Impact of cache interferences on usual numerical dense loop nests[*]

O. Temam
University of Leiden
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

C. Fricker
INRIA
Domaine de Voluceau
78153 Le Chesnay Cedex
France

W. Jalby
University of Versailles
MASI
78000 Versailles
France

Abstract

In numerical codes, the regular interleaved accesses that occur within do-loop nests induce cache interference phenomena that can severely degrade program performance. Cache interferences can significantly increase the volume of memory traffic and the amount of communication in uniprocessors and multiprocessors. In this paper, we identify cache interference phenomena, determine their causes and the conditions under which they occur. Based on these results, we derive a methodology for computing an analytical expression of cache misses for most classic loop nests, which can be used for precise performance analysis and prediction. We show that cache performance is unstable, because some unexpected parameters such as arrays base address can play a significant role in interference phenomena. We also show that the impact of cache interferences can be so high, that the benefits of current data locality optimization techniques can be partially, if not totally, eradicated.

Keywords: memory reference patterns, software optimization, data locality, numerical codes, modeling cache interferences, performance analysis, performance prediction.

1 Introduction

As CPU cycle time decreases, main memory and network latencies rapidly increase and cache misses become very costly. Furthermore, the increasing issue rate of processors worsen the burden on caches. Moreover, most CPU chips are now being designed for integration into a massively parallel supercomputer or a parallel workstation, and therefore minimizing memory traffic, i.e. optimizing memory hierarchy utilization, is becoming critical. For all these reasons, optimizing the cache behavior has become a major issue. To achieve such optimizations, many studies have been performed to understand the workings of cache memories and derive proper optimizations.

The first category of studies [?] relies on numerous simulations of *representative codes*, i.e. a collection of codes which corresponds to the average workload of a computer. Such simulations provide a good summary of average cache performance and some hints at the relationships between the different cache parameters (cache size, line size, set-associativity). Furthermore they provide nearly exact indications on the behavior of cache memories for specific examples. A major problem inherent to such techniques is to find codes which are truly representative in terms of memory referencing. A second category of studies [?] aims at building analytical models for synthesizing the behavior of cache under most circumstances. Such models provide better insight on the relationship between the different parameters. These models can also be used for performance prediction,

[*]This work was funded by the BRA Esprit III European Project APPARC, European Agency DGXIII.

avoiding numerous simulations. However, while such analytical models are more representative than simulation based studies, they are generally less accurate. And they are intrinsicly limited because they cannot and are not designed for understanding specific phenomena which occur within a cache. Modeling the global behavior of cache may be sufficient as far as trends are needed. However, for understanding the weaknesses of caches and deriving either software or hardware optimizations more precise modeling is necessary.

Therefore a good understanding of a program reference pattern needs to be extracted. Some models have addressed this problem [**?**]. Although they are valuable tools, they still lack accuracy because they aim at characterizing the behavior of *most* programs. Therefore, they again sacrifice accuracy for generality and representativity.

However, there is a category of codes, numerical codes, that are emerging as some of the most demanding programs in terms of execution time and memory usage. Many architectures are targeted or at least tuned for such programs. The widespread use of numerical codes as benchmarks is a clear sign of their growing influence. Therefore, studying the cache behavior under numerical workloads is critical. However, numerical codes have specific properties in terms of memory addressing (spatial and temporal locality) which hardly allow them to fit in the classic framework of general models.

Fricker and al. [**?**] developed a model for direct-mapped and set-associative caches that is dedicated to regular (and some irregular) numerical codes. This model takes into account the fact that references within numerical codes correspond to chunks of consecutive addresses which recur periodically. The main asset of the model is to show the behavior of cache under numerical workloads, and to allow dimensioning of cache parameters for such codes. So, if this effort allows a better understanding of cache behavior under such workloads, it does not help in unveiling hot-spot and irregular phenomena which are specific to numerical codes and alter the cache performance. Furthermore numerical codes are actually made of a limited number of typical loop nests. Therefore, efforts should be concentrated on modeling and understanding the *actual and most frequent* types of loop nests. This problem is twofold: the first step is identifying these *typical* cases and consequently restricting problem hypotheses. The second and main step is deriving a model which encompasses the majority of such cases.

Porterfield [**?**] developed a model dedicated to numerical codes, which does examine specific pieces of codes and determine their behavior on cache. However, mostly fully-associative caches have been considered. Because such caches are unlikely to experience interference phenomena, conclusions can hardly be derived for interferences in *real* caches.

An important step towards accurate evaluation of cache interferences has been made in [**?**], where blocked Matrix-Matrix multiply is carefully studied. Cross and self-interference misses are evaluated, and a model for this algorithm is provided. This paper clearly unveils that interferences can severely alter locality exploitation. However, the model still lacks accuracy and is not capable of catching some of the specific phenomena and performance fluctuations induced by the mapping of direct-mapped caches. Furthermore, new parameters (such as *arrays base address*) which can play an important role in interference phenomena are not taken into account. Moreover, this model is dedicated to one particular example, while a methodology suitable to many numerical algorithms would be very useful.

Indeed, many powerful software optimization techniques for exploiting numerical codes locality have now been designed [**?**, **?**, **?**]. Although they stress the possible impact of cache interferences, no real evaluation of these phenomena nor a study of their frequency of occurrence have been performed yet. In the next sections, we will show that cache interferences can have a strong impact on performance and occur frequently.

Ferrante and al. [**?**] propose a realistic approach to locality optimization by evaluating the number of cache lines used (as opposed to the number of elements as in most other methods). It is mentioned that evaluating cache conflicts would help refining even more such *realistic* locality optimization techniques. In one example, it is briefly shown how to detect self-interferences.

To address the issues related to cache interferences, a model named *NUMODE* (NUmerical MODEl) [**?**] is being developed within the *APPARC*[1] project. The goal of NUMODE is to provide a framework for modeling cache interferences of a given loop nest, and then deriving an analytical expression for the number of cache misses. Therefore, NUMODE is both a model and a methodology. It is no doubt that such a model cannot be as flexible as others for global performance predictions, or for extracting global trends on cache behavior. However, it is possible to precisely quantify the interference phenomena that occur in real caches, determine their causes and conditions of occurrence. Many such phenomena can be identified. Moreover, the cache performance is shown to be actually unstable in many situations. It namely appears that software optimization techniques lack accuracy and can consequently lose their efficiency. It is also shown that the number of additional memory references due to cache interferences can be obtained as an analytical function of problem parameters. This result allows a precise analysis of the behavior of most classic algorithms on caches, and can then be used to design new optimization techniques or tune existing ones. Making such a function available to compilers can be profitable to code restructuring techniques. *NUMODE* is currently under development and will be implemented to perform extensive testings of its scope and accuracy.

In section **??** the problem is defined and hypotheses are given. In section **??**, the general method for computing the number of misses is indicated. In section **??**, conclusions are drawn and further work is discussed.

2 Problem statement

The purpose of the paper is twofold. The first goal is to show that cache interferences are not infrequent, that they can have a significant impact on numerical loop nests performance, and that their conditions of occurrence can be determined even though cache interferences are highly irregular. Understanding and then eliminating such interferences would allow stable cache performance. Furthermore, cache line size is kept small because large line sizes are generally considered to bring more interferences. This notion is not completely true: when line size is large, compulsory misses are much less important so that interference misses correspond to a larger portion of total misses. Therefore, numerical loop nests are more *sensitive* to cache interferences when line size is large, but such interferences are not necessarily more important. It derives that exploiting a larger line size requires a good understanding of cache interferences. The second and main goal of this paper is to introduce a method for estimating these cache interferences. General principles and main steps of the technique are indicated and illustrated with examples.

Cache architecture Direct-mapped caches have been chosen for several reasons:

- Direct-mapped caches are more sensitive to interferences than w-way associative caches. Therefore, they are more likely to benefit from studies and optimizations on that matter.

[1]APPARC is a BRA Esprit III European project

- Since the replacement policy of direct-mapped caches is straightforward, computing interferences is easier in direct-mapped caches. Though, we strongly believe the technique can be extended to w-way associative caches with moderate modifications.

- Among the three newest processor chips (DEC Alpha, MIPS R4000, SuperSPARC), two chips (DEC Alpha, MIPS R4000) include a small (8kbytes) direct-mapped on-chip data cache. Since the frequency of such processors is very high, the cost of a cache miss is huge, making it critical to reduce the amount of interferences.

- The placement policy in the DEC Alpha, for example, is such that, a data cache location can generally be determined from the data virtual address. Therefore, a study based on virtual addresses would accurately describe real cache behavior.

In the remainder of the paper, the cache size is indicated by C_S and the line size by L_S. The unit size is 8 bytes, i.e. the size of a double-precision floating point data. In all experiments, cache size is equal to 8-kbyte and line size is equal to 32-byte (the characteristics of the DEC Alpha data cache), so $C_S = 1024$ and $L_S = 4$.

Codes Let us now discuss which types of codes are considered. In numerical codes most data traffic occurs in do-loops, only these code constructs are examined. Only array references are considered because it is probable that other variables would be stored in registers if they are frequently used, and otherwise they would induce minimal perturbations of cache behavior.

Loop Nests A loop nest is composed of n distinct loops, j_i being the loop index of the i^{th} loop, and j_n being the loop index of the innermost loop. Column-major storage is assumed, so, for example, the virtual address of array reference $A(j_1, j_2)$ is $a_0 + Nj_1 + j_2$ where N is the leading dimension of array A and of the starting address of array A (cf figure **??**).

3 Modeling numerical codes behavior

3.1 Restrictive hypotheses

```
DO $j_1 = 0, N_1 - 1$
  DO $j_2 = 0, N_2 - 1$
    ⋮
      DO $j_n = 0, N_n - 1$
        ⋮
        $A(\alpha_1^A j_{k_1} + \beta_1^A, \ldots, \alpha_p^A j_{k_p} + \beta_p^A)$
        $B(\alpha_1^B j_{k_1} + \beta_1^B, \ldots, \alpha_p^B j_{k_p} + \beta_p^B)$
        ⋮
      ENDDO
    ⋮
  ENDDO
ENDDO
```

Figure 1: *An example of loop nest considered.*

A number of restrictions are imposed on the loop nests considered. First, the array subscripts must all be of the form $A(\alpha_1^A j_{k_1} + \beta_1^A, \ldots, \alpha_p^A j_{k_p} + \beta_p^A)$ where $(j_{k_i})_{1 \leq i \leq p}$ (with $p \leq n$) is any subset of loop indices, and $(\alpha_i^A)_{1 \leq i \leq p}$ and $(\beta_i^A)_{1 \leq i \leq p}$ are constants. For example, subscripts such as $A(j_1+j_2)$ are not considered. A close analysis of benchmark suites such as the *Perfect Club* [?] or *NAS* [?] and studies such as the one done by Yew and al. [?] show that such hypotheses encompass the subscripts found within most numerical loop nests. Furthermore, in dusty-deck codes, "irregular" subscripts often correspond to linearization, and therefore, in terms of memory reference patterns, are equivalent to those considered within the scope of the model. For all these reasons, these restrictions on array subscripts are considered to be reasonable constraints. Moreover, further developments of the present model may include more complex subscripts.

Another model hypothesis is that the boundaries of each loop index must be constant (after normalization, $0 \leq j_i \leq N_i$) and the stride of all indices equal to *1*. For any rectangular loop nest, the loop indices and array subscripts can be changed so as to satisfy these hypotheses. However, the fact non-rectangular loops do not fir these hypotheses. Since this point is relatively restrictive, further developments of the model will mainly focus on including this kind of loop nests.

3.2 General principles

If cache was fully-associative and replacement was optimal, cache misses would occur in two cases only. First, when data are loaded in cache for the first time; such misses are called *compulsory misses*. Second, when cache space is too small to store all loop nest data. Then, an element is flushed from cache each time a new element needs to be loaded; such misses are called *capacity misses*.

However, in direct-mapped caches (and in set-associative caches as well), cache misses can occur though cache space is sufficient, because a data element can only be mapped into one specific cache location (or w locations in w-way associative caches). Therefore, such cache misses do not occur because of *capacity* conflicts, but because of *mapping* conflicts; they are called *mapping misses* [?].

The main effect of such unexpected misses is to degrade the spatial and temporal reuse of data. Interferences can either correspond to interferences of an array with itself (*self-interferences*), or with another array (*cross-interferences*). In the remainder of the paper, these two types of interferences are analyzed.

For self-interferences the principle is to study the mapping of the set of elements of an array to be reused, and check whether these elements overlap with themselves. If so, self-interferences occur, and estimating the degree of overlapping yields the number of additional memory accesses brought by self-interferences. In general, self-interferences mostly correspond to temporal interferences.

For cross-interferences, once the set of elements of an array to be reused is identified (taking into account self-interferences), the overlapping between these elements and elements of another array is determined. Knowing the number of times the two sets of elements overlap, and the amount of overlapping each time, is sufficient to compute the number of additional memory accesses brought by cross-interferences. Cross-interferences can correspond to either temporal or spatial interferences.

3.3 Self-interferences

Theoretical reuse set Thanks to the subscript types considered, it is easy to identify where reuse due to self-dependences occurs. If coefficient $a_i = 0$ in the virtual address expression $a_0 + a_1 j_1 + \ldots + a_n j_n$ of a reference to array A, then loop i carries reuse. Let us define l as the lowest loop level where reuse occurs. On all loops k with $k > l$, no reuse occurs. So, on each iteration of these loops, the array elements referenced are all distinct. This set of elements is called the reuse set. During each execution of the sub loop nest $j_n, \ldots, j_{l+1}$, all elements of the reuse set are referenced.

Definition For array reference $a_0 + a_1 j_1 + \ldots + a_n j_n$, the reuse set can be defined if there exists a_k such that $a_k = 0$. The loop level of a reuse set is $l = \max\{k/a_k = 0\}$. The **theoretical reuse set** is equal to $RS(A)_l = \{a_0 + a_1 j_1 + \ldots + a_n j_n, (0 \leq j_i \leq N_i - 1)_{i>l}\}$.

Reuse can occur on different loop levels. However, reuse that is carried on loop levels higher than l is at least one order of magnitude less important than the reuse on loop l. For example, let us consider the following loop nest:

```
DO J1 = 0 , N1-1
  DO J2 = 0 , N2-1
    DO J3 = 0 , N3-1
      DO J4 = 0 , N4-1
                 .
                 .
          ...Y(J1,J3)...
                 .
                 .
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

For reference $Y(j_1, j_3)$, reuse occurs on loop levels 4 and 2; here $l = 4$. On one execution of loop j_4, the array element $Y(j_1, j_3)$ can be reused $N_4 - 1$ times (except for the first iteration of loop j_4). So, during execution of the whole loop nest, $N_1 \times N_2 \times N_3 \times (N_4 - 1)$ reuses of elements of Y can be achieved on loop j_4. On one execution of loop j_2, $Y(j_1, 0), \ldots, Y(j_1, N_3 - 1)$ can be reused $N_2 - 1$ times. So, during execution of the whole loop nest, $N_1 \times (N_2 - 1) \times N_3$ reuses can be achieved on j_2. Consequently, the potential reuse on loop j_4 is approximately N_4 times more important than the potential reuse on loop j_2. Furthermore, the time interval between two reuses on loop j_4 is minimum, i.e. it is equal to one iteration of loop j_4, while $N_4 \times N_3$ iterations of loop j_4 are executed between two reuses on loop j_2. Therefore, not only reuse is more scarce on loop j_2, but the probability it can be exploited is also much smaller.

That is why only the first reuse level is considered in general. However, when some coefficients consecutive to a_l (i.e. $a_{l-1}, a_{l-2}, \ldots$) are also equal to 0, reuse on these loop levels is considered to still be achieved (with respect to the reuse set, it is the same as if the boundary of loop l were $N_l \times N_{l-1} \times N_{l-2} \times \ldots$ instead of N_l). Note that if $a_l = a_{l-1} = a_{l-2} = \ldots = a_k = 0$, the reuse set on loop k is the same as on loop l.

Size of the theoretical reuse set Let us compute the number of cache lines corresponding to the theoretical reuse set assuming no self-interferences. First, let us determine the cache line stride of the reuse set of a reference A (where reuse occurs on loop l). It is equal to $\min(1, \frac{\alpha}{L_S})$, where $\alpha = \min_{l+1 \leq k \leq n}(a_k)$. This term is the ratio of the smallest coefficient (only considering loop levels defining the reuse set) to the line size. If this ratio is greater than 1, then it is necessarily equal to 1, since at most one new cache line is referenced on each iteration. For example, the access stride of reference $A(3j_3)$ is $\min(1, \frac{3}{L_S})$. Let us consider another example. The virtual address of reference $A(j_3, B * j_2)$ is $a_0 + Nj_3 + Bj_2$ where N is the leading dimension of A (here reuse occurs on loop level 1). Then the access stride of the reuse set is $\min(1, \frac{\min(N,B)}{L_S})$. Then, the number of cache lines in the theoretical reuse set is equal to $N_n \times \ldots \times N_{l+1} \times$ *acces stride*.

Actual reuse set Because self-interferences occur, not all elements of the reuse set can actually be reused. In direct-mapped caches, as soon as two elements of the reuse set compete for the same cache line, none of the two elements can be reused: they are victim of *self-interferences*. The elements of the reuse set not victim of self-interferences belong the *actual reuse set. The* **actual reuse set** *is the set of cache lines of the theoretical reuse set where no self-interferences occur.* Characterizing self-interferences is equivalent to determining the actual reuse set. And determining the actual reuse set is equivalent to studying the mapping of the theoretical reuse set in cache.

To compute the overlapping within the theoretical reuse set of a reference A, the loops n to $l+1$ are successively executed, starting with loop n. On each loop level k, the cache lines used by loops $n, \ldots, k+1$, which are not victim of interferences, form a temporary reuse set called $RS(A)_l^k$. On loop level k, the interferences between N_k such temporary reuse sets $RS(A)_l^{k+1}$ are determined, and the cache lines still not victim of interferences form the new temporary reuse set $RS(A)_l^k$.

1. Loop level n (reuse occurs on loop l): on this loop level, the reuse set is $\{a_0 + a_1 j_1 + \ldots a_n j_n, 0 \leq j_n < N_n\}$. Let $a_n^0 = a_0 + a_1 j_1 + \ldots + a_{n-1} j_{n-1}$. A temporary reuse set $RS(T)_l^n$ corresponds to $\min(1, \frac{a_n}{L_S}) \times N_n$ cache lines, starting at cache position a_n^0 *mod* C_S. If $C_S < \min(1, \frac{a_n}{L_S}) \times N_n$ then capacity interferences occur, and the victim cache lines are removed from the temporary reuse set (cf example on Matrix-Vector multiply below).

2. Loop level $n-1$: let $a_0^{n-1} = a_0 + a_1 j_1 + \ldots + a_{n-2} j_{n-2}$. The temporary reuse sets $RS(A)_l^n$ start at cache positions $a_0^{n-1} + a_{n-1} j_{n-1}$ *mod* C_S. By checking whether two such temporary reuse sets interfere (depending on their relative cache positions) and evaluating the amount of overlapping, it is possible to compute the number of cache lines not victim of interferences. These cache lines correspond to reuse set $RS(A)_l^{n-1}$ (cf example on Matrix-Matrix multiply below).

3. All subsequent steps are identical to step 2. The process stops on loop level l.

4. For each iteration of loop l, the number of cache lines of the theoretical reuse set victim of self-interferences is equal to *Number of cache lines of the theoretical reuse set - Number of cache lines of the actual reuse set.* So the number of additional memory requests per iteration of loop l is equal to *Number of cache lines of the theoretical reuse set - Number of cache lines of the actual reuse set.* The total number of additional memory requests is equal to $N_1 \times \ldots \times N_l \times$ (*Number of cache lines of the theoretical reuse set— Number of cache lines of the actual reuse set*).

This process is generally not too complex because few levels i have to be considered (reuse sets of dimension 1 or 2, i.e. depending on 1 or 2 loop levels, correspond to a majority of cases). Furthermore, very often the layout of sets in cache is straightforward due to the way array elements are referenced (stride one access to arrays), making it relatively easy to evaluate self-interferences, and compute the actual reuse set.

Matrix-Vector multiply Let us illustrate the previous notions with *matrix-vector multiply.* The do-loop nest is the following

```
DO J1 = 0 , N-1
  DO J2 = 0 , N-1
    Y(J1) += A(J1,J2) * X(J2)
  ENDDO
ENDDO
```

The linear function corresponding to each array reference is the following

Y: $0j_2 + j_1 + y_0$

X: $j_2 + 0j_1 + x_0$

A: $j_2 + Nj_1 + a_0$ (where N is the leading dimension of array A)

For array Y, only loop level 2 carries reuse, the theoretical reuse set is $RS(Y)_2 = \{Y(j_1)\}$. For array X, the theoretical reuse set is $RS(X)_1 = \{X(0), \ldots, X(N-1)\}$. No reuse occurs for array A.

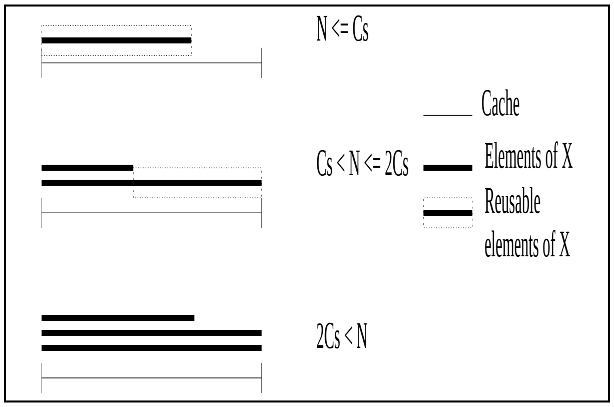

Figure 2: *Mapping of elements of X into cache.*

The theoretical reuse set of Y is equal to the actual reuse set of Y and corresponds to one cache line only; no self-interferences can occur. On the other hand the theoretical reuse set of X contains N consecutive elements (the access stride is 1), or $\frac{N}{L_S}$ cache lines. Therefore, if $N \geq C_S$, self-interferences occur. More precisely, if $C_S \leq N \leq 2C_S$, $2(N - C_S)$ elements of X are overlapped by other elements of X before they can be reused. Therefore, only the reuse of $2C_S - N$ elements of X can actually be exploited. So, the actual reuse set size is $\frac{2C_S - N}{L_S}$. $\frac{2(N-C_S)}{L_S}$ additional memory requests are necessary on each iteration of loop j_1 because of self-interferences of X. If $N > 2C_S$, no reuse occurs, the overlap between elements of X is total (cf figure **??**).

This problem is well known in the domain of loop restructuring. It corresponds to capacity interferences rather than mapping interferences. The most classic method for dealing with it is to block the loop, so that the reuse set of X is smaller than cache size. However, a less obvious and known fact is that, even when the loop is blocked interferences can still occur. Let us consider the blocked version of *Matrix-Matrix multiply.*

Blocked Matrix-Matrix multiply

```
DO J1 = 0 , N-1 , Bs
  DO J2 = 0 , N-1 , Bs
    DO J3 = 0 , N-1
      DO J4 = J1 , J1 + Bs-1
        DO J5 = J2 , J2 + Bs-1
          C(J3,J5) +=  A(J3,J4) * B(J4,J5)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

where B_S is the block size (for sake of simplicity it is assumed B_S divides N).

Matrix-Matrix multiply is generally blocked that way so as to keep in cache a submatrix $B_S \times B_S$ of matrix B. Let us normalize the loop indices so that they fit the model hypotheses (constant boundaries and stride *1*): j_5 is defined by $j_5 = J5 - J2$ and similarly $j_4 = J4 - J1$, $j_1 = \frac{J1}{B_S}$, $j_2 = \frac{J2}{B_S}$, and for all other i, $j_i = Ji$. Then, the linear function corresponding to array B is $j_5 + N_B j_4 + 0 j_3 + B_S j_2 + B_S N_B j_1 + b_0$, so its reuse set is $RS(B)_3 = \{j_5 + N_B j_4 + j_2 + N_B j_1 + b_0,\ 0 \leq j_5 \leq B_S - 1, 0 \leq j_4 \leq B_S - 1\}$, where N_B is the leading dimension of array B. Let us consider what happens in cache when this block is referenced. When j_5 varies between 0 and $B_S - 1$, an interval of B_S elements of B is mapped into consecutive cache locations, or $\frac{B_S}{L_S}$ cache lines. These cache lines correspond to the temporary reuse set $RS(B)_3^5$. Then, j_4 is increased by 1, and another interval of B_S elements is loaded into cache at a distance N_B from the previous interval. Due to the finite size of the cache, the actual distance in cache between two consecutive intervals of size B_S is $N_B \ mod \ C_S$ (where *mod* is the modulo operator) instead of N_B. Therefore depending on the value of $N_B \ mod \ C_S$, *successive intervals can overlap in cache*, thereby more or less degrading the potential reuse gained from blocking (all the cache lines of the subintervals of size B_S constitute the temporary reuse set $RS(B)_3^4$).

For example, if $0 \leq N_B \ mod \ C_S < B_S$, two consecutive intervals overlap by $B_S - (N_B \ mod \ C_S)$ cache locations or $\frac{B_S - (N_B \ mod \ C_S)}{L_S}$ cache lines (cf figure **??**). Since an interval has two neighbors, each interval overlaps by $2\frac{B_S - (N_B \ mod \ C_S)}{L_S}$ cache lines with its neighbors (except for the first and last interval which have only one neighbor and consequently overlap by $\frac{B_S - (N_B \ mod \ C_S)}{L_S}$ cache lines only). So, in each interval, only $\frac{B_S - 2(B_S - (N_B \ mod \ C_S))}{L_S}$ cache lines are not victim of self-interferences (except for the first and last interval, where $\frac{B_S - (B_S - N_B \ mod \ C_S)}{L_S}$ cache lines are not victim of self-interferences). Therefore, the actual reuse set only corresponds to $(B_S - 2) \times \frac{B_S - 2(B_S - (N_B \ mod \ C_S))}{L_S} + 2 \times \frac{B_S - (B_S - N_B \ mod \ C_S)}{L_S} = \frac{(B_S - 2) \times (2(N_B \ mod \ C_S) - B_S) + 2(N_B \ mod \ C_S)}{L_S}$ cache lines, while the theoretical reuse set (assuming no self-interferences) corresponds to $B_S \times \frac{B_S}{L_S}$ cache lines. Therefore the self-interferences of array B breed $\frac{B_S^2}{L_S} - \frac{(B_S - 2) \times (2(N_B \ mod \ C_S) - B_S) + 2(N_B \ mod \ C_S)}{L_S}$

additional memory requests per iteration of loop 3, and $N_1 \times N_2 \times N_3 \times \frac{(B_S-2)\times(2(N_B \ mod \ C_S)-B_S)+2(N_B \ mod \ C_S)}{L_S}$ additional memory requests in total.

Padding can be used for matrix B so that $B_S = N_B \ mod \ C_S$ (where N_B is the leading dimension of matrix B). In that case, placement of intervals in cache is optimum (cf figure **??**).

Conclusions For modeling purposes, such examples stress the fact that it is critical to properly evaluate mapping of array elements in cache To obtain accurate evaluation of self-interference misses (cf figure **??**). On a broader scope, it shows that precise analysis of interferences can be valuable for code optimizations such as loop restructuring since part or all the benefits of classic loop restructuring can be lost because of cache interferences. Moreover, such phenomena can appear frequently and with varying importance (in our example, depending on the parameter $N_B \ mod \ C_S$, interferences can be maximum when $N_B \ mod \ C_S = 0$ or minimum when $N_B \ mod \ C_S = B_S$).

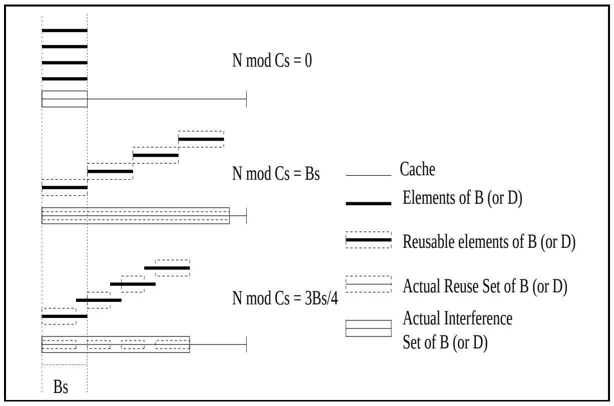
N mod Cs = 0
N mod Cs = Bs
N mod Cs = 3Bs/4
Bs
Cache
Elements of B (or D)
Reusable elements of B (or D)
Actual Reuse Set of B (or D)
Actual Interference Set of B (or D)

Figure 3: *Mapping of elements of B into cache.*

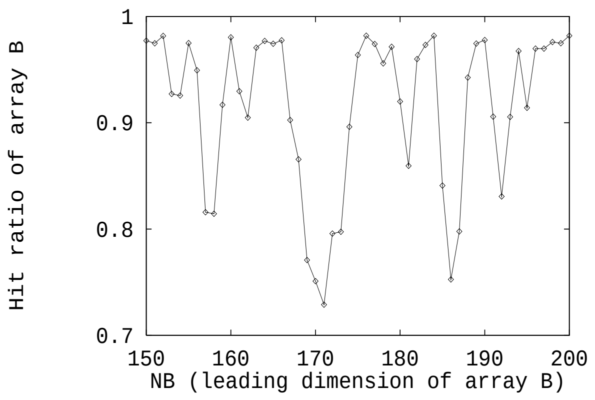

Figure 4: *Influence of parameter N_B on self-interferences.*

3.4 Cross-interferences

Two main cases of cross-interferences can occur between two references: either the difference between the corresponding two virtual addresses is constant (independent of loop indices), or it varies with the loop indices. These two cases must be distinguished because such cross-interferences are very different. In the first case, the two references are in translation, they always overlap and the amount of overlapping is constant, while in the second case the two references overlap only periodically and the amount of overlapping varies. So, detecting and estimating cross-interferences in the first case basically amounts to comparing the constant parameter of the two virtual addresses (it generally depends on arrays dimensions and base address), while in the second case, the relative movement of the two references must be analyzed.

The first type of interferences is called *internal cross-interferences* (because a set of references in translation constitutes a kind of *class* of references, and such cross-interferences then occur within a class), and the second type of interferences is called *external cross-interferences* (interferences among two references not belonging to the same class).

3.4.1 Estimating cross-interferences

Computing the impact of cross-interferences between two references amounts to estimating how much of the reuse of one reference is lost because of cross-interferences with the other reference. So, let us consider two references R^1, R^2, and compute the impact of R^2 on the reuse of R^1.

First, the set of elements R^1 can reuse must be estimated; it is the reuse set defined in section **??**. Second, the set of elements of R^2 that can interfere with R^1 must be estimated as well; this set is called the interference set. Because the reuse set is computed on a given loop level l, the interference set should be computed on the same loop level. Recall, that below loop l, no reuse can occur for R^1. Therefore, the number of additional memory requests for R^1 due to cross-interferences with R^2, on each reutilization of the reuse set (i.e. on each iteration of loop l), is exactly equal to the number of cache lines used by both the reuse set and the interference set. This notion is fundamental in the computation of cross-interferences. Computing interferences this way allows to make abstraction of time considerations, i.e. *when* interferences occur. It is sufficient to estimate the intersection between the set of cache lines corresponding to the reuse set and the interference set. It is important to note that, in the following sections, the reuse set considered is the actual reuse set, otherwise cross-interferences would be counted where self-interferences already occur, resulting in an overestimate of additional memory requests.

The interference set The definition of the theoretical interference set is the same as that of the theoretical reuse set.

Definition For array reference $a_0 + a_1 j_1 + \ldots + a_n j_n$, the **theoretical interference set**, on loop level l (this loop level is determined by the victim reuse set), is equal to $IS(A)_l = \{a_0 + a_1 j_1 + \ldots + a_n j_n, (0 \leq j_i \leq N_i - 1)_{i > l}\}$.

Moreover, determining the actual interference set is done much the same way as for the actual reuse set. The actual reuse set is the subset of cache lines of the theoretical reuse set where no self-interferences occur, while the **actual interference set** is simply the set of cache lines used by the theoretical interference set. So if a cache line of the theoretical interference set is victim of self-interferences, this cache line is still counted in the actual interference set (while such a cache line is rejected from the actual reuse set). Intuitively, the actual interference set corresponds to the cache surface (the number of cache lines) used by the theoretical interference set. The amount of cross-interferences is directly correlated to the size of the actual interference set (the larger the set, the higher the probability of overlapping with the reuse set). The process for determining the actual reuse set is the following:

1. Loop level n (for the reuse set, reuse occurs on loop l): on this loop level, the interference set is $\{a_0 + a_1 j_1 + \ldots a_n j_n, 0 \leq j_n < N_n\}$. Let $a_0^n = a_0 + a_1 j_1 + \ldots + a_{n-1} j_{n-1}$. A temporary interference set $IS(T)_l^n$ corresponds to $\min(1, \frac{a_n}{L_S}) \times N_n$ cache lines, starting at cache position a_0^n *mod* C_S. If $C_S < \min(1, \frac{a_n}{L_S}) \times N_n$ then capacity interferences occur, and the cache lines used twice or more are only counted once.

2. Loop level $n-1$: let $a_0^{n-1} = a_0 + a_1 j_1 + \ldots + a_{n-2} j_{n-2}$. The temporary interference sets $IS(A)_l^n$ starts at cache positions $a_0^{n-1} + a_{n-1} j_{n-1}$ *mod* C_S. So, by checking whether two such temporary interference sets interfere (depending on their relative cache positions) and evaluating the amount of overlapping, it is possible to compute the number of cache lines corresponding to the union of the two intervals. The union of the cache lines of all temporary interference sets $IS(A)_l^n$ is the temporary interference set $IS(A)_l^{n-1}$.

3. All subsequent steps are identical to step 2. The process stops on loop level l.

Let us consider the same example used for illustrating the computation of the actual reuse set, i.e. Matrix-Matrix multiply, except that reference $B(j_4, j_5)$ is replaced by $B(j_4, j_5) + D(j_4, j_5)$. Then, computing cross-interferences on B due to D implies computing the interference set of D on loop 3 (since reuse occurs on loop 3 for B). Then, as for array B, if $0 \leq N_D \ mod \ C_S < B_S$, each interval of size B_S of D overlaps by $\frac{2(B_S - (N_D \ mod \ C_S))}{L_S}$ cache lines with its two neighbor intervals (except for the first and the last interval). However, in opposition to the actual reuse set, these cache lines are still counted in the actual interference set. Because of this overlapping, the size of the actual interference set is $\frac{(B_S - 1) \times (N_D \ mod \ C_S) + B_S}{L_S}$ cache lines instead of $\frac{B_S^2}{L_S}$ (cf figure **??**).

In opposition to the reuse set, it is preferable that overlapping occurs within the interference set, because the larger the overlapping within a theoretical interference set, the smaller the corresponding actual interference set, and the less likely the actual interference set overlaps with cache lines of the actual reuse set. Therefore, the optimal case is obtained for $N_D \ mod \ C_S = 0$ (note that it corresponds to the worse case for the reuse set of D; cf section **??**).

3.4.2 Internal cross-interferences

Internal cross-interferences occur between two references in translation. Thanks to that property the relative cache position between the reuse set of the *victim* reference and the interference set of the *interfering* reference is always the same. Therefore, if the reuse set is defined on loop level l (reuse occurs on loop l), the total number of additional memory requests due to cross-interferences between the two references, is equal to the total number of iterations of loop l times the number of cache lines used by both the actual interference set and the actual reuse set.

The process for computing internal cross-interferences on a reference R^1 due to reference R^2 is the following:

1. Compute the actual reuse set of R^1; the reuse occurs on loop l. Compute the actual interference set of R^2 on loop l.

2. Compute the number of cache lines $CL(R^1, R^2)$ used by both the actual reuse set and the actual interference set. This is done by simply comparing the relative starting position of both sets.

3. The number of total additional memory requests due to internal cross-interferences between R^1 and R^2 is equal to $N_1 \times \ldots \times N_l \times CL(R^1, R^2)$.

Elementary linear algebra operation Let us consider the matrix operation $A = X \times {}^tY$ where A is an $N \times N$ matrix and X, Y are two vectors of dimension N. Often, vectors X, Y are linear combination of several vectors. For example, if $X = X_1 + X_2$, the operation is $A = (X_1 + X_2).\, {}^tY$. The corresponding loop is the following

```
DO J1 = 0 , N1-1
  DO J2 = 0 , N2-1
    A(J1,J2) = (X1(J2) + X2(J2))*Y(J1)
  ENDDO
ENDDO
```

We want to compute cross-interferences on X_1 due to X_2. The virtual addresses of $X_1(j_1), X_2(j_2)$ are $x_{1_0} + j_1$ and $x_{2_0} + j_2$. $x_{1_0} + j_2 - (x_{2_0} + j_2) = x_{1_0} - x_{2_0}$, the difference is constant so the two references are in translation. The cross-interferences are internal cross-interferences. For array X_1, reuse occurs on loop j_1 and the reuse set of X_1 corresponds to $\frac{N_2}{L_S}$ consecutive cache lines. On this loop level, the interference set of X_2 corresponds to $\frac{N_2}{L_S}$ consecutive cache lines. The cache distance between the two sets is equal to $x_{1_0} - x_{2_0} \ mod \ C_S$. If $x_{1_0} - x_{2_0} \ mod \ C_S \in [0, N_2 - 1] \bigcup [C_S - (N_2 - 1), C_S]$, the two sets overlap (cf figure **??**). In that case, the amount of overlapping is equal to $CL(X_1, X_2) = \frac{\min((x_{1_0} - x_{2_0} \ mod \ C_S), C_S - (x_{1_0} - x_{2_0} \ mod \ C_S))}{L_S}$ cache lines (cf figure **??**). Therefore, internal cross-interferences between X_1 and X_2 bring $CL(X_1, X_2)$ additional memory requests per iteration of loop j_1, or $N_1 \times \ldots \times N_l \times CL(X_1, X_2)$ additional memory requests in total (cf figure **??**).

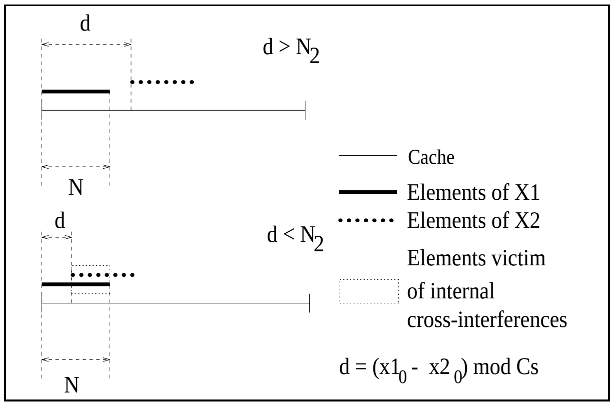

Figure 5: *Internal cross-interferences between X_1 and X_2.*

Spatial interferences Note that internal cross-interferences correspond to spatial interferences, only if the relative cache positions of the actual reuse set and the actual interference set is smaller than the line size (in the above example, $((x_{1_0} - x_{2_0} \ mod \ C_S) < L_S)$. So, spatial interferences have a low probability to occur. On the other hand, when such a case happens, little or no spatial reuse can be achieved and no temporal reuse can be achieved also. These cases are extremely costly, they correspond to "ping-pong", i.e. when two arrays translate in cache and constantly compete for the same cache location.

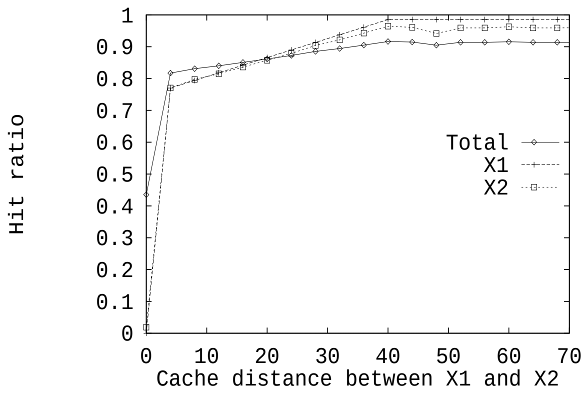

Figure 6: *Impact of internal cross-interferences on the hit ratio ($N_1 = 200, N_2 = 40$).*

Group-dependence reuse In this paper, mostly reuse due to self-dependences (a reference reuses itself) is analyzed. However, the reuse due to group-dependences (a reference reuses elements of another reference) can also be significant, though it is in general less important than reuse due to self-dependences. However, since the way internal cross-interferences perturbate group-dependence reuse is original, the phenomenon is worth being illustrated with an example.

Example from FLO52 Let us consider the following simple example extracted from a version of a *Perfect Club* code *FLO52* [**?**].

```
DO 10 J1=2,N1
  DO 10 J2=1,N2
    XY(J1,J2) = X(J1-1,J2,1) - X(J1,J2,1)
  ENDDO
ENDDO
```

Array XY is actually a temporary array used for scalar expansion. The leading dimension of arrays XY and X can be considered equal to N_2. In this case, there is no self-dependence on array X, but there is a group-dependence between references $X(j_1, j_2, 1)$ and $X(j_1 - 1, j_2, 1)$ which benefits to array reference $X(j_1 - 1, j_2, 1)$. Let us call R^1 the reference $X(j_1 - 1, j_2, 1)$ and R^2 the reference $X(j_1, j_2, 1)$. Since reuse occurs on a given loop level (loop 1), it is possible to extend the definition of the reuse set to group-dependences (except the reuse set is not reused by the reference itself). The reuse set of R^1 is $RS(X)_2^1 = \{X(j_1, 1, 1), \ldots, X(j_1, N_2, 1)\}$.

The dependence distance between the two references is equal to N_2. It is assumed $N_2 < C_S$ so that the reuse set of R^1 ($\frac{N_2}{L_S}$ cache lines) fits in cache. Now, array XY can have internal cross-interferences with reference R^1. However, the way such cross-interferences occur is not straightforward. The amount of overlapping is not equal to the number of cache lines used by both the reuse set of R^1 and the interference set of XY. Indeed, internal cross-interferences occur in a *boolean* way. Depending on the cache distance $x_0 - xy_0 \ mod \ C_S$ between the interference set of XY and the reuse set of R^1 two cases can occur. Either $x_0 - xy_0 \ mod \ C_S \in [C_S - (N_2 - 1), C_S]$ and the interference set of XY flushes *no element* of the reuse set of R^1 because a given cache line is used by R^1 *after* it has been used by XY (cf figures **??** and **??**). There is no additional memory request. Or $x_0 - xy_0 \ mod \ C_S \in [0, N_2 - 1]$ and the interference set of XY flushes *all elements* of the reuse set of R^1 because a given cache line is used by R^1 *before* it is used by XY. Therefore, elements referenced by $X(j_1, j_2, 1)$ are flushed before they can be reused by $X(j_1 - 1, j_2, 1)$ (cf figures **??** and **??**; note also that for $x_0 - xy_0 \ mod \ C_S = 0$ and $x_0 - xy_0 \ mod \ C_S = N_2$, reference XY induces ping-pong phenomenon with respectively reference $X(j_1, j_2, 1)$ and reference $X(j_1 - 1, j_2, 1)$). The total number of additional memory requests is equal to the number of cache lines of the reuse set of $X(j_1, j_2, 1)$ that would have been reused by $X(j_1 - 1, j_2, 1)$. Since $N_2 - 1$ array elements would have been reused, the number of additional memory requests per iteration of loop 1 is $\frac{N_2 - 1}{L_S}$, and the total number of additional memory requests due to internal cross-interferences is equal to $(N_1 - 1) \times \frac{N_2 - 1}{L_S}$.

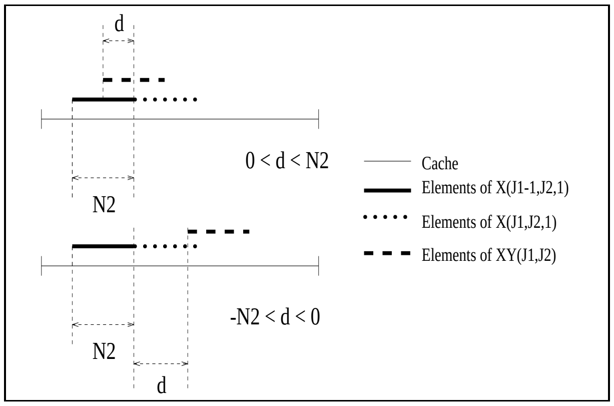

Figure 7: *Perturbation of group-dependence reuse of array X.*

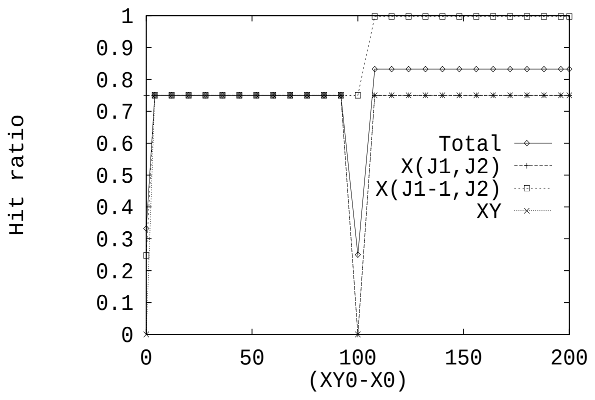

Figure 8: *Influence of initial positions of X and XY on the total hit ratio; $N_1 = N_2 = 100$.*

There again, by simply changing a base address (the base address of array XY), it is possible to eliminate cache interferences.

Conclusions Internal cross-interferences can be very significant because they occur on each reutilization of the reuse set. They are as frequent and can be as important as self-interferences. The above examples illustrate the fact that cross-interferences can vary significantly, though apparently randomly. Considering arrays base address is critical for detecting and estimating internal cross-interferences.

3.4.3 External cross-interferences

If two references R^1, R^2 are not in translation, the cross-interferences on R^1 due to R^2 are called external cross-interferences. Computing such interferences amounts to estimating the relative position of R^1 and R^2. Each time the reuse set of R^1 and the interference set of R^2 overlap, the amount of overlapping is estimated (the time unit here is one iteration of the loop where reuse occurs for R^1, i.e. loop l).

The virtual addresses of references of R^1, R^2 are $r_0^1 + r_1^1 j_1 + \ldots + r_n^1 j_n$ and $r_0^2 + r_1^2 j_1 + \ldots + r_n^2 j_n$. The positions of the interference set and the reuse set are determined by loop indices j_i with $i \leq l$ (loop indices j_i with $i > l$ determine the sets themselves). Therefore, the relative address distance of the two sets is $(r_0^1 - r_0^2) + (r_1^1 - r_1^2)j_1 + \ldots + (r_l^1 - r_l^2)j_l$. Let $r_i = r_i^1 - r_i^2$ for $i > 0$, $r_0 = r_0^1 - r_0^2 \; mod \; C_S$, and $D = \; gcd \;_{1 \leq i \leq l}(r^i)$. Then for any value of $(j_i)_{1 \leq i \leq l}$, there exists $\lambda \in \mathbf{Z}$ such that $r_0 + r_1 j_1 + \ldots + r_l j_l = r_0 + \lambda D$. Let $d = \; gcd \; (D, C_S)$. The possible relative cache positions of the two references are necessarily of the form $r_0 + \lambda d \; mod \; C_S$. Therefore, $RS(R^1)_l$ and $IS(R^2)_l$ have only $\frac{C_S}{d}$ possible relative cache positions. Approximately on each iteration of reuse loop l, a new relative position is reached. Therefore, after $\frac{C_S}{d}$ iterations of loop l, all possible relative positions have been reached. Consequently, the relative "movement" of the two references is periodic, of period $\frac{C_S}{d}$. The total number of iterations of loop l executed is $N_1 \times \ldots \times N_l$, so there are $\frac{N_1 \times \ldots \times N_l}{\frac{C_S}{d}}$ periods.

Since the period is $\frac{C_S}{d}$, for any interval of $\frac{C_S}{d}$ values of λ, $r_0 + \lambda d \; mod \; C_S$ describes all possible relative cache positions. Let I_λ be one such interval. Then for any value of λ, the number of cache lines used by both the reuse set and the interference set, i.e. $CL(R^1, R^2; \lambda)$, is computed the same way internal cross-interferences are evaluated, i.e. by computing the beginning and the end of each set and then deducing their overlap. Then, the total number of additional memory requests for one period is $\sum_{\lambda \in I_\lambda} CL(R^1, R^2; \lambda)$ and the total number of additional memory requests is $\frac{N_1 \times \ldots \times N_l}{\frac{C_S}{d}} \times \sum_{\lambda \in I_\lambda} CL(R^1, R^2; \lambda)$.

So the process for determining external cross-interferences on R^1 due to R^2 is the following

1. Determine the reuse set of R^1.

2. Determine d. The relative cache positions of the two sets are $r_0 + \lambda d \; mod \; C_S$. The period of the movement is $\frac{C_S}{d}$. Any interval I_λ of $\frac{C_S}{d}$ values of λ is picked. Compute $CL(R^1, R^2; \lambda)$ for any $\lambda \in I_\lambda$.

3. The total number of additional memory requests is equal to $\frac{N_1 \times \ldots \times N_l}{\frac{C_S}{d}} \times \sum_{\lambda \in I_\lambda} CL(R^1, R^2; \lambda)$.

Blocked Matrix-Vector multiply Let us consider the following blocked version of *Matrix-Vector multiply*

```
DO J1 = 0 , N-1 , Bs
  DO J2 = 0 , N-1
    DO J3 = J1 , J1 + Bs-1
      Y(J2) += A(J2,J3) * X(J3)
    ENDDO
  ENDDO
ENDDO
```

where B_S is assumed to divide N for simplicity's sake.

As for the example of blocked Matrix-Matrix multiply, new indices are defined: $j_3 = J3 - J1$, $j_1 = \frac{J1}{B_S}$ and $j_2 = J2$. The linear function corresponding to the subscripts of X and A are the following

X: $j_3 + 0j_2 + B_S j_1 + x_0$

A: $j_3 + Nj_2 + B_S j_1 + a_0$ (where N is the leading dimension of array A)

Let us consider how array A perturbates the actual reuse of array X in Matrix-Vector multiply. The reuse set $RS(X)_2$ of X is an interval of size $\frac{B_S}{L_S}$ cache lines (reuse occurs on loop 2). The corresponding interference set of A is an interval of $\frac{B_S}{L_S}$ cache lines also ($IS(A)_2 = \{0 + Nj_2 + B_S j_1 + a_0, \ldots, B_S - 1 + Nj_2 + B_S j_1 + a_0,\ \ 0 \leq j_2 \leq N - 1\}$). The relative position of the two sets change on each iteration of j_2. It is assumed $B_S < \frac{C_S}{2}$ so that $RS(X)_2$ and $IS(A)_2$ do not necessarily overlap in cache.

The relative positions of A and X are $j_3 + Nj_2 + B_S j_1 + a_0 - (j_3 + B_S j_1 + x_0) = a_0 - x_0 + Nj_2$. Let $r_0 = a_0 - x_0 \ mod\ C_S$. So $D = \ gcd\ (N) = N$ and $d = \ gcd\ (N, C_S)$, and there are $\frac{C_S}{d}$ relative cache positions.

Intuitively, studying the relative cache positions of the two sets means that the reuse set is considered to be fixed, while the interference set is considered to be moving in cache.

General case Let us pick an interval I_λ of $\frac{C_S}{d}$ values of λ such that $-\frac{C_S}{2} \leq r_0 + \lambda d \leq \frac{C_S}{2}$, i.e. $I_\lambda = \left[\frac{-\frac{C_S}{2} - r_0}{d}, \frac{\frac{C_S}{2} - r_0}{d}\right]$. Then, external cross-interferences occur when the beginning of the interference set, i.e. $r_0 + \lambda d$ is located within the reuse set, i.e. within $[0, B_S]$, or when the end of the interference set, i.e. $r_0 + \lambda d + B\ mod\ C_S$ is located within the reuse set, i.e. $[0, B_S]$. This can be expressed by the constraints $-B_S \leq r_0 + \lambda d \leq B_S$. Therefore, it is possible to define $I_\lambda(B_S)$, the subinterval of I_λ where overlapping occurs $I_\lambda(B_S) = \left[\frac{-B_S - r_0}{d}, \frac{B_S - r_0}{d}\right]$. For any value of $\lambda \in I_\lambda(B_S)$, the overlapping is $CL(X, A; \lambda) = \frac{|B_S - (r_0 + \lambda d)|}{L_S}$ cache lines, and the number of additional memory requests per period is $\sum_{\lambda \in I_\lambda(B_S)} CL(X, A; \lambda)$. The total number of additional memory requests is $\frac{N^2}{B_S \frac{C_S}{d}} \times \sum_{\lambda \in I_\lambda(B_S)} CL(X, A; \lambda)$.

Spatial interferences Estimating spatial interferences is done exactly the same way as for temporal interferences, except that spatial interferences occur only if $-L_S \leq r_0 + \lambda d \leq L_S$ instead of $-B_S \leq r_0 + \lambda d \leq B_S$ for temporal interferences. The number of additional memory requests due to spatial interferences is equal to $CL_s(X, A; \lambda) = \frac{B_S}{L_S} \times (|L_S - 1 - (r_0 + \lambda d)|)$ per value of λ (note that

$L_S - 1$ is used instead of L_S because the first reference to the line is considered to be a temporal reuse, not a spatial reuse). As above, an interval $I_\lambda(L_S)$ of values of λ where spatial interferences occur can be defined. Then, the number of additional memory requests per period is $\sum_{\lambda \in I_\lambda(L_S)} CL_s(X, A; \lambda)$. The total number of additional memory requests is $\frac{N^2}{B_S \frac{C_S}{d}} \times \left(\frac{B_S}{L_S} \times \sum_{\lambda \in I_\lambda(L_S)} CL_s(X, A; \lambda)\right)$.

Note that spatial interferences occur very few times (or none) over a period. So, though there are numerous additional memory requests for each value of λ where spatial interferences occur, there are few such values of λ so that the total number of spatial interferences is small in general.

Particular cases Depending on the values of r_0 and d many different situations can occur. If $B_S < d$, there are "holes" between the different possible cache positions of the interference set. One such hole has size $d - B_S$, so that if $d > 2B_S$, the reuse set can fit entirely within such a hole. In that case, no external cross-interferences would occur between the two arrays. Let us consider the case $d = C_S$, i.e. there is only one possible cache position for the interference set. Then if $r_0 \geq B_S$, no external cross-interference can occur, while if $r_0 < B_S$, external cross-interferences occur on each iteration of the reuse loop j_2 (cf figure **??**). In that case, $\frac{B_S - r_0}{L_S}$ cache lines overlap, so that a total of $\frac{N^2}{B_S} \times \frac{B_S - r_0}{L_S}$ additional memory requests are due to such external cross-interferences. The worse case corresponds to $r_0 = 0$, total overlapping occurs, not only temporal locality but also spatial locality cannot be exploited.

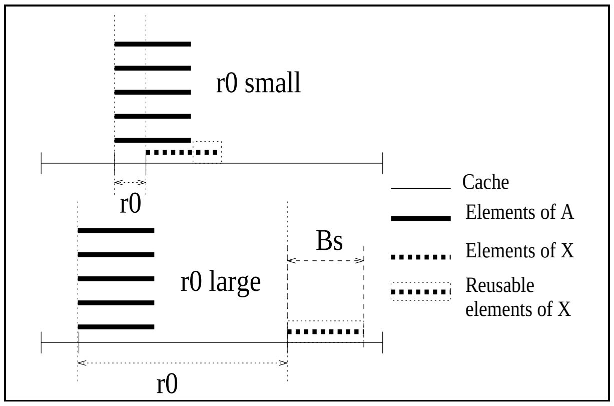

Figure 9: *Cross-interferences between A and X (N* mod $C_S = 0$*).*

Conclusions In opposition to internal cross-interferences, external cross-interferences occur periodically and with varying importance. Therefore detecting and estimating external cross-interferences is more difficult. Still, a precise estimate can be derived by studying such interferences over a period. It is possible to compute the period of interferences and the number of cross-interferences over a period. Then, the total number of external cross-interferences is equal to the number of periods times the external cross-interferences over one period. Though, external cross-interferences operate in a more irregular manner than internal cross-interferences, they can still be very damaging.

3.5 Interferences with multiple array references

The previous section provides a method for estimating how much an array can perturbate the potential reuse of another array. Now, when the reuse of an array is perturbated by several

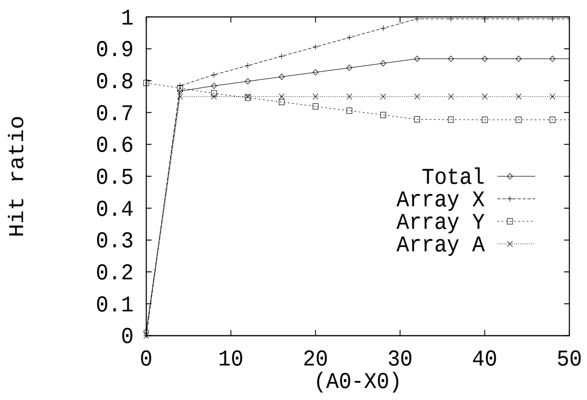

Figure 10: *Extreme case of influence of initial positions of A and X.*

different arrays it is necessary to evaluate whether the interferences caused by these several arrays are *cumulative* or *redundant*, so that total interferences can be evaluated. However, because it is unlikely three arrays are mapped to the same cache line within a short time interval, redundant interferences are relatively infrequent (this assertion is discussed in [**?**, **?**]). Therefore, considering multiple cross-interferences as cumulative is not an important approximation in general.

However, there are some cases where redundant interferences can be significant. If two arrays in translation overlap, then the size of the union of their two actual interference sets can always be much smaller than the sum of the size of their respective actual interference set. Therefore, such cases need to be considered when internal cross-interferences are evaluated. Indeed, it frequently happens that within the same loop nest, references are in translation (i.e. the difference between their virtual addresses is constant). If it is assumed all arrays in translation form a class, then the whole class can be considered together since the movement of all these references is the same. The actual interference set of a class is the union of the actual interference sets of all references within a class. External and internal cross-interferences due to arrays of one such class, should not be evaluated separately.

Translating arrays Let us now consider the following loop nest

```
DO J1 = 0 , N-1
  DO J2 = 0 , N-1
    C(J2) += A(J1,J2) - B(J1,J2)
  ENDDO
ENDDO
```

The linear mapping corresponding to the array subscripts is the following

A: $j_2 + Nj_1 + a_0$ (where N is the leading dimension of array A)

B: $j_2 + Nj_1 + b_0$ (where N is the leading dimension of array B).

The two array references $A(J1, J2)$ and $B(J1, J2)$ translate in cache; their relative cache distance is $r_0 = a_0 - b_0 \ mod \ C_S$. As for previous examples, the interference set of these two arrays

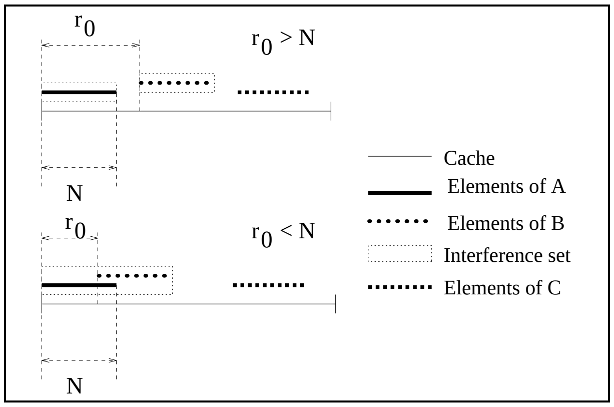

Figure 11: *Redundant interferences between two arrays in translation.*

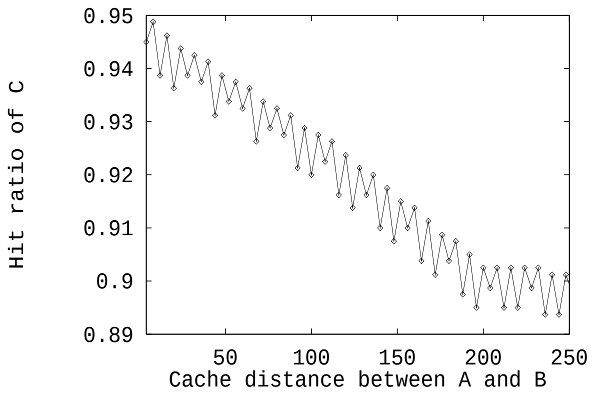

Figure 12: *Impact of redundancy of interferences on the hit ratio of array C.*

are intervals of size $\frac{N}{L_S}$. If $r_0 \geq N$ the two intervals never overlap and their respective effect on the reuse of C cannot be redundant. Now, if $0 \leq r_0 < N$, then $N - r_0$ elements of A and B have redundant impacts on the reuse of array C. So if array A actually brings $CL(C, A; j_1)$ additional memory requests, array B only brings $\frac{r_0}{N} \times CL(C, B; j_1)$ additional memory requests instead of $CL(C, B; j_1)$ (cf figures **??** and **??**). So, depending on the base address of arrays A and B the degree of cross-interferences on array X can vary by a factor of 2 (in figure **??** the miss ratio varies from 5% to 10%).

Conclusions In general, the redundancy between several external cross-interferences is small enough to be ignored. Note that self-interferences and internal cross-interferences are estimated first to avoid redundancy between classes of interferences. Only redundancy due to arrays in translation needs to be evaluated carefully.

4 Conclusions and Further work

The importance and frequency of occurrence of cache interferences are generally considered to be very irregular. These interference phenomena prevent optimum utilization of cache memory, and render cache performance unstable. The purpose of NUMODE is to understand, detect and quantify cache interferences. NUMODE is basically a framework and a method for computing the number of cache interferences within a given numerical loop nest. The different types of cache interferences are identified and separated into distinct classes. Then, for each class, it is shown how to evaluate the number of additional memory requests due to such interferences.

NUMODE can be used for determining a "glossary" of conflicting situations which can arise in numerical loop nests. Some of these situations have been illustrated in this paper. The occurrence and importance of these interferences have already been shown to be dependent on specific problem parameters, such as arrays base address or arrays dimensions.

Because analytical expressions of cache misses can be derived, this technique can be used for performance evaluation and prediction of a given algorithm. The influence of problem parameters on cache interferences can be extracted, and therefore, the algorithm can be optimized so as to minimize cache conflicts. In many examples, it has been shown that simple software optimization techniques, such as array padding or changing arrays base address, can help delivering optimum performance by minimizing or even eliminating cache interferences. Current software optimization techniques could strongly benefit from such enhancements.

The goal of NUMODE is different from usual cache models. Rather than providing informations on global cache behavior, the purpose of this model is to achieve a good comprehension of the phenomena happening in cache, spotting problems and unveiling their causes, so that software and hardware optimizations can be derived to reduce or eliminate cache conflicts.

Software optimization for cache interferences is one of our main research goals. First, using the algorithms presented in this paper, it is possible to systematically evaluate cache interferences at compile-time or at run-time. This information can be used to assist data locality optimizing algorithms [**?**, **?**, **?**] in the precise evaluation of the optimal block size. The theoretical values currently used have been shown to considerably lack precision [**?**]. Second, data locality optimizing algorithms are usually designed for local memories, i.e. they ignore the impact of cache conflicts. It is generally suggested [**?**, **?**, **?**] that copying can be used for eliminating cache conflicts. However, copying is a costly operation that cannot be used blindly [**?**]. The capacity to detect and estimate cache interferences allows to determine when copying is useful and apply it only then. Third, even when algorithms are not blocked, significant cache interferences occur. Reducing such conflicts by tuning problem parameters is another application. Fourth and last, though it seems possible to tune problem parameters for achieving efficient (deprived of interferences) execution of a given loop nest, it is still a challenge to either make such transformations transparent to other loop nests within the same program, or to find good tradeoff values for all code loop nests.

Acknowledgements

We would like to thank Elena Granston for her careful reviewing of the paper.

References

[1] A. J. Smith: *Cache memories*, ACM Computing Surveys, vol. 14, pp. 473-530, September 1982.

[2] A. Agarwal, M. Horowitz, J. Hennessy: *An analytical cache model*, ACM TOCS, Vol. 7, No. 2, May 1989, Pages 184-215.

[3] G.S. Rao: *Performance analysis of cache memories*, J. ACM 25, 3, 378-395.

[4] C. Fricker, P. Robert: *An analytical cache model*, INRIA Report No 1496, INRIA Rocquencourt, France, July 1991.

[5] A. K. Porterfield: *Software methods for improvement of cache performance on supercomputer applications*, PhD Thesis, CRPC-TR89009, May 1989.

[6] M. S. Lam, E. E. Rothberg, M. E. Wolf: *The cache performance and optimizations of blocked algorithms*, Proceedings of 4th ASPLOS, 1991.

[7] J. Ferrante, V. Sarkar and W. Thrash: *On estimating and enhancing cache effectiveness (extended abstract)*, Proc. of the 4th Workshop on Languages and Compilers for Parallel Computing, 1991.

[8] L. Pointer: *Perfect Club Report*, July 1989, CSRD Report No. 896.

[9] D. Bayley, J. Barton, T. Lasinsky and H. Simon: *The NAS parallel benchmarks*, Technical Report RNR-91-002, NASA AMes Research Center, August 1991.

[10] She, Zhiyu, Zhiyuan Li and P-C. Yew: *An empirical study on array Subscripts and Data Dependencies*, August 1989, CSRD Report No. 840.

[11] M. E. Wolf and M. Lam: *A Data Locality Optimizing Algorithm*, Proc. of PLDI.

[12] C. Eisenbeis, W. Jalby, D. Windheiser, F. Bodin: *A strategy for array management in local memory*, Advances in Languages and Compilers for Parallel Processing, MIT Press, 1991.

[13] K. Kennedy, K. McKinley: *Optimizing for parallelism and data locality*, Proceedings of ICS'92.

[14] O. Temam: *Study and optimization of numerical codes cache behavior*, PhD Thesis, Univeristy of Rennes, 1993.

[15] R.A.Sugumar and S.G.Abraham: *Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization*, to appear in the 1993 ACM SIGMETRICS Conference.

[16] C. Fricker, O. Temam, W. Jalby: *Accurate evaluation of blocked algorithms cache performance*, Technical Report, University of Leiden, The Netherlands, March 1993.