*To Copy or Not to Copy: A Compile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts**

Olivier Temam[†], Elana D. Granston[†] William Jalby[‡]

University of Leiden, University of Versailles

Abstract

In recent years, loop tiling has become an increasingly popular technique for increasing cache effectiveness. This is accomplished by transforming a loop nest so that the temporal and spatial locality can be better exploited for a given cache size. However, this optimization only targets the reduction of capacity misses. As recently demonstrated by several groups of researchers, conflict misses can still preclude effective cache utilization. Moreover, the severity of cache conflicts can vary greatly with slight variations in problem size and starting addresses, making performance difficult to even predict, let alone optimize. To reduce conflict misses, data copying has been proposed. With this technique, data layout in cache is adjusted by copying array tiles into temporary arrays that exhibit better cache behavior. Although copying has been proposed as the panacea to the problem of cache conflicts, this solution experiences a cost proportional to the amount of data being copied. To date, there has been no discussion regarding either this tradeoff or the problem of determining *what* and *when* to copy. In this paper, we present a compile-time technique for making this determination, and present a selective copying strategy based on this methodology. Preliminary experimental results demonstrate that, because of the sensitivity of cache conflicts to small changes in problem size and base addresses, selective copying can lead to better overall performance than either no copying, complete copying, or copying based on manually applied heuristics.

Key words: selective copying, cost– benefit analysis, data copying strategies, loop tiling, data reuse, cache conflicts, compiler-directed cache management, program optimization.

1 Introduction

In recent years, loop tiling [1, 6, 11, 12, 13] has become an increasingly popular technique for increasing cache effectiveness. Accomplished via a combination of strip-mining, interchanging, and occasionally skewing, loop tiling can be used to transform a loop nest so that the temporal and spatial locality can be better exploited for a given cache size. A simple example of loop tiling is presented in Figure 2, where a tiled version of the loop nest from Figure 1 is presented.

```
DO j1 = 0, N3-1
  DO j2 = j1, N1-1
    DO j3 = j2, N2-1
      C(j1,j3) += A(j1,j2) * B(j2,j3)
    ENDDO
  ENDDO
ENDDO
```

Figure 1: *Original loop nest.*

```
DO jj2 = 0, N2-1, B2
  DO jj3 = 0, N3-1, B3
    DO j1 = 0, N1-1
      DO j2 = jj2, min(jj2+B2-2,N2-1)
        DO j3 = jj3, min(jj3+B3-1,N3-1)
          C(j1,j3) += A(j1,j2) * B(j2,j3)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Figure 2: *Tiled loop nest.*

Unfortunately, loop tiling only targets the reduction of *capacity misses*, namely misses that are due to using a cache whose size is too small to hold the working set of cache lines. Typically, however, caches have a very low degree of associativity. Often, they are direct-mapped. Therefore, as demonstrated in [3, 7, 10], they can still suffer from a high degree of *conflict misses*, thereby precluding effective cache utilization. Moreover, the severity of performance degradation can vary greatly with slight variations in problem size and starting addresses, making performance difficult to even predict, let alone optimize.

To illustrate this, consider the reference to B in the loop nest in Figure 2. During any given execution of the j_1 loop (i.e., execution of all N_1 iterations of the j_1 loop), the same tile of array B is accessed. Therefore, if this tile of B_2B_3 elements can be brought into cache on the first iteration of the j_1 loop and remain there until the last iteration, $(N_1 - 1)B_2B_3$ of the $N_1B_2B_3$

*Support was provided by the Esprit Agency DGXIII under Grant No. APPARC 6634 BRA III.

†High Performance Computing Division, Department of Computer Science, Leiden University, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands

‡Universite de Versailles, 78000 Versailles, France

accesses to elements in this tile can be eliminated.

Can this tile be kept in cache during this period? Note that this tile consists of B_2 intervals of B_3 elements. Assume row major order and suppose that a direct-mapped cache is used. The virtual address corresponding to reference $B(j_2, j_3)$ is ADDR $B + j_3 + N_2 j_2$, where N_2 is the leading dimension of array B and ADDR B is the base address of array B. Assuming that the cache can hold C_S array elements, the cache distance between the starting locations of two consecutive intervals is $N_2 mod\ C_S$. As can be seen in Figure 3, if $N_2 mod\ C_S < B_3$, the elements of the different intervals associated with the tile will *interfere* with each other in cache, causing elements to be flushed prematurely.

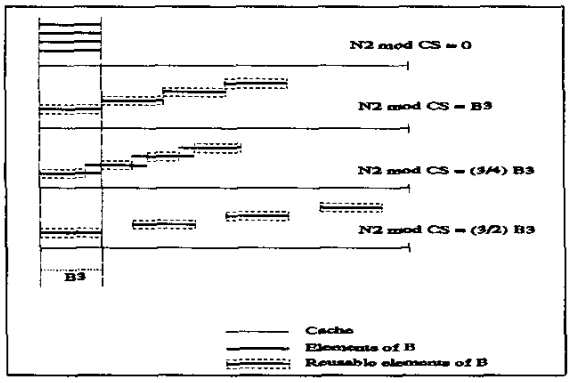

Figure 3: *Four possible cache mappings for the tile associated with the reference to array B in the tiled loop nest (Figure 2).*

To reduce conflict misses, [7] have proposed *data copying*. With this technique, data layout in cache is adjusted by copying array tiles to temporary arrays that exhibit better cache behavior. For example, as shown in Figure 4, to eliminate the aforementioned interferences between elements within a tile of array B, the tile can be copied so that intervals are stored consecutively in a temporary $COPY$ array. The copying is performed before executing the loop where the array is reused. When the loop is executed, the copy is accessed instead of the original array. Because the copy has a better *footprint* in cache, reuse preventing interferences among the elements of the tile are avoided.

The effects of cache conflicts and data copying on the example loop nest can be seen graphically in Figure 5, where hit ratios for three versions of tiled matrix-matrix multiply are depicted, as the problem size N is varied, where $N = N_1 = N_2 = N_3$, and all matrices are of size $N \times N$.

Architectural parameters are based on those of Digital's Alpha chip [8]. We assume a direct-mapped

```
DO jj2 = 0, N2-1, B2
  DO jj3 = 0, N3-1, B3

C   Copy tile for array B into array COPY

    DO k2 = jj2, min(jj2+B2-2,N2-1)
      DO k3 = jj3, min(jj3+B3-1,N3-1)
        COPY(k3+B3*k2) = B(k2,k3)
      ENDDO
    ENDDO

    DO j1 = 0, N1-1
      DO j2 = jj2, min(jj2+B2-2,N2-1)
        DO j3 = jj3, min(jj3+B3-1,N3-1)
          C(j1,j3) += A(j1,j2) * COPY(j3+B3*j2)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Figure 4: *Tiled loop nest from Figure 2 after copying array B.*

cache of $C_S = 1024$ elements (8K bytes) partitioned into lines of $L_S = 4$ elements (32 bytes) each. Based on these cache parameters, a block size of $B_2 = B_3 = 24$ has been selected, which is in the range recommended in [7] when no copying is used.

The three versions correspond to the following cases:
(1) no copying as shown in Figure 2,
(2) copying only array B as shown in Figure 4, and
(3) complete copying, where all arrays (A, B, and C) are copied.
Note the instability in the hit ratio for the no copying case, which varies greatly with small changes in problem size. In contrast, the hit ratio is fairly stable when copying is used. This trend can also be observed with other benchmarks [10]. In the case of matrix multiply, most of the benefit can be derived from just copying array B, indicating that most of the conflict misses are due to this array. However, slightly more benefit can be derived from copying other arrays as well.

If we were to judge solely by hit ratios, clearly, we would always want to copy everything possible. However, *copying is not free*. Furthermore, hit ratios alone do not tell the whole story. First, there is loop overhead in creating a copy. Second, whenever a temporary $COPY$ array is loaded with data from an original array, the elements from the original array are loaded into cache. Because a copy is being made, these elements will not be reused. Hence, they pollute the cache.

The overall performance of these same three cases has also been simulated, accounting for delays due to misses and instruction overhead for copying. A memory access latency of 20 cycles is assumed on a cache miss. Copying is assumed to require 5 cycles per array element. Figure 6 shows the overhead for copying

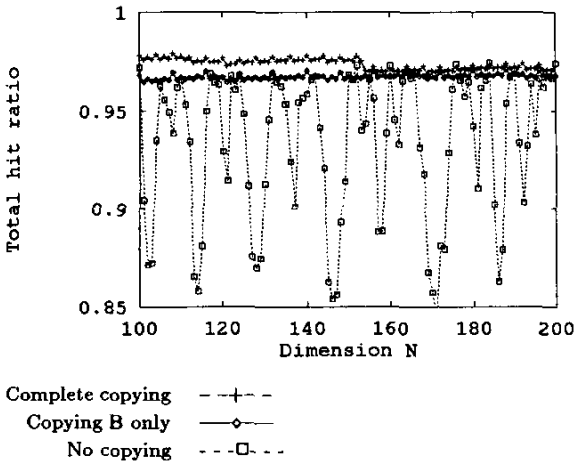

Figure 5: *Hit ratio of tiled matrix-matrix multiply with and without copying.*

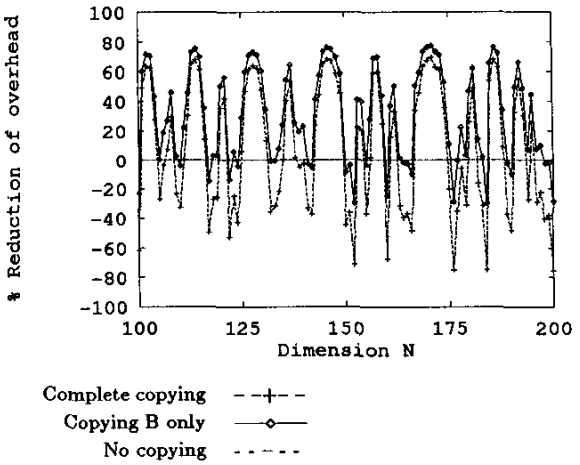

Figure 6: *Simulated performance of tiled matrix-matrix multiply, with and without copying.*

relative to the case of no copying, where

The remainder of this paper is organized as follows.

Assessment of Costs and Benefits The ability to effectively apply data copying is *critical* for efficient utilization of loop tiling. [4, 7] demonstrate experimentally that on average without copying only a small fraction of the cache can be efficiently utilized, or equivalently, that the best tile size can be far smaller than the theoretical optimum. This significant difference arises because cache interferences are ignored in the computation of the theoretically optimum tile size. However, when copying is used, most significant interferences can be removed. Therefore, the best tile size approaches the theoretical optimum and more of the cache space is effectively utilized.

The benefits achieved from data copying are directly proportional to the impact of cache interferences. For example, suppose that in a loop nest of index j_1 two references $Y(j_1)$ and $Z(j_1)$ overlap in cache, but only by one or two cache lines. Then, the cost of copying probably outweighs potential gains.

The cost is proportional to both cost of making the copy and the cost of useful data that is flushed from cache in the process. However, copying has a cost. Consider Figure 4 where array B is copied into array $COPY$. Because all elements of B must still be brought into cache once, they can flush elements from other arrays, including array $COPY$, that could otherwise have been reused. Similar pollution occurs when a temporary array is copied back to an original array. This pollution increases the cost of copying.

2.2 Strategies

On what basis should copying decisions be made? In the literature, complete copying and copying based on manual insights have been used. We describe these two approaches. We also introduce the notion of *selective copying* based on a cost–benefit analysis of the worthwhileness of copying.

Complete Copying If complete copying is used, as proposed by Bodin et al. [1], all copyable arrays are blindly copied. The term *copyable* excludes those that cannot be copied, due to complicated subscript expressions, or because coherency problems might arise. For example, such problems might arise when indirect addressing is used.

With respect to the copyable references, the cache is managed like a local memory, so no interferences can arise among copied arrays. However, as shown in Figure 6, complete copying can entail significant overhead that can easily outweigh the benefits.

Heuristic Copying Alternatively, we can copy only some of the tiles. Although no automatic selection method is discussed in the literature, several heuristics have been manually applied to sample codes based on researchers' insights. The first of these heuristics is based on detecting when the intervals associated with a tile might overlap and copying to avoid self-interferences. For example, [7] present a case study of matrix-matrix multiply (Figure 2), where the authors detect that the most significant source of interference is due to self interference on array B. Using this insight, they successfully eliminate a major source of interferences by copying this array. Another insight that can be used is to note that regular and severe interference can occur when two references have the same linearized stride, and have base addresses that map close together in cache.

Selective Copying The above and other heuristic approaches can detect some cases where copying *might* be useful. However, these heuristics are not easily automatable. Moreover, as demonstrated in Section 1, the benefits of copying can vary drastically with small changes in problem size and starting address. To overcome these limitations, we propose *selective copying*, whereby the benefits and costs of copying are estimated on a case-by-case basis, and copying applied only where profitable.

As an example of the benefits of using selective copying, recall the performance graph in Figure 6. For some problems sizes, only array B should be copied and for some problem sizes, copying should not be used all at. Because of the instability demonstrated by the example presented here and by other experimental studies [10], only an automatable analysis technique can consistently select the best option.

3 Definitions

We now present formal definitions of concepts that are needed for the development of our selective copying algorithm that is described in Section 4.

3.1 Reuse Sets and Interference Sets

Consider a tiled loop nest in which there are n loops $j_1, \ldots, j_n$, n being the loop level of the innermost loop.

```
DO j1 = 0 , N1-1
  DO j2 = 0 , N2-1
     ...
     DO jn = 0 , Nn-1
       Loop Body
     ENDDO
     ...
  ENDDO
ENDDO
```

There are m controlling loops and $n - m$ tiled loops. For example, in Figure 2, the controlling loops are $jj_2, \ldots, jj_m, j_1$, and the tiled loops are $j_2, j_3 \ldots j_{n-m+1}$.)

Definition 3.1 *A* **reference** $R(c_1 j_{q_1} + d_1, c_2 j_{q_2} + d_2, \ldots, c_n j_{q_n} + d_n)$, $1 \leq q_i \leq n$, *is a memory access to virtual address* $r = r_0 + \sum_{i=1}^{n} r_i j_i$. *(Note: q_i's need not be distinct.)*

Note that we limit subscript expressions to a subset of those that can be expressed as linear combinations of the loop indices. This restriction should not be very limiting in practice, since such subscript expressions encompass a very large number of array references commonly found in numerical codes [14]. These

Definition 3.7 *The cache space occupied by a set, potentially wrapping around cache, and ignoring holes is the* **envelope** *of the set, denoted* ENV(< set >).

Definition 3.8 *The* **density** *of a set, denoted* DEN(< set >), *is the ratio of the number of elements*

$N_2 \bmod C_S$	theoretical reuse set size $\lvert \mathsf{TRS}(R_B) \rvert$	actual reuse set size $\lvert \mathsf{ARS}(R_B) \rvert$	size of envelope $\lvert \mathsf{ENV}(\mathsf{FT}(R_B)) \rvert$	density of envelope $\mathsf{DEN}(\mathsf{FT}(R_B))$
0	B_2B_3	0	B_3	1
B_3	B_2B_3	B_2B_3	B_2B_3	1
$\frac{3}{4}B_3$	B_2B_3	$\frac{1}{2}(B_2+1)B_3$	$\frac{1}{4}(3B_2+1)B_3$	1
$\frac{3}{2}B_3$	B_2B_3	B_2B_3	$\frac{3}{2}(B_2-1)B_3$	$\frac{2B_2}{3(B_2-1)}$

Table 1: *Characteristics of the reuse set for the reference R_B to array B in Figure 2 for each of the four mappings shown in Figure 3.*

If references R and R' are in translation, the two arrays are moving through cache at the same speed.

Definition 3.12 *A* **translation group** τ *includes all references that are mutually in translation.* [2]

Observation 3.1 *The translation groups $\tau_1, \ldots, \tau_{n_\tau}$ partition the array references of a loop into equivalence classes.*

Each translation group τ_i can be further subdivided into into *array groups.*

Definition 3.13 *If there exist two references $R, R' \in \tau_i$ such that their reuse sets intersect when measured in cache lines, then R and R' belong to the same* **array group** α^i.

For this determination, only intersections between references in the same translation group are considered. We ignore those between references of distinct translation groups.

Observation 3.2 *The array groups $\alpha^i_1, \ldots, \alpha^i_{n_\alpha}$ of a translation group τ_i partition the references of a translation group into equivalence classes.*

Examples To illustrate the concepts of translation groups and array groups through examples, consider a normalized loop nest where loop indices j_2 and j_3 correspond to the tiled loops. Assume that these two loops have upper bounds N_2 and N_3, respectively. Assume that reuse groups are measured in units of cache lines. Suppose the loop nest contains the pair of references $U(j_3)$ and $U(j_3+5)$ (with $N_3 > 5$). Because this pair has a non-zero intersection between their reuse sets, they would belong to both the same translation group and the same array group.

[2] The notion of an translation group is based on an intuition similar to the notion of uniformly generated dependences [5]. Specifically, both are based on the notion of references whose relative difference in memory addresses is constant over time. However, the use of this information is quite different. We are interested in collecting groups of references with similar motion with respect to their footprints in cache, while uniformly generated dependences capture those references that access the same data. Therefore, a translation group can include references that access differing arrays, while a group of uniformly generated dependences, by definition of dependence, clearly does not.

Consider instead the pair of references $U(j_3)$ and $U(j_3+N_3+5)$. In this case, the pair would belong to the same translation group. However, there is no intersection between their actual reuse sets. Therefore, the two would belong to distinct array groups.

Suppose that the loop nest contained the two references $V(j_3)$ and $W(j_3)$. This pair would belong to the same translation group but, trivially, to distinct array groups. Suppose we consider a pair of 2 dimensional references: $A(j_2, j_3)$ and $B(j_2, j_3)$. If A and B have the same leading dimension, then their positions are constant and they belong to the same translation set. In either case, they belong to different array groups.

Meanwhile, the pair $V(j_3)$ and $V(j_2)$ would correspond to distinct translation groups, and therefore, even though they reference the same array, to distinct array groups. If line size is greater than one, the pair $V(2 * j_3)$ and $V(2 * j_3 + 1)$ belong to the same array group, even though they never reference the same array elements.

The notions of *reuse set, interference set, dimension, envelope* and *density* defined earlier in terms of a single reference can be applied to a group of references as well. Because the definitions are the natural extensions of these terms, formal definitions are omitted in the interest of brevity.

3.3 Interferences

As suggested in [7], interferences can be partitioned into self and cross interferences. We further partition the latter category into internal and external cross interferences.

Definition 3.14 *Interference between references R and R' is termed* **internal cross interference** *if R and R' belong to the same translation group.*

Definition 3.15 *Interference between references R and R' is termed* **external cross interference** *if R and R' belong to distinct translation groups.*

4 Selective Copying

4.1 Overview

Selective copying can be used to evaluate the costs and benefits of eliminating each of the three categories of interferences. This is done in three steps.

- **Step 1:** Each array group is evaluated separately. Where benefits outweigh costs, references that suffer from self interference are copied.
- **Step 2:** Each translation group is processed separately. Where benefits outweigh costs, references that exhibit internal cross interference are copied.
- **Step 3:** Interferences between translation group are evaluated. Where benefits outweigh costs, references that exhibit external cross interference are copied.

Let t_{lat} be the number of cycles required to access a line from memory, and t_{copy} be the instruction overhead (in cycles) for copying a single element. Given, a memory reference R, its benefit from copying is the expected decrease in the number of memory accesses multiplied by the memory latency.

$$\mathsf{benefit}(R) = \#\_of\_misses\_removed \times t_{lat},$$

where $\#\_of\_misses\_removed$ is the total number of interference-related cache misses eliminated during the entire execution of the loop. Meanwhile, the *cost* of copying R is the overhead for copying, that includes both the additional memory accesses and the instruction overhead for performing the copying.

$$\begin{aligned}\mathsf{cost}(R) &= (\#\_of\_cache\_lines\_to\_be\_copied) \times t_{lat} \\ &\quad +(\#\_of\_elements\_to\_copy) \times t_{copy}.\end{aligned}$$

Intuitively, a reference is copied if the benefit outweighs the cost. All the cost/benefit computations presented in this paper can be done symbolically, with final decisions delayed until run time when necessary.

At any given step, only interferences due to the type of interference being targeted are evaluated. Note that deciding to copy an array during one step can alter the translation set to which it belongs. This must be taken into account during later steps.

By eliminating *damaging* self interferences first (i.e., all those for which the benefit outweighs the cost), those that remain after Step 1 can generally be considered to be insignificant. Therefore, they can be ignored during Step 2.

Similarly, after Step 2, damaging self and internal cross interferences have been eliminated. Therefore, when evaluating interferences between translation groups in Step 3, we can assume that there are no significant interference within translation groups.

Note that state of the art algorithms for optimizing data locality [1, 6, 11] compute the optimal block size assuming the sum of the tile sizes is smaller than the cache. Therefore, there should be sufficient cache space for all tiles (equivalently, reuse sets) to be copied. In opposition to the technique described in [1], however, all tiles are not necessarily copied.

Section 4.2 discusses coherence and pollution issues. Sections 4.3–4.5 expand on each of these steps, respectively. Additional details and examples can be found in [9, 10].

4.2 Maintaining Coherence and Minimizing Data Duplication

Consider two references R and R' with non-empty reuse sets. Suppose there is a dependence between the two reuse sets that is carried by a tiled loop. In general, the degree of overlap between their respective reuse sets depends on the array group to which these references belong. If R and R' belong to different array groups, then the overlap is likely to be relatively minor. If R and R' belong to the same array group, then overlap is likely to be very significant.

Therefore, we use the simple heuristic: *array groups are copied together.* A dominant tile is computed for the array group, and the cost of copying the group is computed as the cost of copying the dominant tile. Based on this assumption, it is straightforward to detect and mark uncopyable references as a preprocessing step. The following rule is used: *if there is a dependence between two references R and R' such that*
(1) *one of R and R' is a write, and*
(2) *either R and R' belong to distinct array groups, or one of R and R' has been previously marked uncopyable,*
then both R and R' are marked uncopyable.

4.3 Step 1: Removing Self Interferences

Proposition 4.1 *If reuse occurs on loop level l, the total number of additional memory requests due to self interferences for one reference is*

$$N_1 \times \ldots \times N_l \times (\mid \mathsf{TRS}(R) \mid - \mid \mathsf{ARS}(R) \mid).$$

Therefore, the benefit of copying (in cycles) corresponds to the number of self-interference misses avoided:

$$\mathsf{benefit(R)} = \mathsf{N_1} \times \ldots \times \mathsf{N_l} \times (\mid \mathsf{TRS}(R) \mid - \mid \mathsf{ARS}(R) \mid) \times \mathsf{t_{lat}}.$$

If copying is used, the reuse set must be copied $N_1 \times \ldots \times N_{l-1}$ times (since the reuse set changes on loop $l-1$). The cost corresponds to the number of memory requests needed to load the elements of the reuse set of R ($\mid \mathsf{TRS}(R) \mid$ requests, where $\mid \mathsf{TRS}(R) \mid$ is expressed in cache lines) and to the instructions necessary to make the $N_{l+1} \times \ldots \times N_n$ copies. Furthermore, each time the reuse set is copied, it flushes $\mid \mathsf{ARS}(R) \mid$ cache lines from cache, and therefore brings the same amount of cache misses [3]. Consequently, the total cost of copying in cycles is

$$\begin{aligned}\mathsf{cost}(R) = N_1 \times \ldots \times N_{l-1} \times ((&\mid \mathsf{ARS}(R) \mid + \mid \mathsf{TRS}(R) \mid) \times t_{lat} \\ &+(N_1 \times \ldots \times N_{l-1} \times N_{l+1} \times \ldots \times N_n) \times t_{copy})\end{aligned}$$

If one reference is the victim of self interferences, other references within the same translation group are

[3] this is a conservative estimate of the cost

also victims of self interferences. Consequently, it is often possible to evaluate the cost and benefit for one array group (or reference, if there is a single reference in an array group), and extend the result to all other array groups (or references) within the translation group [10].

The main steps of the algorithm for removing self interferences via copying are the following:

```
/* Step 1: Remove self interferences */
For each translation group τ
    Define an array COPY_τ.
    /* COPY_pos is the index of COPY_τ
       where the next tile will be copied. */
    Initialize copy position COPY_pos to 0
    For each array group α in τ
        Compute cost and benefit of copying α
            to remove self interferences
        If benefit(α) > cost(α)
            Copy α into consecutive locations of
                COPY_τ starting at index COPY_pos.
        Endif
    Endfor
Endfor
```

Note that copying to remove self interferences can also remove internal cross interferences. Suppose there were internal cross interferences between two references prior to Step 1. If both these references are copied, the cross internal interferences between them are also eliminated at the end of this step. Further note however, that copying increases the size of the actual reuse set to that of the theoretical reuse set. Because of this, copying can potentially increase external cross interferences.

Example Let us consider the example of Figure 2, i.e., tiled matrix-matrix multiply. Let us pick two cases that induce none and many interferences respectively according to graph 5, i.e., $N = 176$ and $N = 128$. Let us also choose $B_2 = B_3 = 24$. Because $N < C_S$ in both cases ($C_S = 1024$), $N \mod C_S = N$. Therefore, the cache distance between two consecutive intervals is N.

Let us consider $N = 176$, and assume that the cache position of the beginning of the first interval is cache location 0 (the initial position does not influence the amount of sel-interferences). The number of possible cache positions for the beginning of the intervals of array B is $\frac{C_S}{gcd\,(N,C_S)} = 64$. Because 64 is larger than the number of intervals, i.e., 24, it is not certain overlapping, i.e., self interference, occurs. Because $24 \times 176 > 1024$, the 24 intervals to be loaded are going to wrap around cache. $\lfloor \frac{1024}{176} \rfloor = 5$, so wrap around occurs on the 6th interval. Similarly every 5 or 6 intervals wrap around occurs. Let us now check the distribution of intervals in cache. After wrap around, the cache position of the 6th interval is 32 (Figure 7). This means the interval falls in between the first and second interval of the previous series of intervals. The distance between the end of the first interval and the beginning of this interval is $32 - 24 = 8$. Similarly, when the second wrap around occurs, the beginning of the first interval of the third series is located 8 cache locations after the end of the first interval of the second series. Therefore, after 6 such series (approximately 36 intervals) overlapping occurs between the first interval of the sixth series and the second interval of the first series (Figure 7). Because $B_2 = 24$, there are approximately 4 series, so no overlapping actually occurs in our case. There are no additional memory requests due to self interferences. If B_2 were equal to or greater than 36, then self interferences would have occurred.

Therefore, in this case, the layout of elements of B in cache is optimal with respect to self interferences, so it is not necessary to copy to avoid them. In fact, copying would actually *degrade* performance, because of the overhead entailed (Figure 6).

Next consider the case where $N = 128$. There are $\frac{C_S}{gcd\,(N,C_S)} = 8$ possible cache positions for the beginning of the intervals of array B (Figure 7). Therefore, self interference begins when the 9^{th} interval is brought in cache. Whenever 16 or more intervals of array B are brought into cache, total overlapping occurs and no element can be reused.

Because $B_2 = 24$, all elements of array B are victims of self interferences. No reuse is possible. There are approximately $B_2 \times \frac{B_3}{L_S}$ additional memory requests due to self interferences, per iteration of loop j_1. In total, there are $\frac{N_1 N_2 N_3}{B_2 B_3} \times (B_2 \frac{B_3}{L_S})$ additional memory requests due to self interferences. Because $N_2 \times N_3$ copy operations are needed to copy array B, the total cost of copying is equal to $\frac{N_2 N_3}{L_S} \times t_{lat} + N_2 N_3 \times t_{copy}$ cycles, whereas the benefit is equal to $\frac{N_1 N_2 N_3}{L_S} \times t_{lat}$ cycles.

Consequently, in this case, copying is extremely useful. It allows reutilization of *all* elements of B, whereas, in the version of the loop nest without copying, *no* element of B can be reused. Thus, the overhead due to copying is far smaller than the benefit.

In this case, $24 \times 8 = 192$ cache locations are flushed when elements to be copied are brought into cache. In contrast, in the previous case where no overlapping occurred, using copying would cause $24 \times 24 = 576$ cache locations to be flushed. Note that the less useful copying is, the more costly it is.

4.4 Step 2: Removing Internal Cross Interferences

Proposition 4.2 *The total number of additional memory requests due to internal cross interferences between two references of the same translation group is*

$$N_1 \times \ldots \times N_l \times \mathrm{DEN}(\mathrm{ARS}(R^1)) \times \mathrm{DEN}(\mathrm{AIS}(R^2)) \times \mathrm{CL}(R^1, R^2).$$

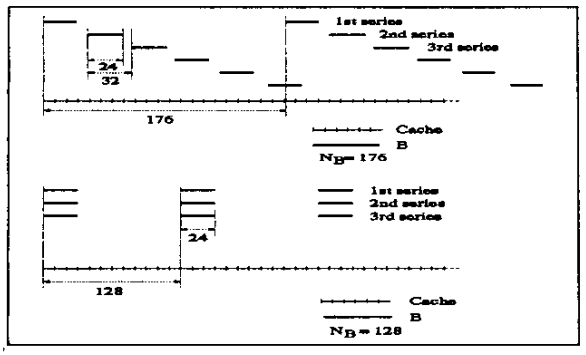

Figure 7: *Effect of the leading dimension of array B from tiled matrix-matrix multiply (Figure 2) on the occurrence of self interferences involving elements of B.*

where $CL(R^1, R^2)$ is the number of cache lines where the envelopes of the actual reuse set of R^1 and actual interference set of R^2 overlap.

Therefore, the benefit of copying corresponds to the number of external cross-interference misses avoided:

$$\text{benefit}=N_1 \times \ldots \times N_l \times \text{DEN}(\text{ARS}(R^1)) \times \text{DEN}(\text{AIS}(R^2)) \times \text{CL}(R^1, R^2).$$

Based on the above, the benefit of copying can be computed for an entire array group [10].

The cost of copying in Step 2 is generally the same as in Step 1, except when the array group has already been selected for copying in Step 1, in which case the array can be copied for free. The algorithm for removing internal cross interferences via copying is shown below:

```
/* Step 2: Remove internal cross interferences */
For each translation group τ
    Define COPY_τ (if not defined in Step 1)
    For each array group α in τ
        Compute total potential benefit from removing
            internal cross interferences within α
        If benefit(α) > cost(α)
            /* Copy α if possible */
            Compute the relative cache distances (or
                "holes") between all actual interference
                sets of all array groups in τ
            If ∃ hole ≥ size of theoretical reuse set of α
                Set COPY_pos to start of hole
                Copy reuse set of α into COPY_τ
                    starting at COPY_pos
            Endif
        Endif
    Endfor
Endfor
```

It is necessary to find a sufficiently large hole between all actual interference sets of the other array groups of the translation group τ, so that no more internal cross interference occur between α and other array groups. The size of the hole needs to be greater or equal than the size of the theoretical reuse set of α since the entire reuse set is copied in consecutive locations of array $COPY_\tau$.

4.5 Step 3: Removing External Cross Interferences

The relative cache positions of the actual reuse set and the actual interference set of two array groups belonging to different translation groups repeat periodically, with a period of P iterations of loop l) [10]. Let I_λ correspond to an interval of possible relative cache positions over a period, and $CL(R^1, R^2; \lambda)$ to the overlapping of the two sets (expressed in cache lines) for a given value of $\lambda \in I_\lambda$.

Proposition 4.3 *The total number of additional memory requests due to external cross interferences between two references belonging to distinct translation groups is*

$$\frac{N_1 \times \ldots \times N_l}{P} \sum_{\lambda \in I_\lambda} \text{DEN}(\text{ARS}(R^1)) \times \text{DEN}(\text{AIS}(R^2)) \times \text{CL}(R^1, R^2; \lambda).$$

The benefit of copying corresponds to the number of external cross-interference misses avoided:

$$\text{benefit}=N_1 \times \ldots \times N_l \times \sum_{\lambda \in I_\lambda} \text{DEN}(\text{ARS}(R^1)) \times \text{DEN}(\text{AIS}(R^2)) \times CL(R^1, R^2; \lambda).$$

The benefit can also be computed for an entire array group. The cost for copying an array group is the same as in Step 1, unless the array was selected for copying in Step 1 or Step 2, in which case the cost is nothing. The algorithm for removing external cross interferences is outlined below:

```
/* Step 3: Remove external cross interferences */
For each translation group τ
    For each array group α in τ
        For each translation group τ' ≠ τ
            For each array group in τ'
                Evaluate amount of external cross
                    interferences with α
            Endfor
            Deduce benefit of copying α with respect to τ'
        Endfor
        If ∃ τ' such that benefit(α,τ') > (α,τ')
            Pick τ' for which benefit is maximal
            Copy reuse set of α into COPY^τ' (or into
                any other COPY array that would not
                conflict with COPY_τ')
        Endif
    Endfor
Endfor
```

It may be possible to copy α into a $COPY$ array other than the one associated with τ', because sometimes the mapping of a translation group is such that this will not interfere with other translation groups, or array groups within other translation groups [10]. Array groups already tagged for copying during Steps 1 or 2 are ignored because copying them again could introduce internal cross interferences, which are usually more significant than external cross interferences.

There are two types of external cross interferences: cross interferences between two references that exhibit temporal reuse, and cross interferences between one

reference which exhibits temporal reuse and another reference that does not. In general, copying to remove the first type of external cross interferences brings little benefit as illustrated in Figure 6. The second type of interferences is more damaging, but it is usually considered to be difficult to remove such interferences. Precise analysis of external cross interferences sometimes allows partial if not total removal of such interferences [10].

5 Summary

Interferences can prevent reuse from being exploited, even in loops that were optimized *specifically* to exploit this reuse. To address this problem, several researchers have proposed the use of data copying. Yet, no guidelines have been presented in the literature for determining what and when to copy.

In this paper, we have demonstrated that, with blind copying, the cost can outweigh the benefit. We have also shown that hit ratio alone cannot be used to determine the optimal set of references to copy. Furthermore, this optimal set can be difficult to assess without analyzing the costs and benefits on a case-by-case basis. Additionally, care must be taken because copying can introduce both coherence problems and pollution via duplication of data. To address these problems, we proposed *selective copying*, a compile-time strategy for determining what and when to copy based primarily on an analysis of cache conflicts. Our strategy uses a simple heuristic approach to preserve coherence and minimize data duplication.

In the development of our strategy, a new framework has evolved for categorizing references and the three types of interferences that can occur. We showed that all three are necessary, as substantially different techniques are needed to asses the impact of each type.

There are three steps in our compile time strategy. During each step, we target the elimination of a particular interference type. Within each step we evaluate the cost and benefit of eliminating interferences of that type on a case-by-case basis. By attacking the interference categories in the specified order, computation is substantially simplified. Computations can be done symbolically at compile time when necessary, with final decisions postponed until runtime.

Acknowledgements

The authors would like to thank Christine Fricker for her thought-provoking insights on the subject of cache interferences.

References

[1] F. Bodin, C. Eisenbeis, W. Jalby, and D. Windheiser. A Quantitative Algorithm for Data Locality Optimization. In *Code Generation-Concepts, Tools, Techniques*. Springer-Verlag, 1992.

[2] C. Eisenbeis, W. Jalby, D. Windheiser, and F. Bodin. A Strategy for Array Management in Local Memory. In *Proceedings of the Third Workshop on Programming Languages and Compilers for Parallel Computing*, 1990.

[3] J. Ferrante, V. Sarkar, and W. Thrash. On Estimating and Enhancing Cache Effectiveness (Extended Abstract). In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, 1991.

[4] C. Fricker, O. Temam, and W. Jalby. Accurate Evaluation of Blocked Algorithms Cache Interferences. Technical report, Leiden University, Mar. 1993.

[5] K. Gallivan, W. Jalby, and D. Gannon. On the Problem of Optimizing Data Transfers for Complex Memory Systems. In *Proceedings of the International Conference on Supercomputing*, pages 238–253, July 1988.

[6] K. Kennedy and K. S. McKinley. Optimizing for Parallelism and Data Locality. In *Proceedings of the International Conference on Supercomputing*, pages 323–334, July 1992.

[7] M. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance of Blocked Algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 1991.

[8] R. L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[9] O. Temam. *Study and Optimization of Numerical Codes Cache Behavior*. PhD thesis, University of Rennes, France, May 1993.

[10] O. Temam, E. Granston, and W. Jalby. To Copy or Not to Copy: A Compile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts. Technical Report 93-11, University of Leiden, 1993.

[11] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26(6), pages 30–44, June 1991.

[12] M. J. Wolfe. Iteration Space Tiling for Memory Hierarchies. In *Proceedings of the 3rd SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, 1987.

[13] M. J. Wolfe. More Iteration Space Tiling. In *Supercomputing '89*, 1989.

[14] S. Zhiyu, Z. Li, and P.-C. Yew. An Empirical Study on Array Subscripts and and Data Dependendencies. Technical Report 840, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Aug. 1989.