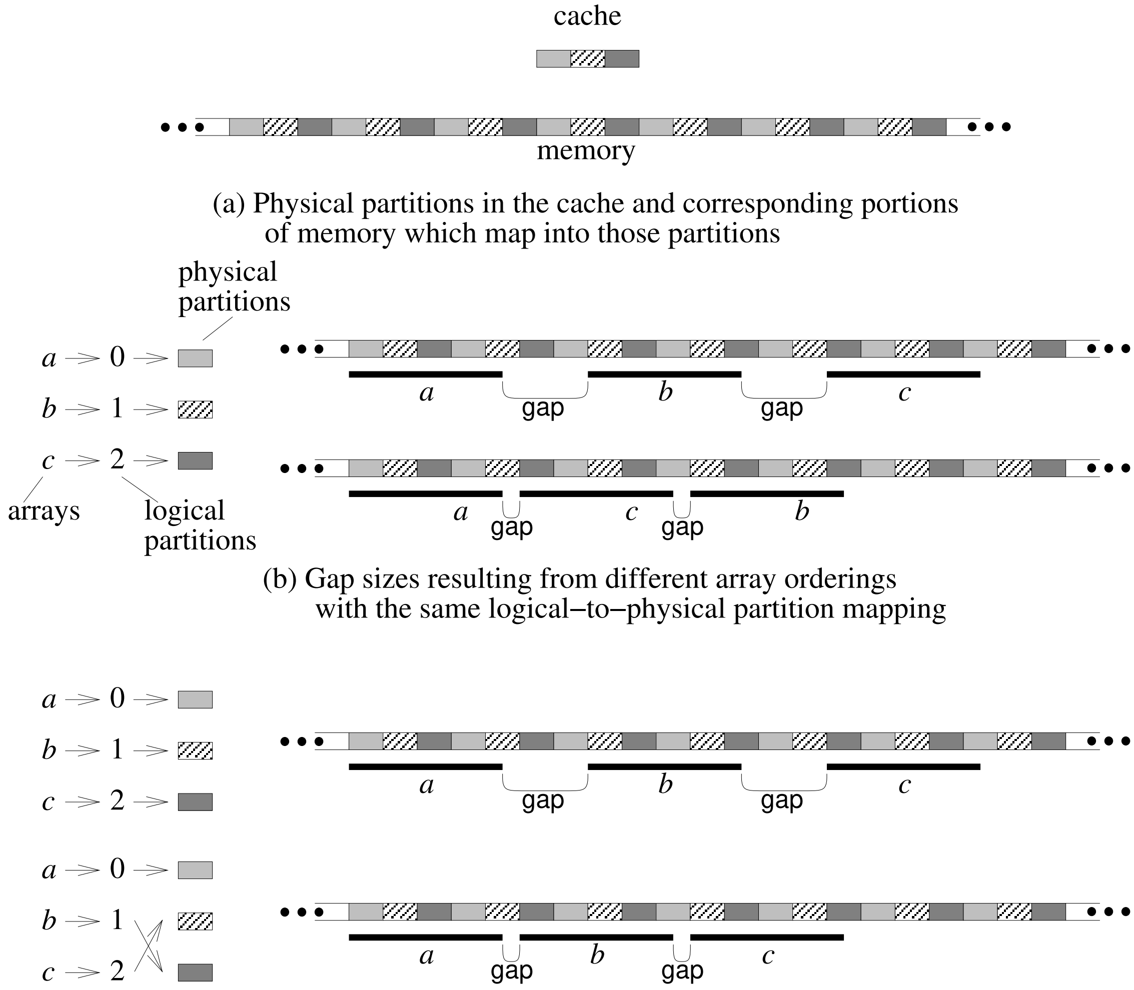
The logical-to-physical partition mapping can be formulated as follows. The physical partitions in the cache are numbered $0, 1, \ldots, n_p - 1$. The size s_p of each partition can be easily computed from the cache capacity c (in words) and the number of partitions n_p as

$$s_p = \left\lfloor \frac{c}{n_p} \right\rfloor .$$

Therefore, the starting addresses *in the cache* for each physical partition is given by $p \cdot s_p$, where $p \in \{0, 1, \ldots, n_p - 1\}$. The logical partitions are also numbered $0, 1, \ldots, n_p - 1$. The logical-to-physical partition mapping is a one-to-one mapping $P \mapsto P$, where $P = \{0, 1, \ldots, n_p - 1\}$. Consequently, this mapping establishes the cache starting address for each logical partition.

The gaps introduced between arrays to enforce data layout constraints constitute overhead when compared to contiguous memory allocation. Hence, it is desirable to minimize the total size of these gaps. The gap sizes are determined by the logical-to-physical partition mapping and by the order in which arrays are placed in memory. This is shown in Figure 8. For the given partition mapping in Figure 8(b), different orders for placing the arrays a, b, and c in memory yields different gap sizes. Similarly, for the given array ordering in Figure 8(c), different partition mappings yield different gap sizes.

Hence, to minimize the overhead, it is necessary to select a logical-to-physical partition mapping and an ordering of arrays in memory which minimizes the total size of the gaps. There are $n_p!$ possible logical-to-physical partition mappings, and $n_a!$ ways to order the arrays, which gives a



(a) Physical partitions in the cache and corresponding portions of memory which map into those partitions

(b) Gap sizes resulting from different array orderings with the same logical–to–physical partition mapping

(c) Gap sizes resulting from different logical–to–physical partition mappings with the same array ordering

Figure 8: Effect of array ordering and partition mapping on gap sizes

total of $(n_p!) \cdot (n_a!)$ alternatives to consider. This suggests that the problem of minimizing gap sizes is NP-hard. Consequently, we use a suboptimal polynomial-time algorithm.

We propose the following *greedy* approach to define the logical-to-physical partition mapping. Arrays are selected in an arbitrary order for placement in memory. As each array is selected, gap sizes for all available physical partitions are computed. The physical partition which yields the smallest gap size is chosen, and the process is repeated for the remaining arrays. An algorithm

```
GREEDYENFORCECONSTRAINTS::
    P = {0, 1, ..., n_p − 1}                          (available physical partition indices)
    q = q_0                                           (starting address of available storage)
    S = A[G]                                          (set of arrays)
    foreach p ∈ P
        physical_partition(p) = −1                    (initial mapping is undefined)
    endfor
    do
        select a ∈ S                                  (selection is arbitrary)
        ℓ = ASSIGN(a)                                 (get logical partition for array)
        if physical_partition(ℓ) = −1 then            (physical partition not yet assigned)
            mapped_cache_address = CACHEMAP(q)
            foreach p ∈ P                             (determine gaps)
                target_cache_address(p) = p · s_p
                gap(p) = target_cache_address(p) − mapped_cache_address
                if target_cache_address(p) < mapped_cache_address then
                    gap(p) = gap(p) + cache_size
                endif
            endfor
            select p_opt ∈ P where gap(p_opt) = min_{p∈P} gap(p)   (select minimum gap)
            gap = gap(p_opt)
            P = P \ {p_opt}                           (remove from available indices)
            physical_partition(ℓ) = p_opt             (establish mapping)
        else
            p = physical_partition(ℓ)                 (physical partition has been assigned)
            target_cache_address = p · s_p
            mapped_cache_address = CACHEMAP(q)
            gap = target_cache_address − mapped_cache_address
            if target_cache_address < mapped_cache_address then
                gap = gap + cache_size
            endif
        endif
        START(a) = q + gap                            (insert gap)
        q = START(a) + SIZE(a)                        (adjust start for next array)
        S = S \ {a}
    while S ≠ ∅
```

Figure 9: Greedy data layout constraint enforcement algorithm

implementing this approach is given in Figure 9. The algorithm begins by initializing the logical-to-physical partition mapping to an undefined state. A set P of unassigned physical partitions is also initialized. Beginning at an address q_0, arrays are selected in an arbitrary order for placement in memory. For each array a, the logical partition ℓ to which the array has been assigned is given by the mapping *ASSIGN*(a). If the logical partition ℓ has not yet been assigned a physical partition, the gap sizes for all available physical partitions in P are computed. The size of the gap for each physical partition p for the array a is the distance between the *target_cache_address* for p, given by $p \cdot s_p$, and the *mapped_cache_address* obtained with the hardware mapping function CacheMap(q), where q is the initial memory starting address for the array a. When the *mapped_cache_address* exceeds the *target_cache_address*, the gap must be increased by the cache size c in order to reach the next closest address *in memory* which maps to the same location in the cache. Once the gap sizes are computed for all available partitions, the partition p_{opt} yielding the minimum gap size is selected. This physical partition is assigned to the logical partition ℓ, and it is removed from the set P. The starting address of a in memory is then adjusted by the minimum gap size, and the new starting address for the next array is determined using the size of a. The process is then repeated by selecting another array.

When $n_p = n_a$, each logical partition (and hence each array) is assigned a unique physical partition. When $n_p < n_a$, there must exist at least two arrays a_1, a_2 for which *ASSIGN*(a_1) =*ASSIGN*(a_2). Consequently, a_1 and a_2 must use the same physical partition. If a_1 is placed in memory before a_2, then a_2 must use the same physical partition as a_1. Hence, the algorithm checks whether or not the logical partition of an array a has been assigned a physical partition. If it has been assigned a partition, then the algorithm does not select among available physical partitions, but rather computes the gap size using the assigned physical partition.

The complexity of this algorithm is $O(n_a \cdot n_p)$; n_a arrays are each selected arbitrarily in constant time, and in the worst case, a choice must be made among n_p physical partitions for each array. On termination, the output of the algorithm is a mapping *START*(a) specifying the starting address in memory for each array a.

To characterize the memory overhead resulting from the introduction of the gaps between arrays, we assume that arrays have the same size d. We consider first an upper bound. The maximum gap size for an array is $c - 1$, which occurs when *mapped_cache_address* = *target_cache_address*+1.

If a gap of this size is introduced for each array, the maximum overhead is given by

$$o_m^{max} = \frac{n_a \cdot (c-1)}{n_a \cdot d} < \frac{c}{d}. \tag{4}$$

However, it is unlikely that the actual overhead will be close to the upper bound since the greedy algorithm selects physical partitions to minimize gap size. The actual overhead is difficult to predict in the general case when arrays must share physical partitions (i.e., when $n_p < n_a$) because the resulting gap sizes depend on the order in which arrays are placed in memory, and this order is arbitrary in our algorithm.

However, it is possible to show that the algorithm in Figure 9 yields the minimum possible overhead *regardless of the array selection order* when $n_p = n_a$ and when all arrays have the same size d. In this case, the optimal overhead can be quantified exactly. The size of each partition is $s_p = \lfloor c/n_p \rfloor = \lfloor c/n_a \rfloor$. A required parameter for computing the overhead is the distance *in the cache* between the start and end of an array of size d. If the start of an array maps to address 0 in the cache, then the end of an array maps to the cache address $e_c = d \bmod c$. Translated into units of cache partitions, this distance is $m = e_c/s_p$ partitions. With contiguous data layout, the *next* array would map to address e_c in the cache. However, the algorithm in Figure 9 adjusts the starting address of the next array in memory such that it maps into the closest available physical partition to e_c in the cache. The closest physical partition is given by rounding up m to an integral number $\hat{m} = \lceil m \rceil$. The gap size is then $(\hat{m} - m) \cdot s_p$, *which is the minimum possible in this case.* There are $n_a - 1$ gaps of this size *between* arrays. In addition, when selecting partitions using the greedy approach, the arrays will cluster into a number of groups determined by $n_g = \gcd(n_p, \hat{m})$. Within each group, the closest physical partition can always be chosen for up to n_p/n_g arrays. After that point, the mapping cycles back to the first physical partition in the group, which has already been assigned. To break out of the cycle and assign the remaining arrays, an extra gap consisting of one whole partition s_p must be added. This extra gap is added $n_g - 1$ times because there are n_g groups. The memory overhead for the gaps between arrays can now be expressed as

$$o_m = \frac{(n_a - 1) \cdot (\hat{m} - m) \cdot s_p + (n_g - 1) \cdot s_p}{n_a \cdot d}. \tag{5}$$

The first term accounts for the gaps to the closest physical partition within each group, and the second term accounts for the extra gap of one whole partition between groups. The gaps *between arrays* are the smallest possible, and the additional gaps *between groups* are unavoidable, hence the algorithm yields the minimum possible overhead in this case.

Table 1: Memory overhead from cache partitioning for 8 arrays

Cache size (Mbytes)	Array size N (data size $= N \cdot N \cdot 8$ bytes)					
	600		800		1000	
	Exact	Worst	Exact	Worst	Exact	Worst
0.25	1.0%	9.1%	0.6%	5.1%	0.4%	3.3%
0.5	2.0%	18.2%	1.1%	10.2%	0.7%	6.6%
1	4.0%	36.4%	2.2%	20.5%	1.4%	13.1%
2	8.0%	72.8%	4.5%	41.0%	2.9%	26.2%

The extent of memory overhead determined by Equation 5 is shown in Table 1 using 8 arrays for a range of cache and data sizes. Also shown is the worst-case overhead determined by Equation 4. The table indicates that the exact overhead is normally much less than the worst-case overhead, and that the worst case overhead decreases for larger array sizes. In typical scientific applications, in which arrays are of equal size and where the array size is much larger than the cache size, the overhead will be acceptably low.

5 Cache-Partitioned Tiling

The cache-partitioned data layout presented in the previous section eliminates cache conflicts and enables portions of each array to remain cached. Cache-partitioned tiling *across* loop nests is then used to ensure that the cached portion of each array is indeed reused across the loop nests. This is done by tiling the iteration space of each loop nest and reordering tile execution. Tiles from different loop nests which reference the same array data are brought closer together in time to ensure that the required data is not ejected from the cache. The tiling transformation is illustrated using the the simple program shown in Figure 10(a). The outermost j loop in each loop nest is individually tiled by introducing a controlling jj loop to step across tiles. This is shown in Figure 10(b). Tile execution is reordered by combining the loop nests together to share a single controlling jj loop. This is shown in Figure 10(c). Cache-partitioned data layout has assigned arrays a, b, c, and d to nonconflicting cache partitions. The tile step size s is chosen such that each array does not exceed its partition size. As a result, portions of arrays a and c loaded in the first loop nest remain cached for reuse in the second loop nest. In the remainder of this section, the tiling transformation and the selection of tile size are presented in detail.

```
L1 : do j = 1, n
       do i = 1, m
         c[i, j] = b[i, j] + a[i, j]
       enddo
     enddo
L2 : do j = 1, n
       do i = 1, m
         d[i, j] = c[i, j] + a[i, j]
       enddo
     enddo
```

(a) Original loop nests

```
     do jj = 1, n, s
L1 :   do j = jj, min(jj + s − 1, n)
         do i = 1, m
           c[i, j] = b[i, j] + a[i, j]
         enddo
       enddo
     enddo
     do jj = 1, n, s
L2 :   do j = jj, min(jj + s − 1, n)
         do i = 1, m
           d[i, j] = c[i, j] + a[i, j]
         enddo
       enddo
     enddo
```

(b) Tiling each loop nest

```
     do jj = 1, n, s
L1 :   do j = jj, min(jj + s − 1, n)
         do i = 1, m
           c[i, j] = b[i, j] + a[i, j]
         enddo
       enddo
L2 :   do j = jj, min(jj + s − 1, n)
         do i = 1, m
           d[i, j] = c[i, j] + a[i, j]
         enddo
       enddo
     enddo
```

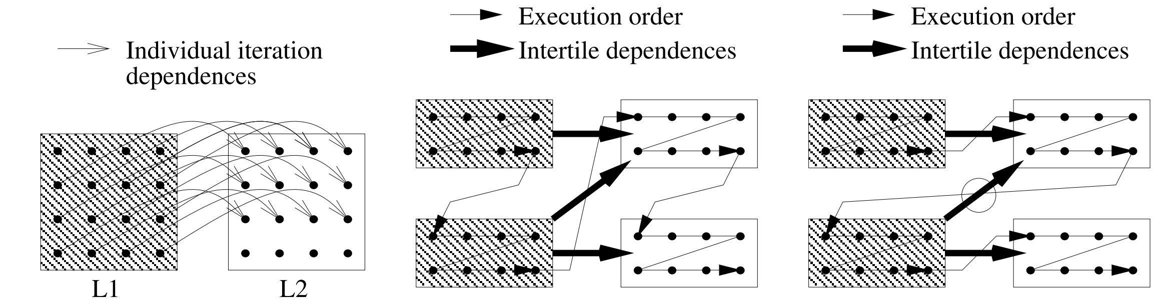
(c) Reordering tile execution

Figure 10: Tiling transformation

5.1 Data Dependences in Cache-Partitioned Tiling

In this section, we discuss the application of the tiling transformation in the presence of data dependences between loop nests such that the original program semantics are preserved. The program model discussed in Section 3 assumes affine subscript expressions which give rise to *uniform* dependences between iterations from different loop nests. To ensure legal tiling, we must only identify the dependences which may be violated and make any necessary adjustments for correctness.

In order to illustrate how dependences may be violated through tiling, assume a program consisting of two loop nests L_1 and L_2, where L_1 executes before L_2. The iteration spaces of the individual loop nests are shown in Figure 11(a). L_1 and L_2 reference the same data, giving rise to dependences between *individual* iterations in L_1 and *individual* iterations in L_2, as shown in Figure 11(a). The dependences *must* flow from L_1 to L_2 since L_1 is executed before L_2. Individual tiling of the outermost j loop in each loop nest is always legal, because it does not change the iteration order *within* each loop nest [2]. The dependences between individual iterations of L_1 and L_2 can be summarized after tiling by *intertile* dependences, as shown in Figure 11(b). An intertile dependence between a tile from L_1 and a tile from L_2 exists if an individual iteration in the L_2 tile depends on an individual iteration in the L_1 tile. Since all tiles in L_1 execute before any tiles in L_2, no intertile dependences are violated. However, when tiling is applied *across* the two



(a) Dependences between iteration spaces of two loop nests

(b) Intertile dependences after individually tiling loop nests

(c) Intertile dependences after tiling across loop nests

Figure 11: Intertile dependences

loop nests, they share a common controlling jj loop and the execution order of the tiles changes. The resulting order is shown in Figure 11(c); for each iteration of the outermost controlling loop, one tile from each of L_1 and L_2 is executed. The intertile dependences may now be classified as loop-independent or loop-carried *with respect to the outermost controlling loop.* Loop-independent intertile dependences flow between tiles from L_1 and L_2 which are executed in the *same* iteration of the controlling jj loop; these dependences cannot be violated because the L_1 tile is always executed before the dependent L_2 tile in the resulting tile order. For example, there are two loop-independent intertile dependences in Figure 11(c). Loop-carried intertile dependences flow between tiles from L_1 and L_2 which are executed in *different* iterations of the outermost controlling loop. The direction of a loop-carried dependence is positive if the dependent L_2 tile is executed in a later iteration of the controlling loop than the L_1 tile. Similarly, the direction is negative if the dependent L_2 tile is executed in an earlier iteration of the controlling loop than the L_1 tile. The new execution order of the tiles may violate loop-carried intertile dependences. For example, the intertile dependence marked with a circle in Figure 11(c) is violated because the dependence requires that the L_1 tile be executed before the L_2 tile, but in the new execution order, the tiles are executed in the opposite order. *A loop-carried intertile dependence is violated by tiling if and only if it has a negative direction* which is the case for the loop-carried dependence in Figure 11(c). Loop-carried dependences with a positive direction are not violated by tiling. Hence, in order for tiling to be legal, all intertile dependences must be either loop-independent or loop-carried with a

$L_1:$	do $j = 1, n$
	  do $i = 1, m$
	    $a[i, j] = \ldots$
	  enddo
	enddo
		$L_1:$	do $j = 1, n$
			  do $i = 1, m$
			    $\ldots = a[i, j - D]$
			  enddo
			enddo

$L_2:$	do $j = 1, n$
	  do $i = 1, m$
	    $\ldots = a[i, j + D]$
	  enddo
	enddo
		$L_2:$	do $j = 1, n$
			  do $i = 1, m$
			    $a[i, j] = \ldots$
			  enddo
			enddo

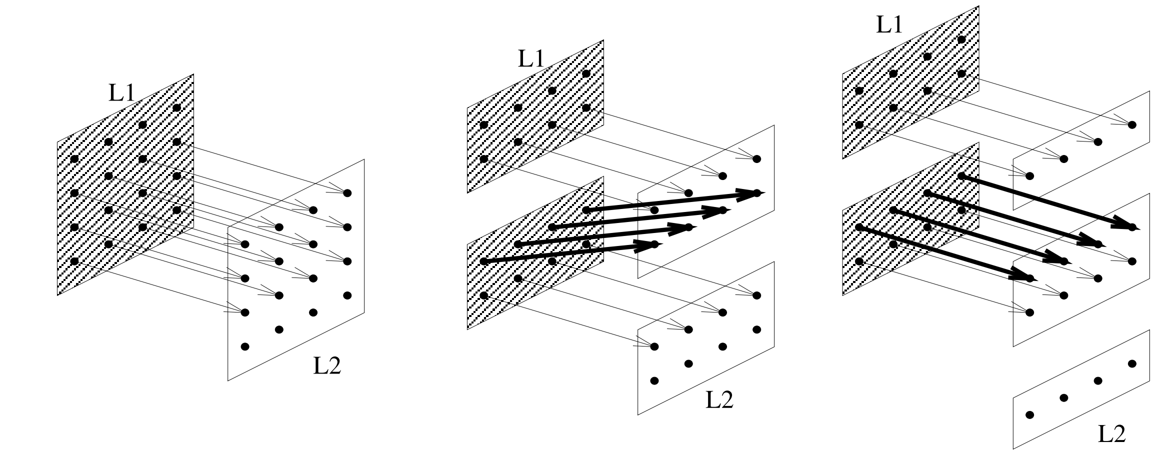
	(a) Leading flow dependence		(b) Lagging antidependence

Figure 12: Examples for negative loop-carried intertile dependences

positive direction.

There are two types of dependences between individual iterations in two loop nests L_1 and L_2 which can lead to negative loop-carried intertile dependences. We denote the first type as a *leading* flow dependence where data values produced in an iteration j of the outermost loop in L_1 are consumed in iteration $j - D$ of the outermost loop in L_2. This dependence has a distance D, where $D > 0$, and is illustrated by the example in Figure 12(a). We denote the second type of dependences as a *lagging* antidependence where data values must be consumed in an iteration j of the outermost loop in L_1 before they are overwritten in iteration $j - D$ of the outermost loop in L_2. This dependence also has a distance D and is illustrated by the example in Figure 12(b). In both cases, dependences flow from iteration j in L_1 to iteration $j - D$ in L_2. When L_1 and L_2 share a common controlling jj loop after tiling, dependences flowing from an iteration j in L_1 to an iteration $j - D$ in L_2 become loop-carried intertile dependences when iterations j and $j - D$ correspond to *different* iterations of the controlling jj loop. This case is exemplified by the circled dependence in Figure 11(c). The direction of this dependence is negative because iteration $j - D$ corresponds to an earlier iteration of jj than j.

It is possible to adjust the tiling such that negative loop-carried intertile dependences are made positive or loop-independent. To illustrate the procedure, the iteration spaces of the two loop nests L_1 and L_2 are shown in Figure 13(a). The individual dependences shown in bold in Figure 13(b) are loop-carried intertile dependences with a negative direction after tiling. To make tiling legal, the tiles in the iteration space of L_2 are *shifted* with respect to those in L_1, as shown



(a) Individual iteration dependences between iteration spaces of two loop nests

(b) Iteration dependences after tiling; dependences carried by controlling loop are in bold

(c) Shifting tiles of L2 prevent dependences from being carried by controlling loop

Figure 13: Resolving intertile dependence violations

in Figure 13(c). The shift changes the order of iterations from L_2 tiles with respect to L_1 tiles, making the negative intertile dependences loop-independent. The tiles are shifted in the iteration space by an amount equal to the dependence distance D which generated the negative loop-carried intertile dependences. The loop bounds for the tiled j loop in L_2 are modified such that the L_2 tiles are offset from L_1 tiles by D iterations. The step size for L_2 tiles remains the same as for L_1 tiles. There will remain D uncompleted iterations of the original j loop in L_2, and a new loop nest must be introduced to execute these remaining iterations.

An application of tile shifting is illustrated in Figure 14. The original program in Figure 14(a) contains two antidependences involving array b, one of which is a lagging antidependence with a dependence distance of 1. The result of tiling across these loop nests is shown in Figure 14(b). A tile shift of 1 iteration is applied to L_2 to transform the negative loop-carried intertile dependence resulting from the lagging antidependence into a loop-independent intertile dependence. The other antidependence initially yields a loop-independent intertile dependence which is transformed after the tile shift into a positive loop-carried intertile dependence. The j loop bounds in L_2 are adjusted by the dependence distance of 1, and a new loop L_2' performs the one remaining j-loop iteration for L_2 after the jj loop has completed.

```
L1 :  do j = 1, n
        do i = 1, m
          a[i, j] = b[i, j − 1] + b[i, j]
        enddo
      enddo
L2 :  do j = 1, n
        do i = 1, m
          b[i, j] = c[i, j]
        enddo
      enddo
```

(a) Untiled loop nests

```
      do jj = 1, n, s
L1 :    do j = jj, min(jj + s − 1, n)
          do i = 1, m
            a[i, j] = b[i, j − 1] + b[i, j]
          enddo
        enddo
L2 :    do j = max(jj − 1, 1), min(jj + s − 2, n − 1)
          do i = 1, m
            b[i, j] = c[i, j]
          enddo
        enddo
      enddo
L2' : do i = 1, m
        b[i, n] = c[i, n]
      enddo
```

(b) Tiled loop nests

Figure 14: Example of tiling with a lagging antidependence

In summary, a tile shift ensures that tiling across loop nests is legal by transforming negative loop-carried intertile dependences into loop-independent ones. Loop-carried dependences of this type are caused by leading flow dependences or lagging antidependences. A tile shift does not affect the legality with respect to other dependences; loop-independent intertile dependences are potentially transformed into positive dependences, and positive dependences remain positive. Implementation of a tile shift only requires an adjustment of loop bounds using the distance from the original leading flow dependence or lagging antidependence. In the following section, we present a general algorithm for deriving tile shifts.

5.1.1 Derivation of Tile Shifts

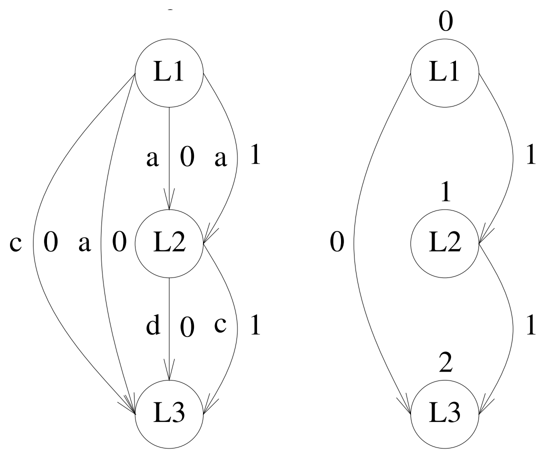
The preceding section has discussed a tile shift for dependences in two loop nests. In general, a sequence of more than two loop nests may contain dependence chains which require several tile shifts. To derive tile shifts in this general case, *all* dependences between loop nests must be extracted. Since our program model assumes affine subscript expressions, the dependence distances are known. We summarize dependences in a *tile shift multigraph* $TSM(L, E)$, consisting of a set of vertices $L[TSM]$ to represent loop nests and a set of weighted, directed edges $E[TSM]$ to represent the dependences. Each directed edge $e = (\ell_s, \ell_d) \in E[TSM]$ indicates a dependence from a source loop nest ℓ_s to a destination loop nest ℓ_d for some array reference. Each edge e also has an associated *weight*(e). For leading flow dependences and lagging antidependences, which give rise to negative loop-carried intertile dependences after tiling, *weight*$(e) = D$, where D is the dependence distance. The weight represents the required tile shift for legality. All other dependences are assigned *weight*$(e) = 0$, since they result in loop-independent or positive intertile dependences after tiling and do not require tile shifts. A *multi*graph is needed to address the possibility of multiple dependences between two loop nests. The multigraph is always acyclic, as the order of the loop nests in the original sequence before tiling ensures that all dependences flow in the same direction.

The multigraph *TSM*(L, E) contains redundant information. We reduce the multigraph to a simpler tile shift graph *TSG*(L, E) by reducing multiple directed edges between two vertices into a single directed edge. The weight of the new edge is the maximum weight of the multiple edges between the two vertices. The maximum weight corresponds to the largest dependence distance between the two loop nests. A tile shift of least this maximum distance is needed to ensure that the negative loop-carried dependences with this distance are made loop-independent. The remaining dependences with distances less than the maximum will become loop-carried with a positive direction.

The graph *TSG*(L, E) now contains all the information necessary to derive the tile shifts necessary to make tiling legal. We associate a weight with each vertex and initialize it to zero. A depth-first traversal of the graph is then performed to sum up edge weights (which represent required tile shifts) along each directed path to each loop vertex, updating the vertex weight to record the maximum path length to that vertex. The final vertex weights are tile shifts for each loop nest *relative to the first loop nest in the sequence*, which are then used to adjust the loop bounds and complete the tiling transformation.

An example loop nest sequence with a chain of dependences is shown in Figure 15(a). The tile shift multigraph extracted from the loop nest sequence is given in Figure 15(b), with the edges labelled by the array causing the dependence and the required tile shift. The reduced tile shift graph and the results of the depth-first traversal to derive the final vertex weights are shown in Figure 15(c). L_2 tiles must be shifted by 1 iteration with respect to L_1, and L_3 tiles must be shifted by 2 iterations with respect to L_1. The shift for L_3 is in spite of the zero-weight direct path from L_1 to L_3; the L_3 vertex accumulates a weight of 2 from the indirect path $L_1 \rightarrow L_2 \rightarrow L_3$.

```
L1 :  do j = 1, n
        do i = 1, m
          a[i, j] = b[i, j] + c[i, j]
        enddo
      enddo
L2 :  do j = 1, n
        do i = 1, m
          d[i, j] = c[i, j − 1] + a[i, j + 1] + a[i, j]
        enddo
      enddo
L3 :  do j = 1, n
        do i = 1, m
          c[i, j] = a[i, j] + d[i, j] + d[i, j + 1]
        enddo
      enddo
```



(a) Loop nest sequence (b) Tile shift multigraph (c) Tile shift graph

Figure 15: Example to illustrate tile shift derivation

5.2 Determining Tile Size

The tile size s is the step size of the controlling jj loop in Figure 10, i.e., the number of j loop iterations per tile. It must be chosen such that the portion of each array used within a tile does not exceed the allocated partition size. Assume that arrays are stored in column-major format. An array with dimensions $R \times C$ has R elements stored contiguously in memory for each column. In the simplest case, tiles in one iteration of the controlling jj loop do not reference data from tiles in other iterations of the controlling loop, i.e., there are no loop-carried dependences with respect to the jj loop. As a result, cache partitions may be fully utilized by completely overwriting data loaded in tiles from preceding iterations of the controlling loop. The largest step size for full utilization of the cache partitions is given by

$$s_{max} = \left\lfloor \frac{s_p}{R} \right\rfloor ,$$

where s_p is the partition size as computed in Section 4, and R is the number of elements per column. It should be stressed that while *any* positive step size is legal, choosing a step size larger than s_{max} means that the array data used within a tile will exceed the cache partition size, causing unnecessary cache misses due to conflicts with data in neighboring cache partitions.

In the presence of loop-carried intertile dependences, tiles in different iterations of the controlling

jj loop reference the same data. In this case, a portion of data loaded by tiles in a preceding iteration of the controlling loop must remain cached. This portion is the data which is referenced at tile boundaries. In order to prevent the required data from being overwritten in the next iteration of the controlling loop, the step size must be reduced by an appropriate amount such that *all* data from one array which is referenced within a tile fits into a single cache partition. The amount by which the step size should be reduced is given by the largest tile shift, i.e., the largest vertex weight in the tile shift graph $TSG(L, E)$ after the depth-first traversal. Let w_{max} denote this maximum weight; that is

$$w_{max} = \max_{v \in L[TSG]} weight\ (v).$$

The reduced step size is then given by

$$s = s_{max} - w_{max}.$$

If $w_{max} \geq s_{max}$, then $s \leq 0$, even though the step size must be positive for the controlling jj loop to execute properly. The interpretation of this result is that the amount of data needed within a tile exceeds that which will fit into a single cache partition. In other words, even if the smallest positive step size s is chosen ($s = 1$), it is not possible to load new data for a tile in the *next* jj iteration without displacing data which is still needed from the *previous* jj iteration. Displacing this data will lead to additional cache misses which can offset any performance improvement from tiling.

When tiling a sequence of loop nests $L_1, \ldots, L_n$, the simplest cause of the condition $w_{max} \geq s_{max}$ is the existence of a single loop-carried dependence with a dependence distance $D \geq s_{max}$ between two loop nests L_a and L_b, where $1 \leq a < b \leq n$. We assume that L_a does not require a tile shift relative to L_1. However, a tile shift of $D \geq s_{max}$ is required for L_b relative to L_a. Hence, this shift for L_b is sufficient to cause the condition $w_{max} \geq s_{max}$. Any of the remaining loop nests $L_{b+1}, \ldots, L_n$ which reuse data from L_b must also be shifted by at least D, and dependences between pairs of loop nests in $L_{b+1}, \ldots, L_n$ may also require additional shifts. The large tile shifts in the latter set of loop nests beginning with L_b displace any reusable data loaded by the preceding loop nests. Consequently, the benefit of tiling the *entire* sequence $L_1, \ldots, L_n$ is lost. In this case, it is preferable to tile the subsequence $L_1, \ldots, L_{b-1}$ separately from the subsequence $L_b, \ldots, L_n$. In this manner, the tile shift greater than s_{max} for L_b is avoided. We denote the dependence between loop nests L_a and L_b as a *threshold* dependence, as crossing this dependence introduces a large tile shift which logically divides the loop nest sequence into two subsequences.

More generally, the condition $w_{max} \geq s_{max}$ may arise from an accumulation of individual tile shifts, each being less than s_{max}, in a chain of dependent loop nests. Since there is no clear threshold dependence in this case, it is necessary to traverse the dependence chain until the cumulative tile shift exceeds s_{max} in order to locate a suitable threshold point to divide the loop nest sequence for separate tiling.

6 Analytical Performance Estimation

In this section, we analytically characterize the benefit that can be obtained from cache-partitioned tiling. Tiling improves performance by reducing the number of cache misses, and hence the data traffic between the cache and main memory. The extent to which this reduction improves performance depends on the fraction of execution time spent servicing cache misses. We quantify the amount of data traffic for both tiled and untiled execution and the fraction of execution time spent servicing cache misses, and derive an estimate for the speedup from tiling.

The execution of a sequence of loop nests *sweeps* across the data referenced in those loop nests, and this data must flow through the cache. Sweeping across large data sets results in cache misses which require data to be reloaded into the cache. In caches which use a write-back policy, data which has been modified in the cache must also be written back to memory before being replaced by new data. Hence, there is data traffic both in and out of the cache. We use the number of sweeps to measure the extent of data traffic, which implies that arrays must have the same size.

In untiled execution, each loop nest is executed completely before the next. Consequently, a complete sweep is made through each array referenced in each loop nest in order to bring the data into the cache. Hence, the number of array sweeps for a loop nest ℓ is $|U(\ell)|$ (see Section 4). Across all loop nests, the total number of sweeps for loading data *into* the cache is

$$s_u = \sum_{\ell \in L[G]} |U(\ell)|.$$

Writebacks due to replacement cause modified data to flow *out* of the cache as well. An array modified in a loop nest is completely replaced in the cache by incoming data in untiled execution. Consequently, there is an additional outgoing sweep of writebacks for *each* time an array is modified. Let $U_w(\ell)$ identify the set of arrays which are modified in a loop nest ℓ, where $U_w(\ell) \subseteq U(\ell)$. Across all loop nests, the number of additional sweeps due to writebacks is given by the *number of times* arrays are modified across all loop nests. That is,

$$s_u^w = \sum_{\ell \in L[G]} |U_w(\ell)|.$$

In tiled execution, it is guaranteed that a portion of each array is retained in cache until it is no longer needed. As a result, each array is swept through only once across *all* loop nests. Hence, the total number of sweeps to load data into the cache is

$$s_t = n_a.$$

Clearly, $s_t \leq s_u$, since each array is referenced at least once in untiled execution. Indeed, $s_t = n_a$ is the minimum number of sweeps that can be achieved for n_a arrays.

In tiled execution, data also flows out of the cache due to writebacks. However, an array which is modified in two or more loop nests generates only one writeback sweep across *all* loop nests because tiling retains the same portion of the array in cache until the next tile is executed. Consequently, the number of additional writeback sweeps in untiled execution is given by the *number of arrays* which are modified in *any* of the loop nests; that is,

$$s_t^w = \left| \bigcup_{\ell \in L[G]} U_w(\ell) \right|.$$

Clearly, $s_t^w \leq s_u^w$, since untiled execution requires multiple writeback sweeps when the same array is modified in different loop nests. Indeed, s_t^w is the minimum number of writeback sweeps that can be achieved, as $s_t^w \leq n_a$.

We define the *sweep ratio* as the ratio of the total number of sweeps in untiled execution to the total number of sweeps in tiled execution. That is,

$$s_r = \frac{s_u + s_u^w}{s_t + s_t^w} = \frac{\sum_{\ell \in L[G]} |U(\ell)| + \sum_{\ell \in L[G]} |U_w(\ell)|}{n_a + \left| \bigcup_{\ell \in L[G]} U_w(\ell) \right|}.$$

Since $s_t \leq s_t$ and $s_t^w \leq s_u^w$, it is clear that $s_r \geq 1$. As the sweep ratio increases, the benefit from tiled execution increases, because fewer cache misses and writebacks are incurred.

The total execution time for a loop nest sequence is given by

$$T = T_c + T_m,$$

where T_c is the total time for computation and T_m is the total time for servicing cache misses. We assume that the time spent servicing cache misses is directly proportional to the amount of data traffic. Hence, $T_m^u = k \cdot (s_u + s_u^w)$ for untiled execution, and $T_m^t = k \cdot (s_t + s_t^w)$ for tiled execution. The ratio of the cache miss service times is given by

$$\frac{T_m^u}{T_m^t} = \frac{k \cdot (s_u + s_u^w)}{k \cdot (s_t + s_t^w)} = s_r.$$

Finally, we assume that the total computation time T_c is identical for both untiled and tiled execution. This assumption is reasonable because tiling only changes the order in which loop iterations are executed; the amount of computation for those iterations remains the same. For untiled execution, the fraction of total execution time spent servicing cache misses is defined as

$$f_m^u = \frac{T_m^u}{T_c + T_m^u}.$$

Similarly, the fraction for tiled execution is

$$f_m^t = \frac{T_m^t}{T_c + T_m^t}.$$

The speedup S from tiled execution is given by the ratio of execution times. That is,

$$S = \frac{T_c + T_m^u}{T_c + T_m^t}.$$

With some algebraic manipulation, we obtain

$$S = \frac{T_c + T_m^u}{T_c + T_m^t} \cdot \frac{T_m^u / T_m^u}{T_m^t / T_m^t} = \frac{(T_c + T_m^u)/T_m^u}{(T_c + T_m^t)/T_m^t} \cdot \frac{T_m^u}{T_m^t} = \frac{1/f_m^u}{1/f_m^t} s_r = \frac{f_m^t}{f_m^u} s_r.$$

We wish to eliminate f_m^t from the equation in order to characterize the speedup as a function of the sweep ratio and the fraction of time servicing cache misses in the original *untiled* execution. We solve f_m^u and f_m^t for T_c and set the resulting quantities equal to each other. We then solve for f_m^t to obtain

$$f_m^t = \frac{f_m^u}{f_m^u - s_r \cdot (f_m^u - 1)}.$$

Substitution of this quantity into the equation for speedup yields

$$S = \frac{s_r}{f_m^u - s_r \cdot (f_m^u - 1)}. \tag{6}$$

Intuitively, the speedup is bounded from above by s_r, provided the denominator is no less than 1.

The range of values for the sweep ratio is given by $s_r \geq 1$ and the fraction f_m^u must be in the range $0 \leq f_m^u \leq 1$. If we assume that computation is infinitely fast, then $T_c = 0$ and $f_m^u = 1$. The speedup in this case is

$$S = \frac{s_r}{1 - s_r \cdot (1 - 1)} = s_r,$$

which is governed entirely by the data traffic in and out of the cache. The partial derivative of S with respect to f_m^u is positive for $s_r > 1$, hence the speedup monotonically increases from $f_m^u = 0$ to $f_m^u = 1$. Consequently, the maximum occurs at $f_m^u = 1$ and the sweep ratio is the upper bound on speedup.

More realistically, computation speed is finite and $f_m^u < 1$. As a result, the speedup will be less than the bound given by the sweep ratio. However, as the sweep ratio increases, there is also a bound determined by f_m^u. That is,

$$\lim_{s_r \to \infty} \frac{s_r}{f_m^u - s_r \cdot (f_m^u - 1)} = \lim_{s_r \to \infty} \frac{1}{f_m^u / s_r - (f_m^u - 1)} = \frac{1}{1 - f_m^u}.$$

The partial derivative of S with respect to s_r is also positive for $s_r \geq 1$, hence the speedup asymptotically increases towards the limit of $1/(1 - f_m^u)$ as s_r increases.

Current trends indicate that processor speeds will continue to increase at a faster rate than memory speeds. We use Equation 6 to predict the effect of this increasing gap on the speedup obtained with cache-partitioned tiling. We model the speed gap by assuming memory speed remains constant while processor speed increases. For the base case, we assume that $T_c^{(1)} = 2 \cdot T_m^u$, where $T_c^{(1)}$ is the computation time for the base processor speed. From this relationship, we obtain $f_m^u = 1/3$. If the processor speed increases by a factor of two, then the computation time is reduced by a factor of two, i.e., $T_c^{(2)} = T_c^{(1)}/2 = T_m^u$. As a result, we obtain a new value $f_m^u = 1/2$. We repeat this procedure for processor speed increases by factors of 2, 4, and 8 over the base case, and plot the speedup of cache-partitioned tiling in Figure 16 as a function of the sweep ratio. The results clearly demonstrate that the benefit of cache-partitioned tiling increases as the gap between processor and memory speeds increases.

7 Experimental Results

In this section, we present experimental results that demonstrate improvements in performance when cache-partitioned data layout and tiling are applied to real programs. Our experiments are conducted on a single processor of the KSR1 multiprocessor from Kendall Square Research. This

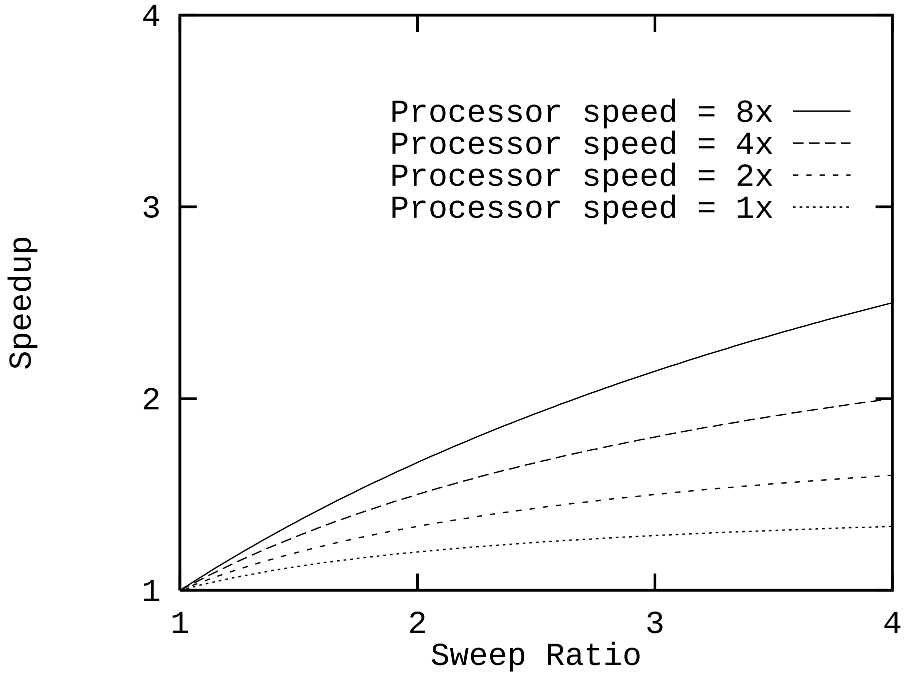


Figure 16: Analytical performance extrapolation for faster processors

choice of platform is motivated by the existence of hardware performance monitoring capabilities in each processor, allowing us to measure not only total execution time, but also the number of cache misses and the time spent servicing cache misses. The KSR1 processor implements a 64-bit superscalar architecture and has a peak performance of 40 MIPS/40 MFLOPS. Each processor has a 512-Kbyte *subcache*, split into a 256-KByte instruction cache and 256-KByte data cache. The subcache is 2-way set-associative with a random replacement policy, and the line size is 64 bytes. The access latency for the subcache is 2 processor cycles and subcache accesses are pipelined. A 32-Mbyte *local cache* is also associated with each processor to implement the cache-only memory architecture (COMA) of the KSR1 multiprocessor system. Local cache accesses resulting from subcache misses incur a latency of 20 processor cycles, during which the processor stalls. If data is not found in the local cache, the latency is increased further by a search to locate the required data in the local cache of another processor. However, since our experiments are conducted on a single processor, all data is automatically placed in the local cache and the COMA feature is not relevant. We are interested primarily in the performance impact of misses in the processor data subcache. For brevity, all subsequent uses of the term *cache* denote the KSR1 data subcache.

Table 2: Characteristics of loop nest sequences

Loop nest sequence	Number of loop nests	Number of arrays	Array size	Num. partitions n_p	Sweep statistics s_u	s_u^w	s_t	s_t^w	Sweep ratio s_r
`calc`	5	6	256×256	4	13	6	6	5	1.73
LL18	3	9	256×256	6	16	6	9	6	1.47
`conduc`	3	8	257×257	4	12	4	8	4	1.33
SIMPLE (1)	11	19	128×128	11	55	15	19	15	2.06
SIMPLE (2)	4	11	128×128	11	19	8	17	7	1.13

The code chosen for experimentation consists of three application kernels and one complete application. We choose kernels in order to isolate the performance impact of tiling. From the SIMPLE hydrodynamics benchmark [8, 1], we consider a kernel of three loop nests extracted from the *conduc* subroutine. From the Livermore Loops benchmark, we consider Kernel 18, an excerpt from a hydrodynamics application. From the application `qgbox` [7], which implements a finite-difference vorticity equation in ocean modelling, we consider a kernel of loop nests from the `calc` subroutine. Finally, we consider the entire SIMPLE application to in order to demonstrate the use of tiling on a larger scale. Table 2 summarizes the characteristics of the code used in our experimentation. The loop nests of SIMPLE are divided into two separate sequences due to a threshold dependence which would require an excessively large tile shift if the entire loop nest sequence were tiled (see Section 5).

Our results are obtained from manual application of cache-partitioned data layout and tiling on the codes. For the data layout, we arbitrarily choose the order in which arrays are declared in the applications as the order in which to place arrays in memory, and compute the necessary gap sizes between arrays using the greedy algorithm described in Section 4. For tiling, subroutine inlining and loop interchange are used to enable the transformation by collecting all loop nests together and ensuring that they traverse their iteration spaces in a compatible manner (see Section 3). Both the untiled and tiled code are instrumented in order to measure execution time and cache misses using the performance monitoring features of the KSR1 processor.

For each experiment, we can estimate the number of cache misses for both tiled and untiled execution. This estimate is based on array sizes, the KSR1 cache line size of 64 bytes (or 8 double-precision floating-point elements), and the quantities s_u and s_t. For example, for `calc`, the size of

Table 3: Experimental results for loop nest sequences

Loop nest sequence	Untiled			Tiled			Speedup from tiling
	Exec. time (sec)	Meas. misses	Est. misses	Exec. time (sec)	Meas. misses	Est. misses	
`calc`	3.569	1052970	1064960	2.884	517771	491520	1.24
LL18	4.028	1312974	1228800	3.361	775701	737280	1.20
`conduc`	3.898	990051	990735	3.651	683445	660490	1.07
SIMPLE (1)	0.7867	136982	112640	0.6598	49006	38912	1.19
SIMPLE (2)	0.2083	35091	38912	0.2038	32328	34816	1.02

each array is 256×256, and a complete sweep through an array of this size incurs $256\cdot256/8 = 8192$ misses. Untiled execution results in 13 complete sweeps through arrays of this size, whereas tiling for reuse requires only 6 sweeps. If the loop nest sequence is executed n times, then the estimated number of misses for untiled execution is $13\cdot 8192\cdot n = 106496\cdot n$, and for tiled execution, the estimated number of misses is $6\cdot 8192\cdot n = 49152\cdot n$.

The experimental results for the three kernels are shown in Table 3. For each kernel, the loop nest sequence is executed 10 times, and the results in the table are the totals for all 10 iterations. The trend in measured speedups correlates with the performance bounds (the sweep ratios in Table 2). Also shown are the estimated and measured number of cache misses for each kernel. The results indicate that cache-partitioned data layout and tiling are successful in reducing the number of misses to the predicted level.

The results for the complete SIMPLE application are also shown in Table 3. The loop nests are executed 20 times, and the results in the table are *averages* for each tiled portion of the application. The measured number of cache misses from part 2 of the loop nest sequence agree with the estimates. However, the measured results from part 1 do not agree with the estimates. This discrepancy is due to a number of small, one-dimensional arrays which are referenced in the loop nests in part 1. These small arrays are ejected frequently from the cache as the data from the main arrays sweeps through the cache, hence the number of misses increases for each time they must be reloaded. Nonetheless, tiling is effective in significantly reducing the *total* number of misses for part 1.

We also compare the estimated speedup from Equation 6 with the measured speedup. Table 4 provides the fraction of execution time spent servicing cache misses and the estimated speedup

Table 4: Estimated and measured speedup

Kernel or application	Miss time (sec)	Exec. time (sec)	Miss fraction f_m^u	Sweep ratio s_r	Est. speedup	Meas. speedup
`calc`	1.433	3.569	0.401	1.73	1.20	1.24
LL18	1.748	4.028	0.434	1.47	1.16	1.20
`conduc`	1.257	3.898	0.322	1.33	1.09	1.07
SIMPLE	6.386	23.23	0.275	1.67	1.12	1.13

Table 5: Memory overhead

Kernel or application	Measured mem. overhead	Worst-case mem. overhead
`calc`	11.5%	25.0%
LL18	4.6%	25.0%
`conduc`	11.0%	24.8%
SIMPLE	27.7%	100.0%

computed with this fraction and the sweep ratio. For SIMPLE, a single sweep ratio is obtained by combining the statistics from parts 1 and 2 in Table 2, and the speedups are for the overall application. For all our experiments, the measured speedups agree very closely with the analytical results; the maximum error is less than 4%.

Finally, we present measurements of the memory overhead from enforcing cache-partitioned data layout in our experiments in Table 5 compared to the worst-case overhead predicted by Equation 4. The overhead is low, even for the relatively small array size used for SIMPLE. A larger array size reduces the memory overhead.

In summary, our experimental results demonstrate that cache-partitioned tiling improves performance by significantly reducing the number of cache misses. In fact, the number of cache misses is reduced to the minimum that can be achieved (with the exception of misses due to the small one-dimensional arrays in the SIMPLE application). Our model for estimating the speedup due to the reduction in cache misses has been validated by our results. Given the current trend of increasing processor speeds relative to memory speeds, the impact of this reduction will be greater for future processors and the resulting speedups will therefore be higher.

8 Summary and Conclusions

This paper has presented cache-partitioned tiling, a novel approach for cache locality optimization. Cache-partitioned tiling has three advantages compared to previous techniques: a *global* view towards locality enhancement for multiple arrays across multiple loop nests, a *systematic* data-management approach which is *integrated* with the code restructuring in order to eliminate conflicts and retain reusable data in the cache, and a *guaranteed* reduction in the number of cache misses to the minimum possible. We have described cache-partitioned tiling in sufficient detail to permit its inclusion in a compiler.

Cache-partitioned tiling consists of data layout and code restructuring; both rely on cache partitioning, in which the cache capacity is divided into a number of equal-sized sections. This partitioning is performed entirely in software; no additional hardware is required. An analysis of data usage across the set of loop nests in a program is performed to identify array reuse and derive array lifetimes in cache, as well as the cache coresidency sets for loop nests. The largest coresidency set determines the number of cache partitions needed. Arrays are then assigned to logical partitions such that no two arrays share the same partition if their lifetimes overlap, which eliminates conflict misses. The resulting assignment constrains the layout of the arrays in memory; consequently, appropriately-sized gaps are inserted between arrays to adjust the starting addresses. In order to reduce memory overhead, a greedy algorithm is used to select the logical-to-physical partition mapping which results in the smallest gap size for each array.

The code for the loop nests is then restructured with a tiling transformation to interleave iterations from different loop nests which reference the same data. This transformation eliminates capacity misses by loading data only once into the cache, where it remains to be reused until it is no longer needed. The step size for the tile controlling loop, which we refer to as the tile size, is carefully chosen to ensure that array data referenced within a tile completely fills each cache partition without overflowing in order to fully utilize the cache capacity. In the presence of data dependences between loop nests, tile shifts are systematically introduced where necessary to prevent violating dependences which are carried by the tile controlling loop.

We have derived an analytical model for estimating the performance improvement which results from cache-partitioned tiling. The model parameters include the sweep ratio, which measures the reduction in data traffic with cache-partitioned tiling, and the fraction of execution time spent servicing cache misses. We have used the analytical model to determine upper bounds on speedup and to demonstrate that the benefit of cache-partitioned tiling increases as the gap between processor and memory speeds increases.

We have applied cache-partitioned tiling to real applications consisting of many loop nests and performed experiments to measure the resulting improvement in execution time. The results have verified our analytical model; the error between estimated and measured speedups does not exceed 4%. Furthermore, the memory overhead resulting from enforcing a cache-partitioned data layout is low, even when the array sizes used in our experiments are small relative to the cache size.

Our work on cache-partitioned tiling can be extended in a number of ways. The algorithm for enforcing a conflict-free layout of arrays in memory may be improved to further reduce the memory overhead. Cache-partitioned tiling could be extended to deal with multiple levels of cache memory. Another interesting avenue for future work is to extend cache-partitioned tiling for parallel execution on shared-memory multiprocessors with non-uniform memory access times. The distribution of data across a number of memory modules in such systems enlarges the scope for data layout decisions. In addition, reducing data traffic with cache-partitioned tiling has greater implications for parallel performance, as contention in parallel systems increases rapidly with data traffic. Since tuning parallel applications is a challenging task, systematic performance optimizations which can be automated have much to offer in this area.

References

[1] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proc. ACM SIGPLAN'93 Conf. on Prog. Lang. Design and Implementation*, pages 112–125, Alburqueque, New Mexico, June 1993.

[2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. Technical Report UCB/CSD-93-781, Computer Science Division, University of California, Berkeley, 1993.

[3] Harpreet Singh Gill. Improved Optimization Strategies for Blocked Algorithms. Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 1994.

[4] Y.-J. Ju and H.Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 344–358. Springer-Verlag, Berlin, 1992.

[5] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing—Sixth International Workshop*, pages 301–320. Springer-Verlag, Berlin, 1994.

[6] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. 4th Int. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, pages 63–74, Santa Clara, Ca., April 1991.

[7] John D. McCalpin. Quasigeostrophic box model–revision 2.3. Technical report, College of Marine Studies, University of Delaware, July 1992.

[8] Keshav Pingali and Anne Rogers. Compiler parallelization of SIMPLE for a distributed memory machine. In Joel Saltz and Piyush Mehrota, editors, *Languages, Compilers, and Run-time Environments for Distributed Memory Machines*, pages 63–78. North-Holland, Amsterdam, 1992.

[9] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, June 1991.