

Cache Interference Phenomena *

O. Temam
University of Leiden
2333 CA Leiden
The Netherlands

C. Fricker
INRIA
78153 Le Chesnay Cedex
France

W. Jalby
University of Versailles
78 Versailles
France

Abstract

The impact of cache interferences on program performance (particularly numerical codes, which heavily use the memory hierarchy) remains unknown. The general knowledge is that cache interferences are highly irregular, in terms of occurrence and intensity. In this paper, the different types of cache interferences that can occur in numerical loop nests are identified. An analytical method is developed for detecting the occurrence of interferences and, more important, for computing the number of cache misses due to interferences. Simulations and experiments on real machines show that the model is generally accurate and that most interference phenomena are captured. Experiments also show that cache interferences can be intense and frequent. Certain parameters such as array base addresses or dimensions can have a strong impact on the occurrence of interferences. Modifying these parameters only can induce global execution time variations of 30% and more. Applications of these modeling techniques are numerous and range from performance evaluation and prediction to enhancement of data locality optimizations techniques.

Keywords: cache interferences or conflicts, numerical codes, data locality, performance evaluation, modeling.

1 Introduction

Three types of cache misses can be distinguished [6, 7]: *cold-start* misses which are compulsory, *capacity* misses and *interference* (or *conflict*) misses. Capacity misses occur when cache space is insufficient to store all data to be reused. Interference misses occur when two data are mapped to the same cache location. Typically, fully-associative caches do not exhibit interference misses.

*This work was supported by the Esprit Agency DG XIII under Grant No. APPARC 6634 BRA III.

Capacity misses in numerical codes have been studied to a great extent [2, 9, 16], and can be relatively easily predicted and estimated. Most of these studies also attempt to take into account the impact of the cache line size. But, there are few studies on interference misses, though several case-studies report that cache interferences can severely affect cache behavior [6, 8]. Also, in [3], suggestions are provided on how to consider interferences for cache performance evaluation and optimization.

Cache interferences are difficult to predict and estimate, because it is necessary to know *where* data are mapped in cache and *when* data are referenced. For instance, consider addresses A and B that are mapped to the same cache location and reused 3 times. Whether the reference sequence is $AAABBB$ or $ABABAB$, interference misses are equal to respectively 0 or 4.

Nevertheless, several reasons press for the study of cache interferences. First of all, cache tends to be a performance bottleneck because of high network and memory latencies, so that significant performance improvements can be obtained through a slight reduction of cache misses. Besides, several on-chip data caches are direct-mapped in order to achieve a low hit time. Such caches are considered to be more sensitive to interferences [8], especially when they are small, which is currently the case because of on-chip space constraints (8 kbytes in the DEC Alpha [11], MIPS R4000 [5]). Moreover, large cache line sizes induce high interferences [12], which is one of the reasons why cache line size is currently kept small. Being capable to detect and avoid cache interferences could allow further increases of the cache line size. Most important of all, cache interferences make program performance unpredictable. Understanding the workings of cache phenomena would allow precise performance analysis and prediction of application codes.

This paper is particularly targeted towards numerical codes which are characterized by large working sets, so that their performance is highly dependent on the memory hierarchy behavior. The goal of the paper is

threefold. First, to illustrate the fact that cache interferences can occur frequently and have a significant impact on program performance (section 2 and 3). The second goal is to introduce a methodology for *predicting and estimating cache interferences in direct-mapped caches for frequently occurring numerical loop nests* (section 3). The third goal is to illustrate the model accuracy through several examples (sections 3 and 4).

A first version of the framework for computing cache interferences has been presented in [14]. In [15], this model has been applied to drive data copying strategies. The present paper provides the details of the computations and illustrates the model accuracy.

Experiments For all examples, three types of statistics are provided: the simulated execution time (for a whole loop or for an array), the estimated execution time (obtained with the model) and the global execution time of the loop timed on an HP/PA-RISC workstation.¹ The execution time curves corresponding to simulations and modeling have been obtained with the following technique. Since the purpose of the model is to predict *variations* of rather than *absolute* execution time, a starting point (x_0, t_0) is picked from the *Real* execution time graph (where x is the parameter varied in the graph, and t is the execution time). Let $M(x)$ the number of simulated or estimated cache misses for parameter x , then $t = t_0 + (M(x) - M(x_0)) \times t_{lat}$.² The purpose of these statistics are twofold. First, to show the impact of cache interferences on global performance. Second, to verify that execution time variations correspond to miss ratio variations.

2 Notion of Cache Interferences

Cache interferences operate by disrupting the reuse of data. Several types of reuse can be distinguished: spatial and temporal reuse. And for each type, self-dependence reuse (one reference reuses its own data) and group-dependence reuse (one reference uses the data of another reference) can also be distinguished (see [16]).

Characteristics of the HP/PA-RISC cache:
 Cache Size = $C_S = 256$ kbytes = 32768 array elements.
 Line Size = $L_S = 64$ bytes = 8 array elements.
 Direct-mapped cache.
 1 word = 4 bytes.
 For the whole paper, 1 array element = 1 double-precision floating-point data = 2 words; all sizes and dimensions are given in array elements.
 For all experiments, 1 array element = 1 word.

¹For the experiments on the HP, the array base addresses have been varied by using one large single array and having other arrays point to that array. By modifying the pointed location, the relative base address between two arrays can be varied.

²For our HP system, the miss penalty has been experimentally timed at $8e - 7$ seconds.

```

I = JU-1
DO 10 J = 0, JU-1
  DO 20 K = 0, KU-1
    LDD = -LDA(K)*U(K, J+2) - LDB(K)*U(K, J+1)
    LDA(K) = LDB(K)
    LDB(K) = LDD
    LDS(K) = LDS(K) + LDD*U(K, J)
    U(K, I) = U(K, I) + LDD*U(K, J)
    F(K, I) = F(K, I) - LDD*F(K, J)
  20 CONTINUE
  10 CONTINUE

```

Loop 2a (I=JU-1; KU=700; JU=500).

Execution time for 100 runs (seconds)

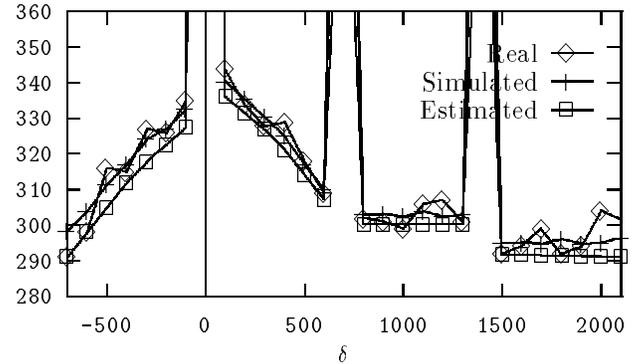


Figure 2b: Execution time.

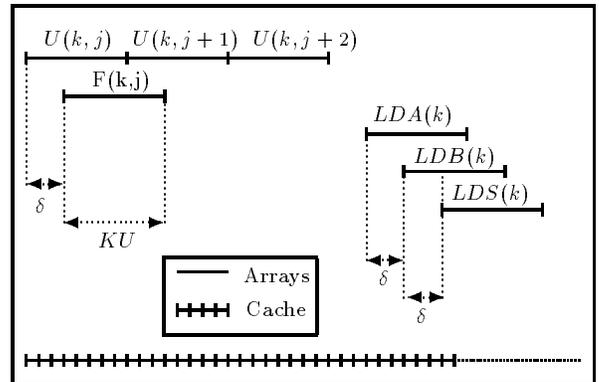


Figure 2c: Layout in cache.

Figure 2: Influence of Cache Interferences.

Frequency of interferences The notion of cache interferences conveys rare and intense phenomena like "ping-pong", where two references (such as references $A(I)$ and $B(I)$ in a Do-Loop with index I) start at the same cache location, translate in cache and constantly compete for the same cache line, therefore preventing spatial reuse.

However cache interferences are much more diverse and frequent, because any type of reuse can be disrupted. In general, *the larger the reuse distance for a given datum, the higher the probability it is flushed from cache, because the more data are loaded before reuse occurs*. Consider loop 2a which is a modified version of a

loop in **ARC2D**, a Perfect Club benchmark [1].

The reuse distance associated with the spatial reuse of $LDA(k)$ is equal to 1 iteration of loop k (unlikely to be disrupted), while the reuse distance associated with the temporal reuse of $LDA(k)$ is equal to KU iterations of loop k (more likely to be disrupted, depending on the value of KU).

Irregularity of interferences Whether two data sets intersect in cache is determined by the cache position of these sets. This position generally depends on two types of parameters: the arrays *base address* and *leading dimensions*. Since these parameters are *arbitrarily* (i.e., not "cache-consciously") determined by the compiler or the programmer, cache interferences occur in an apparently "random" manner. Consider loop 2a.³ If the cache distance between arrays LDA and LDB is smaller than N_2 (i.e., if $|(lda_0 \bmod C_S) - (ldb_0 \bmod C_S)| < KU$, where lda_0, ldb_0 are the base addresses of arrays LDA and LDB), then the two data sets intersect. Assuming a direct-mapped cache, data belonging to the intersection cannot be reused.

Importance of cache interferences In loop 2a, the distance between all arrays is characterized by one parameter, so that most effects can be shown by varying that single parameter: $(ldb_0 - lda_0) \bmod C_S = (lds_0 - ldb_0) \bmod C_S = (u_0 - f_0) = \delta$.⁴ In figure 2b, δ is varied from $-KU$ to $4 \times KU$. Performance variations of 20% (500% in the case of ping-pong) can be observed. The match between the estimated and real execution time graphs strongly suggests that these variations are due to interference phenomena. Note that for each value of δ , the exact same number of references are performed, only the arrays base address is modified. Other experiments conducted in this paper (see sections 3.5, 3.6.1, 3.6.2, 3.7), show frequent and significant performance losses due to cache interferences. Each of these examples illustrate a type of interference phenomenon, and for each case, it is shown in section 3 how to evaluate the corresponding number of cache misses. Section 4 synthesizes these results by showing how to compute the number of cache misses of loop 2a.

3 Modeling Cache Behavior

In this section, the techniques used to estimate the number of interference misses are presented. Though the method applies to all types of reuse, the method is illustrated with self-dependence temporal reuse, because it is the most frequent type of reuse and because it is often the most likely to be victim of interferences. The extensions to group-dependence reuse is explained.

³This example is analyzed in detail in section 4.

⁴Figure 2c illustrates the respective cache positions of the different arrays for one iteration of loop J.

Because of paper length constraints, the extension to spatial reuse is not discussed in this paper (see [13]).

Though it was not our primary goal, the methodology can be applied to compute *capacity misses* as well as *interference misses* (though interference misses are far more complex to evaluate). Consequently, the analytical expression of the total miss ratio for a given loop can be derived, as illustrated in section 4 (compulsory misses can be easily estimated).

3.1 Basic Concepts

Evaluating the miss ratio of a loop nest amounts to *counting, for each reference, the number of cache misses due to disruption of locality exploitation*. So, for each reference, it is necessary to determine *when reuse occurs* (i.e., the loop level where reuse occurs for the first time). Consider loop 3.5a. The reuse of reference $Y(j_3, j_2)$ occurs on loop j_1 and the reuse of reference $Z(j_3, j_1)$ occurs on loop j_2 .

This "reuse" loop defines *the set of data to be reused*. For reference $Y(j_3, j_2)$, this set is equal to $\{Y(0, 0), \dots, Y(N-1, N-1)\}$ while it is equal to $\{Z(0, j_1), \dots, Z(N-1, j_1)\}$ for reference $Z(j_3, j_1)$.

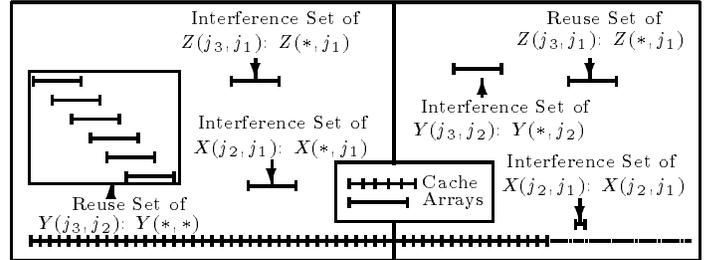


Figure 3.1: *Notion of Interference/Reuse Sets.*

Similarly, this loop level also defines *the sets of data of all other references that can interfere with the "reuse" set*. For reference $Y(j_3, j_2)$, the "interference" sets are $\{Z(0, j_1), \dots, Z(N-1, j_1)\}$ and $\{X(0, j_1), \dots, X(N-1, j_1)\}$ (Z and X can flush elements of Y from cache before they are reused), see figure 3.1. For reference $Z(j_1, j_3)$, the "interference" sets are $\{Y(0, j_2), \dots, Y(N-1, j_2)\}$ and $\{X(j_2, j_1)\}$.

Consider any reference. On each iteration of the reuse loop, *the number of cache misses due to the disruption of reuse for this reference is equal to the size of the intersection (in cache lines) between the "reuse" set and the "interference" sets*.

Considering the problem this way, implicitly *makes abstraction of time considerations*, i.e., when interferences occur. *The problem is then equivalent to computing the intersection size between several sets*. Summing these intersections over all iterations of the reuse loop provides the total number of cache interference misses

for a given reference. The next sections provide a formal treatment of this method.

3.1.1 Reuse Set

Definition 3.1 Assuming loops j_1, \dots, j_n are perfectly nested (with j_n the innermost loop), the reuse loop level l of a reference $R = r_0 + r_1j_1 + \dots + r_nj_n$ is defined as $l = \max \{k \mid r_k = 0\}$.⁵

Example Consider loop 3.5a. The reuse loop level of $Z(j_3, j_1)$ is 2 (or loop j_2).

3.2 Reuse Set

Within a set of array elements to be reused, it is possible that two elements map to the same cache location. These elements then alternately flush each other from cache before they can be reused. Therefore, they should not be counted within the set of elements that can be effectively reused. It is necessary to distinguish between **theoretical reuse set** and **actual reuse set**.

Definition 3.2 For any array reference $R = r_0 + r_1j_1 + \dots + r_nj_n$, which reuse loop is l , the **theoretical reuse set** is equal to $\text{TRS}(R) = \{r_0 + r_1j_1 + \dots + r_nj_n, (0 \leq j_i \leq N_i - 1)_{i>l}\}$.

Definition 3.3 The **actual reuse set** $\text{ARS}(R)$ is the cache footprint of the theoretical reuse set $\text{TRS}(R)$, excluding all cache lines for which mapping conflicts occur.

Definition 3.4 $\|\text{TRS}(R)\|$ is the size of the theoretical reuse set of R expressed in "array" lines, i.e., the number of cache lines used by $\text{TRS}(R)$ assuming an infinite cache size.

Definition 3.5 $\|\text{ARS}(R)\|$ is the size of the actual reuse set of R expressed in cache lines.

Example Consider loop 3.5a. The reuse set of $Z(j_3, j_1)$ is equal to $\{z_0 + j_3 + Mj_1, (0 \leq j_3 \leq N - 1)\}$, where z_0 is the base address of array Z and M is the leading dimension of arrays X, Y, Z . In this case $\|\text{TRS}(R)\| = N$ array elements of $\frac{N}{L_S}$ array lines.⁶ The actual reuse set is equal to the cache lines corresponding to cache locations $\{(z_0 + j_3 + Mj_1) \bmod C_S, (0 \leq j_3 \leq N - 1)\}$. Assuming a cache size of 4 array elements (8 words) and an 1-element (2 words) cache line, the actual reuse set of Z is equal to N cache lines if $N \leq 4$, it is equal to $8 - N$ cache lines if $4 < N < 8$, and it is empty if $8 \leq N$.

⁵Note that only simple dependences are considered; for instance the reuse associated with reference $A(j_1 + j_2)$ is not considered. The most frequently found dependences are simple ones, as mentioned in [17].

⁶Floor and ceiling functions have often been omitted in this paper because experiments showed they generally don't have a significant impact on precision.

3.3 Interference Set

The definition of the set of array elements that can interfere with a reuse set is very similar to the definition of a reuse set.

Definition 3.6 For any array reference $R = r_0 + r_1j_1 + \dots + r_nj_n$, that can interfere with a reuse set defined on loop level l , the **theoretical interference set** is equal to $\text{TIS}(R) = \{r_0 + r_1j_1 + \dots + r_nj_n, (0 \leq j_i \leq N_i - 1)_{i>l}\}$.

In opposition to the actual reuse set, when two elements of a theoretical interference set map to the same cache line, this line is still counted in the actual interference set.

Definition 3.7 The **actual interference set** $\text{AIS}(R)$ is exactly the cache footprint of the theoretical interference set $\text{TIS}(R)$.

Definition 3.8 As for the reuse set, $\|\text{TIS}(R)\|$ and $\|\text{AIS}(R)\|$ denote respectively the size of the theoretical and actual interference set of R .

Example Consider loop 3.5a. The reuse of $Y(j_3, j_2)$ occurs on loop j_1 . So the corresponding interference set of $Z(j_3, j_1)$ is defined on loop j_1 . The interference set of $Z(j_3, j_1)$ is equal to $\{z_0 + j_3 + Mj_1, (0 \leq j_3 \leq N - 1)\}$. In this case $\|\text{TIS}(R)\| = N$ array elements of $\frac{N}{L_S}$ array lines. Assuming again a cache size of 4 array elements (8 words) and a 1-element (2 words) cache line, the actual interference set of Z is equal to N cache lines if $N < 4$, and it is equal to 4 cache lines if $4 \leq N$.

3.4 Evaluating the Actual Sets⁷

Consider a reference $R = r_0 + r_1j_1 + \dots + r_nj_n$, of which we want to compute the cache footprint on loop level l (i.e., the footprint is defined by loops $n, n - 1, \dots, l + 1$).

Observation 3.1 After iterating on any loop $i, l < i \leq n$, the data layout in cache is assumed to be a set of intervals of cache locations, characterized by the average size S of an interval, and the average distance σ between two consecutive intervals.

This assertion is an approximation which aims at simplifying the evaluation process. Using this observation, a recursive process can be applied for each loop level $i, l < i \leq n$.

Problem 3.1 Assuming an initial regular⁸ data layout of intervals of size S separated by a distance of σ cache locations (a layout obtained after iterating on loops $n, n - 1, \dots, i + 1$), the problem is to determine the final layout, i.e., average size S' , average distance σ' after iterating on loop i .

⁷This section can be skipped if in-depth comprehension of the computations is not sought for.

⁸i.e., the distance between any two consecutive intervals is approximately constant.

3.4.1 Actual interference set

For problem 3.1, a recursive process can also be applied. Let $\sigma_0 = C_S$ and $\sigma_1 = \sigma$. After a number of iterations, the intervals wrap around the cache area of size σ_0 . Within each area of σ_1 cache locations, the layout of intervals is identical. So, the study can be restricted to *only one such area*. Within one such area, the spacing between two consecutive intervals is equal to $\sigma_2 = \sigma_1 - \sigma_0 \bmod \sigma_1$. Let us consider a recursive application of this process.

Observation 3.2 *On recursion level k , the size of the area considered is equal to σ_{k-1} , and the spacing between two consecutive intervals in one such area is equal to σ_k . The recursion process stops on a level s when all iterations of loop i have been considered, or when overlapping occurs, i.e., if $\sigma_s < S$. Starting at this point, it is possible to determine the footprint within an area of size σ_{s-1} , and assuming the layout is the same across all areas, the footprint within the whole cache.*

Proposition 3.1 *Let $N = n_s \lfloor \frac{\sigma_0}{\sigma_{s-1}} \rfloor + r$ (with $r = N \bmod \lfloor \frac{\sigma_0}{\sigma_{s-1}} \rfloor$). In $\lfloor \frac{\sigma_0}{\sigma_{s-1}} \rfloor - r$ areas of size σ_{s-1} , n_s intervals of size S have been brought, and in the remaining r areas $n_s + 1$ intervals have been brought.*

Proposition 3.2 *Let $f_i(x, S, \sigma_s) = S + (x - 1)^+ \sigma_s$. Then the cache footprint size of all intervals is equal to $\max \left(\left(\lfloor \frac{\sigma_0}{\sigma_{s-1}} \rfloor - r \right) \times f_i(n_s, S, \sigma_s) + r \times f_i(n_s + 1, S, \sigma_s), \sigma_0 \right)$.*

For this layout of intervals, the average size of an interval is equal to

$$S' = \frac{\max \left(\left(\lfloor \frac{\sigma_0}{\sigma_{s-1}} \rfloor - r \right) \times f_i(n_s, S, \sigma_s) + r \times f_i(n_s + 1, S, \sigma_s), \sigma_0 \right)}{\frac{\sigma_0}{\sigma_{s-1}}}$$

and the spacing between intervals is equal to $\sigma' = \sigma_{s-1}$.

Remark $\sigma_k = \sigma_{k-1} - \sigma_{k-2} \bmod \sigma_{k-1}$, so that all σ_k can be computed *a priori* with the cost of one application of Euclidean algorithm [10]. In practice, this fact also makes the process *non-recursive*, since the level at which recursion stops can be determined beforehand.

After the process has been applied for all loops $i, l < i \leq n$, the resulting layout in cache is the actual interference set of the reference, defined on loop l .

3.4.2 Actual reuse set

The reasoning is nearly identical for the actual reuse set, except that cache lines where overlapping occurs must be excluded.

Proposition 3.3 *The number of elements that can be reused is equal to*

$$\max \left(\left(\frac{\sigma_0}{\sigma_{s-1}} - r \right) \times f_r(n_s, S, \sigma_s, \sigma_{s-1}) + r \times f_r(n_s + 1, S, \sigma_s, \sigma_{s-1}), \sigma_0 \right).$$

where

$$f_r(x, S, \sigma_s, \sigma_{s-1}) = \begin{cases} S & \text{if } x = 1. \\ S + 2 * \sigma_s + (x - 2)^+ (2\sigma_s - S)^+ & \text{if } f_i(x) < \sigma_{s-1}. \\ \max(0, S + 2 * \sigma_s + (x - 2)^+ (2\sigma_s - S)^+ \\ - (f_i(x) - \sigma_{s-1})) & \text{if } f_i(x) \geq \sigma_{s-1}. \end{cases}$$

An example of application of the whole process can be found in section 3.5.

Note that this process is inaccurate (as mentioned above, several approximations were made), so the error can become significant after multiple applications (i.e., for multiple loop levels). Fortunately, a reuse set defined over two loop levels corresponds to a 3-deep loop nest where reuse occurs on the third loop. For such a reuse set, the algorithm needs only to be applied once (except if the access stride is not equal to one). Reuse sets defined over three loops are less common in primitives and real codes [17], or are less likely to be exploited (for instance, loop blocking is rarely performed over more than the two innermost loops).

3.5 Self-Interferences

Assume the reuse loop level is l for reference R .

Proposition 3.4 *The number of cache misses due to self-interferences of R is equal to*

$$N_1 \times \dots \times N_l \times (||\text{TRS}(R)|| - ||\text{ARS}(R)||).$$

Intuitively, each cache line excluded from the actual reuse set because of conflicts generates one cache miss, each time the reuse set is referenced, hence the proposition.

Example Consider loop 3.5a, corresponding to the multiplication of two $N \times N$ submatrices of $M \times M$ matrices.

```
DO j1=0,N-1
  DO j2=0,N-1
    reg = X(j2,j1)
    DO j3=0,N-1
      Z(j3,j1) += reg * Y(j3,j2)
    ENDDO
  ENDDO
ENDDO
```

Loop 3.5a (N=100).

Execution time for 100 runs (seconds)

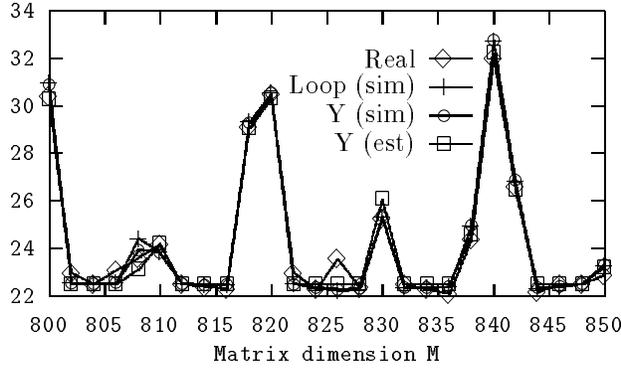


Figure 3.5b: Execution time.

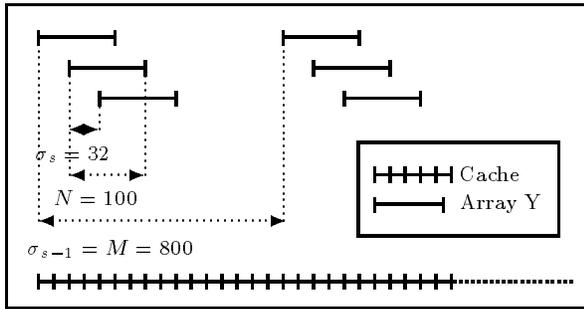


Figure 3.5c: Layout in cache.

Figure 3.5: Self-Interferences.

Since all arrays are reused, the amount of compulsory misses is negligible. Spatial interference misses for $Z(j_3, j_1)$ and $Y(j_3, j_2)$ are negligible because of the small spatial reuse distance; the spatial reuse distance for $X(j_2, j_1)$ is large but since this reference accounts for a small share of all memory accesses, the impact of spatial interference misses should not be significant. References $Z(j_3, j_1)$ and $Y(j_3, j_2)$ generate most memory accesses, but the temporal reuse distance for $Y(j_3, j_2)$ is one order of magnitude larger than for $Z(j_3, j_1)$. So, intuitively, $Y(j_3, j_2)$ is likely to be responsible for most temporal interference misses, so in this case, for most of the loop misses.

Since the theoretical interference set size of $Z(j_3, j_1)$ is equal to N elements and the theoretical reuse set size of $Y(j_3, j_2)$ is equal to N^2 elements, the effect of self-interferences can be much more significant than cross-interferences (i.e., cross-interferences can at most prevent the reuse of N elements, while self-interferences can prevent the reuse of all N^2 elements).

The reuse set of $Y(j_3, j_2)$ is a set of N intervals of size $S = N$ with a relative distance of $\sigma_1 = N \bmod C_S$

which spread within an interval of $\sigma_0 = C_S$ cache locations. The σ_k values are obtained with $\sigma_k = \sigma_{k-1} - \sigma_{k-2} \bmod \sigma_{k-1}$. Using the technique of section 3.4, n_s , $\|\text{ARS}(Y(j_2, j_3))\|$, S' and σ' are computed. Table 3.5 shows the values of the main variables for $N = 100$ and different values of M . Figure 3.5b illustrates the model precision and the impact of self-interferences on global performance. As can be seen, for $M = 800$, self-interferences of Y are nearly total, temporal reuse cannot be exploited. Spatial reuse can still be exploited, so that the miss ratio is close to the theoretical minimum of $\frac{1}{L_S} = 0.125$. But cross-interferences can yield an even lower miss ratio.

M	s	σ_{s-1}	σ_s	n_s	$S_Z\text{ARS}$	S'	σ'	#miss
800	2	800	32	64	2621	64	800	98352
801	2	800	73	192	6892	168	800	45141
802	∞				10000	100	327	0

Table 3.5: Examples of values obtained.

3.6 Cross-Interferences

Consider the reuse set of a reference R (reuse occurs on loop l) and the corresponding interference set of a reference R' .

Definition 3.9 If a reuse or an interference set is defined on loop l , its cache position is defined by⁹

$$\phi(\text{set}(R)) = (r_0 + r_1 j_1 + \dots + r_l j_l) \bmod C_S$$

Lemma 3.1 The relative cache position of the interference set of R' with respect to the reuse set of R is equal to

$$\phi(\text{AIS}(R')) - \phi(\text{ARS}(R)),$$

and the relative distance between the two sets is equal to

$$\delta = |\phi(\text{AIS}(R')) - \phi(\text{ARS}(R))|.$$

Example Consider loop 3.5a. The reuse of $Y(j_3, j_2)$ occurs on loop j_1 . So the corresponding interference set of $Z(j_3, j_1)$ is defined on loop j_1 . The position of the actual reuse set of $Y(j_3, j_2)$ is equal to $\phi(\text{ARS}(Y(j_3, j_2))) = (y_0) \bmod C_S$, and the position of the actual interference set of $Z(j_3, j_1)$ is equal to $\phi(\text{ARS}(Z(j_3, j_1))) = (z_0 + j_1) \bmod C_S$.

3.6.1 Internal cross-interferences

Definition 3.10 If δ is independent of j_1, \dots, j_l , the cross-interferences between R and R' are called **internal cross-interferences**.

Proposition 3.5 If the cross-interferences between R and R' are internal cross-interferences, the size of the intersection $CL(\text{ARS}(R), \text{AIS}(R'), \delta)$ between $\text{ARS}(R)$ and $\text{AIS}(R')$ is constant over the iterations of loops

⁹Note that it actually corresponds to the cache position of the first element of the set.

j_1, \dots, j_l . The number of cache misses due to such cross-interferences is then equal to

$$N_1 \times \dots \times N_l \times CL(ARS(R), AIS(R'), \delta).$$

Proposition 3.6 *The intersection between two intervals of size S_1 and S_2 , separated by a distance of $\delta = |\phi(S_1) - \phi(S_2)|$ cache locations, is equal to*

$$CL(S_1, S_2, \delta) = \frac{\min((S_2 - \delta)^+, S_1) + \min((S_1 + \delta - C_S)^+, S_2)}{L_S}$$

For instance, $S_1 = \|ARS(R)\|$ and $S_2 = \|AIS(R')\|$.

If the sets correspond to a collection of intervals (instead of one single interval), each interval of the reuse set is compared with each interval of the interference set. The size of the intersection is obtained by summing over all these subcases.

Example

```
DO j1=0,M-1
  regA = YA(j1)
  regB = YB(j1)
  DO j2=0,M-1
    C(j1,j2) = (XA(j2)+XB(j2))*(regA+regB)
  ENDDO
ENDDO
```

Loop 3.6.1a (N=1000, M=250).

Execution time for 100 runs (seconds)

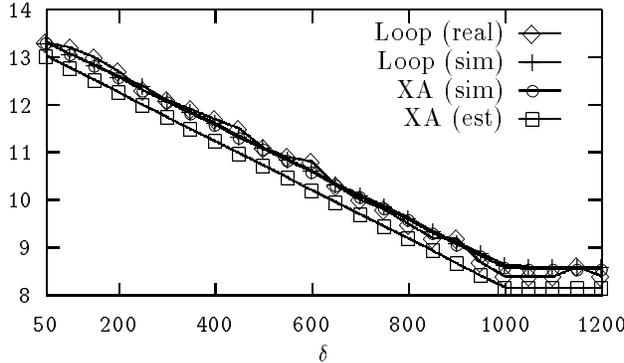


Figure 3.6.1b: Execution time.

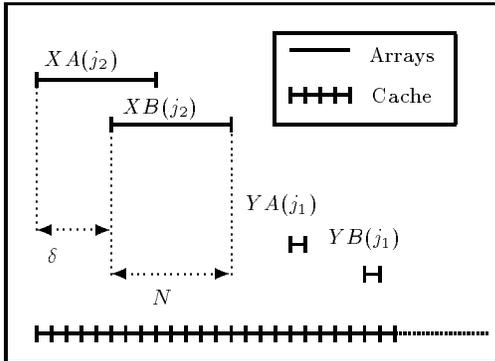


Figure 3.6.1c: Layout in cache.

Figure 3.6.1: Internal Cross-Interferences.

Consider loop 3.6.1a. Temporal interferences between XA and XB are likely to account for most interferences (C only breeds compulsory misses). The role of both arrays is symmetric. Consider array XA . Its reuse set is an interval of $S_1 = N$ cache locations, as well as the corresponding interference set of XB , i.e., $S_2 = N$. The relative cache distance between the two sets is $\delta = |xa_0 \bmod C_S - xb_0 \bmod C_S|$. So, according to the above proposition, the total number of internal cross-interference misses of XA is equal to $N_1 \times \frac{\min((S_2 - \delta)^+, S_1) + \min((S_1 + \delta - C_S)^+, S_2)}{L_S}$. Figure 3.6.1 illustrates the model precision and the impact of internal cross-interferences on global performance. As can be seen, the execution time decreases when δ decreases, until $\delta > N$, i.e., until the two sets do not overlap.

3.6.2 External cross-interferences

For external cross-interferences, an accurate or approximate evaluation can be used, which are tradeoffs between accuracy and complexity. Only the approximate evaluation is described here. The accurate evaluation is described in [13], and see also [4] for the example considered below.

Approximation Let us assume that δ (the distance between the reuse set of R and the interference set of R') is a random variable with a uniform distribution. Interferences are averaged over all values of δ . Hence the proposition:

Proposition 3.7 *The approximate number of cache misses due to external cross-interferences between R and R' over execution of the loop nest is equal to*

$$f_a(ARS(R), AIS(R')) = \frac{N_1 \times \dots \times N_l}{C_S} \times \sum_{\delta=0}^{\delta=C_S-1} CL(\|ARS(R)\|, \|AIS(R')\|, \delta)$$

where $CL(\|ARS(R)\|, \|AIS(R')\|, \delta)$ is the function defined in section 3.6.1.

Example Consider loop 3.6.2a, which corresponds to the multiplication of an $N \times N$ matrix A with a vector X .

```
DO j1=0,M-1
  reg = Y(j1)
  DO j2=0,M-1
    reg += A(j2,j1) * X(j2)
  ENDDO
  Y(j1) = reg
ENDDO
```

Loop 3.6.2a.

Average Execution time per reference (in 10^{-7} seconds)

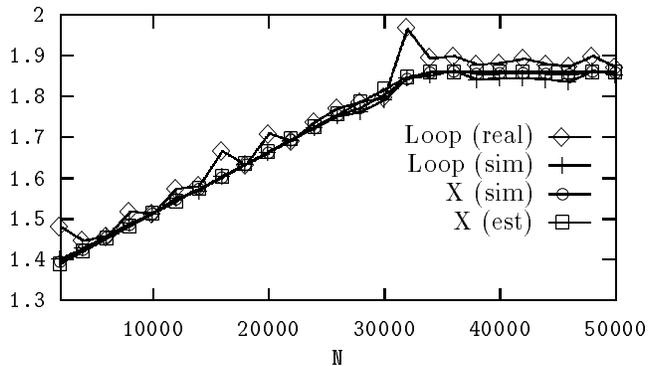


Figure 3.6.2b: Execution time.

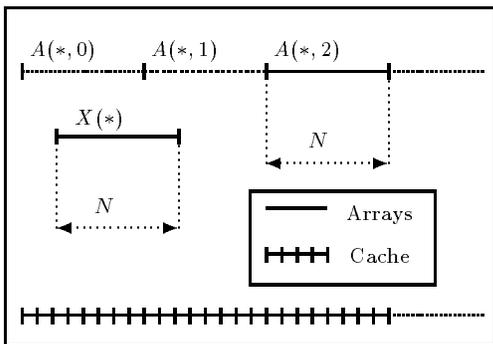


Figure 3.6.2c: Layout in cache.

Figure 3.6.2: External Cross-Interferences.

Assuming that $N < C_S$ (i.e., no capacity miss occurs), $\|ARS(X(j_2))\| = N$ and $\|AIS(A(j_2, j_1))\| = N$. So $CL(N, N, \delta) = (N - \delta)^+ + (N + \delta - C_S)^+$, and the approximate number of external cross-interference misses of X due to A is equal to $N \times \frac{1}{C_S} \times \sum_{\delta=0}^{C_S-1} CL(N, N, \delta) = \frac{N^2(N+1)}{C_S L_S}$.

As can be seen on figure 3.6.2, the time per reference increases with N , mostly because the temporal reuse of array X is disrupted by array A , until a threshold value is reached, where the temporal reuse cannot be exploited at all.

3.7 Extension to Group-Dependence Reuse

The techniques of sections 3.2 to 3.6 can be extended to group-dependence reuse. In opposition to self-dependences, the reuse set is defined by two references R_1, R_2, R_2 reusing the data of R_1 .

The reuse loop level is simply the loop level where reuse occurs. The reuse set is defined as *the set of elements referenced by R_1 that are reused by R_2* . Consequently both the reuse set and the interference set are not defined over the whole execution of one or several

loops, but over a *subset of iterations* of one or several loops.

The main difference with self-dependences is that the reuse set (as well as the interference set) is "moving" over each iteration of the reuse loop. Consequently, a more formal definition of the interference set is required. For each element of the theoretical reuse set, the theoretical interference set is equal to the elements loaded in cache, between two references to that element. This definition is apparently dependent on each element of the reuse set, but generally, all elements within a reuse set behave the same way. A surprising consequence is that internal-cross interferences disrupt group-dependence reuse in a *boolean* way. Either *no element of the reuse set* or *all elements of the reuse set* are flushed from cache.¹⁰ For external cross-interferences, the accurate evaluation can be difficult to derive, but the approximate evaluation can still be used: it is computed for one element of the reuse set and then extended to all elements of the reuse set. These notions are illustrated with an example below.

Example Consider loop 3.7a. There is a group-dependence between $X(j_2, j_1)$ and $X(j_2, j_1 + 1)$, that can be disrupted by $XY(j_2, j_1)$. Since all arrays are assumed to have the same leading dimension, the references are in translation, and cross-interferences are internal cross-interferences. The reuse set is defined on loop j_2 and corresponds to $\{X(0, j_1 + 1), \dots, X(N_2 - 1, j_1 + 1)\}$. For element $X(j_2, j_1 + 1)$, the interference set is equal to $\{XY(j_2, j_1), \dots, XY(N_2 - 1, j_1), XY(0, j_1 + 1), \dots, XY(j_2 - 1, j_1 + 1)\}$. Let $\delta = x_0 + N_2 - xy_0$, i.e., the relative distance between $X(j_2, j_1 + 1)$ and its interference set, then internal cross-interferences occur if $0 \leq \delta < N_2$ (while the condition is $|\delta| < N_2$ for self-dependence reuse). More intuitively, if the interference set is located *before* the reuse set, it is going to "sweep away" all elements of the reuse set before they can be reused (see figure 3.7c). So in this case, if internal cross-interferences occur, no group-dependence reuse can be achieved. This phenomenon is illustrated on figure 3.7, where $N_1 = 500, N_2 = 500$ and δ is varied from -500 to $+1000$.

```
DO j1=0,N1-1
  DO j2=0,N2-1
    XY(j2,j1) = X(j2,j1+1) - X(j2,j1)
  EWDDO
EWDDO
```

Loop 3.7a (N1=N2=500).

¹⁰More exactly, all elements of the reuse set that can be flushed by the interference set. Some cache locations used by the reuse set may never be used by the interference set, even though both sets are "moving" in cache.

Execution time for 100 runs (seconds)

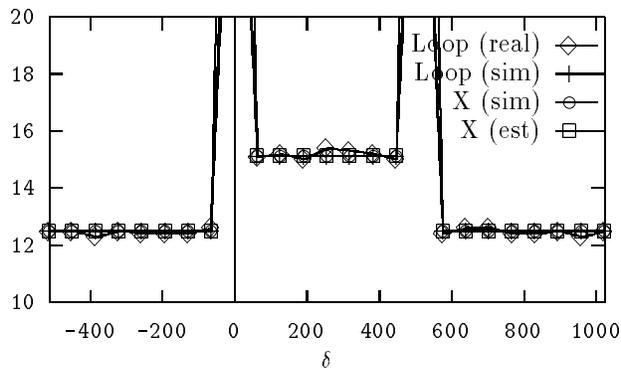


Figure 3.7b: Execution time.

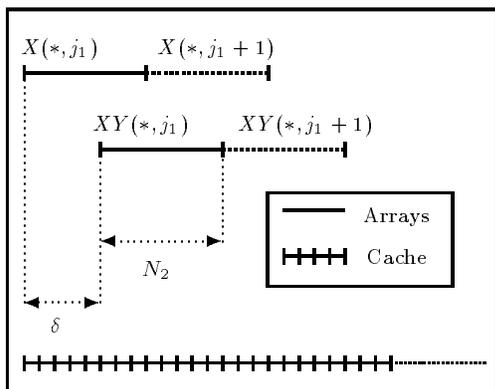


Figure 3.7c: Layout in cache.

Figure 3.7: Disruption of Group-Dependence Reuse.

3.8 Computing the Total Number of Misses

In previous sections, it is shown how to predict and compute the number of cache misses due to a given type of interference. Though this is the major issue, another difficult problem is to combine these results for several references occurring in the same loop.

3.8.1 Cumulating interference sets

It is not possible to simply add the number of cache misses corresponding to each reference and each type of interferences, because some interferences can be *redundant*. Consider the example below.

```

DO j1=0, N1-1
  DO j2=0, N2-1
    Y(j2) = A(j2, j1) - B(j2, j1)
  ENDDO
ENDDO

```

Arrays A and B can both induce external cross-interferences on array Y . The relative cache distance between the actual interference sets of these two arrays is equal to $\delta_{AB} = |(a_0 \bmod C_S) - (b_0 \bmod C_S)|$.

If $\delta_{AB} < N_2$, i.e., the two actual interference sets overlap, and in the same time they overlap with the reuse set of Y then A and B have a *redundant impact on Y* , and cache misses should not be counted twice.

In order to avoid such redundancy, references are not considered individually.

Definition 3.11 *Two references R and R' belong to the same translation group if the relative cache distance between R and R' is constant.*

The actual interference set of a translation group is the union of the actual interference sets of all references within this translation group. So *interferences due to a single reference are not considered anymore, instead, only interferences due to a translation group are considered*. This notion is useful to avoid redundant estimates of cross-interferences. Note that, in general, there are few translation groups within a loop body.

3.8.2 Selecting the proper dependence

Consider the following example.

```

DO j1=0, N1-1
  DO j2=0, N2-1
    Y(j1) = Y(j1) + A(j2, j1) * (X(j2+1) - X(j2))
  ENDDO
ENDDO

```

There is a self-dependence for $X(j_2)$ and a group-dependence from $X(j_2)$ to $X(j_2 + 1)$. The reuse distance of the group-dependence (1 iteration of j_2) is much shorter than that of the self-dependence (N_2 iterations of j_2). So, for most elements (excluding $X(0)$), reference $X(j_2)$ exploits the group-dependence reuse rather than the self-dependence. Therefore, interferences on the group-dependence instead of the self-dependence must be considered.

Interferences are evaluated for the dependence which corresponds to the smallest dependence distance. In general, array subscripts are simple enough to make this task tractable.

3.9 Redundancies

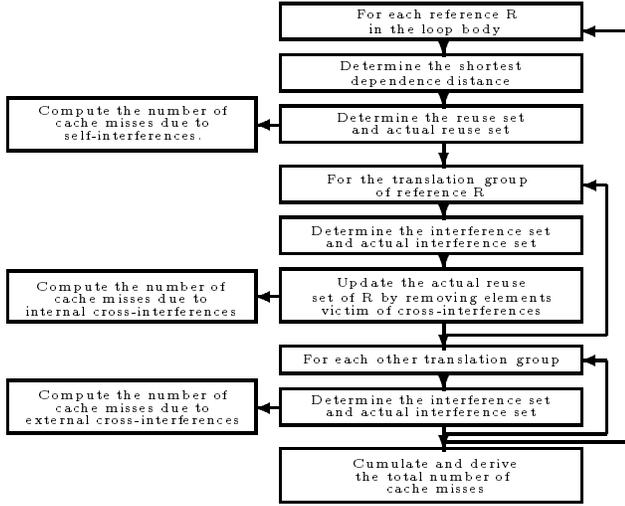
Globally, most redundancies are avoided because of the following reasons:

- Determining cross-interferences on the *actual reuse set* instead of the *theoretical reuse set* avoids redundant evaluation between *self-interferences and cross-interferences*.
- Determining internal cross-interferences *before* evaluating external cross-interferences and then updating the actual reuse set avoids redundant evaluation between *internal cross-interferences and external cross-interferences*.
- Redundancies within external cross-interferences are ignored, because they proved to be negligible in most

cases;¹¹ this assertion is illustrated in section 4.

3.9.1 Global Algorithm

The global algorithm for computing the number of cache interference misses is the following.



4 Putting It All Together

In this section, the number of cache misses of loop 2a is computed, based on the techniques presented in section 3. The leading dimension of arrays F, U is KU . As mentioned in section 2, one parameter δ is used to characterize the distance between the different arrays¹². As can be seen on figure 2, four intervals can be distinguished. Because of paper length constraints, only the interval $[0, KU - 1]$ is considered here. This interval corresponds the most complex case because many interference phenomena occur at the same time.

Compulsory misses

- The number of compulsory misses due to each array LDA, LDB, LDS is equal to $\frac{KU}{L_S}$.

- Since references $U(k, i)$ and $U(k, j)$ do not belong to the same translation group, the potential reuse between the two references is ignored. Therefore it is considered that both references breed compulsory misses: $\frac{KU}{L_S}$ for $U(k, i)$, and $\frac{JU \times KU}{L_S}$ for $U(k, j)$; the same for $F(k, i)$ and $F(k, j)$.

- Consequently, the total number of compulsory misses is equal to

$$N_{\text{comp}} = \frac{5 \times KU + 2 \times JU \times KU}{L_S}$$

Internal cross-interferences: self-dependence reuse

¹¹It is possible, though, to come up with examples where redundancies between external cross-interferences are significant.

¹²This assumption does not influence the complexity of the computations.

- The actual reuse set of $LDA(k)$ is a set of $\frac{KU}{L_S}$ consecutive cache lines. All references $LDB(k), LDS(k), U(k, i), F(k, i)$ belong to the same translation group as $LDA(k)$. The actual interference sets of $U(k, i), F(k, i)$ do not overlap with the actual reuse set of $LDA(k)$ ¹³. The actual interference sets of both $LDB(k)$ and $LDS(k)$ overlap with the actual reuse set of $LDA(k)$, but the impact of $LDS(k)$ is redundant with that of $LDB(k)$ (see figure 2c). The actual interference set size of $LDB(k), LDS(k)$ combined is equal to $\frac{\min(KU + \delta, 2 \times KU)}{L_S}$, and overlaps by $\frac{(KU - \delta)^+}{L_S}$ (where $\alpha^+ = \max(\alpha, 0)$) with the actual reuse set of $LDA(k)$. Therefore, internal cross-interferences on $LDA(k)$ induce $\frac{JU \times (KU - \delta)^+}{L_S}$ cache misses.

- As for $LDA(k)$, internal cross-interferences on $LDS(k)$ induce $\frac{JU \times (KU - \delta)^+}{L_S}$ cache misses.

- For $LDB(k)$, the actual interference set size of $LDA(k), LDB(k)$

combined is equal to $\frac{\min(KU + 2\delta, 2 \times KU)}{L_S}$, and it overlaps by $\frac{2 \times (KU - \delta)^+}{L_S}$ with the actual reuse set of $LDB(k)$. Therefore, internal cross-interferences on $LDB(k)$ induce $\frac{JU \times \min(2 \times (KU - \delta)^+, KU)}{L_S}$ cache misses.

- The actual interference set of $F(k, i)$ overlaps by $\frac{\delta}{L_S}$ cache lines with the actual reuse set of $U(k, i)$. So, there are $\frac{JU \times (KU - \delta)^+}{L_S}$ cache misses due to internal cross-interferences on $U(k, i)$. The role of $U(k, i)$ and $F(k, i)$ is symmetric; therefore there are $\frac{JU \times (KU - \delta)^+}{L_S}$ cache misses due to internal cross-interferences on $F(k, i)$.

So, the total number of internal cross-interferences corresponding to self-dependence reuse disruption is equal to

$$N_{\text{int_self}} = \frac{4 \times JU \times (KU - \delta)^+ + \min(2 \times (KU - \delta)^+, KU)}{L_S}$$

Internal cross-interferences: group-dependence reuse

Consider the group-dependence between $U(k, j)$ and $U(k, j + 1)$. The actual reuse set of $U(k, j + 1)$ starts at location $u_0 + KU$ (since the leading dimension of U is KU). The actual interference set corresponding to $F(k, j)$ starts at f_0 . $F(k, j)$ induces internal cross-interferences if $0 \leq f_0 - u_0 < KU$. This inequation is equivalent to $0 \leq \delta < KU$ (since $f_0 = u_0 + \delta$), which is satisfied by the hypotheses. So, there are internal cross-interferences of the group-dependence reuse between $U(k, j)$ and $U(k, j + 1)$. As mentioned in section 3.7, such interferences are boolean, so none of the

¹³For sake of clarity, the relative distance between u_0 and lda_0 has been chosen so that no internal cross-interference occurs between $U(k, i), F(k, i)$ and $LDA(k), LDB(k), LDS(k)$. However, such cross-interferences would be no more difficult to compute than the ones studied in this paragraph.

$KU-1$ elements of the reuse set can be reused. The condition for interferences to occur is not satisfied for the group-dependence between $U(k, j+1)$ and $U(k, j+2)$. Consequently, the number of cache misses due to internal cross-interferences on group-dependence reuse is equal to

$$N_{\text{int}_{\text{group}}} = \frac{JU \times (KU - 1)}{L_S}.$$

External cross-interferences

- The external cross-interferences of the group-dependence reuse of U due to $LDA(k), LDB(k), LDS(k), F(k, i), U(k, i)$ are negligible and can be ignored.

- The external cross-interferences on $LDA(k), LDB(k), LDS(k), F(k, i), U(k, i)$, due to $U(k, j), F(k, j)$ are estimated for the updated actual reuse sets. The actual interference set size corresponding to the translation group $U(k, j), U(k, j+1), U(k, j+2), F(k, j)$ is equal to $3 \times KU$ (the impact of $F(k, j)$ is redundant). Then, using function f_a defined in section 3.6.2, the total number of external cross-interferences is equal to

$$N_{\text{ext}} = \frac{4 \times f_a((KU - \delta)^+, 3 \times KU) + f_a(\min(2 \times (KU - \delta)^+, KU), 3 \times KU)}{L_S}.$$

Total number of cache misses

$$N_{\text{total}} = N_{\text{comp}} + N_{\text{int}_{\text{self}}} + N_{\text{int}_{\text{group}}} + N_{\text{ext}}$$

The precision of this evaluation is illustrated on figure 2b.¹⁴

5 Conclusions and Future Directions

The main outcome of this work is to show that computing the number of cache misses due to interference phenomena is a tractable task, and a methodology has been presented to perform it. The other outcome of the paper is to show that interference phenomena occur frequently and can be intense. Compilers need to address the issue of cache interferences; optimizing codes for reducing capacity misses is not sufficient. A first application of this work is the integration of the miss ratio evaluation techniques in a compiler, for performance evaluation and prediction purposes. Such an implementation would fully validate the possibility of automatically computing the number of cache misses, and would also enhance the model by unveiling potential issues that would not show in a theoretical study. A second application is the refinement of data locality optimization techniques, and mainly the accurate evaluation of optimal block sizes.

¹⁴The peaks occurring at $\delta = 0, 700, 1400$ correspond to ping-pong phenomena (spatial reuse disruption, see section 2) which have not been discussed here, but can be simply identified by checking the relative position of the different arrays.

References

- [1] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputing Performance Evaluation and the Perfect Benchmarks. In *Supercomputing '90*, pages 254–266, 1990.
- [2] Christine Eisenbeis, William Jalby, Daniel Windheiser, and Francois 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] Jeanne Ferrante, Vivek Sarkar, and Wendy 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, March 1993.
- [5] J. Heinrich G. Kane. *MIPS RISC Architecture*. Prentice-Hall, 1992.
- [6] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.
- [7] Mark Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California Berkeley, 1987.
- [8] Monica Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance of Blocked Algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.
- [9] Kathryn S. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Rice University, Technical Report CRPC-TR92214, April 1992.
- [10] O. Patashnik R. L. Graham, D. E. Knuth. *Concrete mathematics*. Addison-Wesley, 1989.
- [11] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.
- [12] Alan Smith. Cache Memories. *Computing Surveys*, 14(3), September 1982.
- [13] O. Temam, C. Fricker, and W. Jalby. Cache Interference Phenomena. Technical report, University of Leiden, 1993.
- [14] O. Temam, C. Fricker, and W. Jalby. Impact of cache interferences on usual numerical dense loop nests. In *Proceedings of the IEEE, special issue on Computer Performance Evaluation*, 1993.
- [15] 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. In *Proceedings of IEEE Supercomputing*, 1993.
- [16] Michael Wolf and Monica 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.
- [17] She Zhiyu, Zhiyuan Li, and Pen-Chung Yew. An Empirical Study on Array Subscripts and Data Dependencies. Technical Report 840, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, August 1989.