Characterizing the behavior of sparse algorithms on caches

Olivier Temam, William Jalby

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1992
25ème anniversaire

N° 1666

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

CHARACTERIZING THE BEHAVIOR OF SPARSE ALGORITHMS ON CACHES

Olivier TEMAM
William JALBY

Avril 1992

RR.1666

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Characterizing the Behavior of Sparse Algorithms on Caches

Olivier Temam, William Jalby*

April 6, 1992

Publication Interne n° 652 - Avril 1992 - 20 pages - Programme 1

Abstract While there are many studies on the locality of dense codes, few deal with the locality of sparse codes. Because of indirect addressing, sparse codes exhibit irregular patterns of references. In this paper, the behavior on cache of one of the most frequent primitives SpMxV *Sparse Matrix-Vector multiply* is analyzed. A model of its references is built, and then performance bottlenecks of SpMxV are analyzed using model and simulations. Main parameters are identified and their role is explained and quantified. Then, this analysis is used to discuss optimizations of SpMxV. Moreover a blocking technique which takes into account the specifics of sparse codes is proposed.

Caractérisation du Comportement des Algorithmes Creux sur les Mémoires Caches

Résumé Contrairement aux codes denses, peu d'études traitent de la localité des codes creux. En raison de l'adressage indirect, les codes creux exhibent des références irrégulières. Dans cet article, le comportement sur le cache de l'une des primitives creuses les plus fréquentes SpMxV *Matrice-Vecteur Creux* est analysé. Un modèle des références engendrées par cette primitive est construit, et les facteurs limitatifs des performances de SpMxV sont analysés à l'aide du modèle et de simulations. Les principaux paramètres sont identifiés et leur rôle est expliqué et quantifié. Puis cette analyse est utilisée pour discuter des optimisations de SpMxV. De plus, une méthode de blocking prenant en compte les spécificités des codes creux est proposée.

Keywords: sparse primitives, cache, performance prediction, data locality.

*IRISA/INRIA, Campus de Beaulieu, 35042 Rennes CEDEX, France

1 Introduction

Due to the increasing difference between memory speed and processor speed, it becomes critical to minimize communications between memory and processor, by addition of caches on the data path. However, a consequence of this worsening difference is that the cost of a cache miss, in terms of processor clock cycles, is becoming quite large, making it critical to improve the hit ratio [17].

Numerical codes are now some of the most demanding programs in terms of execution time and memory usage. The existing litterature related to the study of numerical codes behavior on cache memories focuses on regular do-loops, i.e with linear references to arrays [16], [5]. There is an important set of numerical codes, "sparse codes", which do not belong to this category. Sparse numerical codes like classic numerical codes are made of a collection of simple numerical primitives. We chose to study *Sparse Matrix-Vector multiply* (SpMxV) because it is among the most frequently used ones along with *gaussian elimination* [7], and still it is simple enough to allow a sharp analysis of its workings (cf. figure 5); moreover, a number of sparse primitives exhibit rather similar patterns of references (i.e a few arrays addressed regularly and indirect addressing to another array). Because of indirect addressing, sparse codes have the particularity of breeding *irregular patterns of references to memory and to cache*, and consequently, the behavior of caches under numerical workloads is seemingly non-predictible and hard to analyze. Because of this *apparently random* behavior, caches, which principles rely on locality of programs, are generally said to be inefficient with sparse codes [25].

However, in this paper, we show that this assumption is true only for a restricted domain of main problem parameters (cache size, line size, matrix bandwidth and number of non-zero elements). Even then, in some cases, it is possible and *worthwhile* to exploit unused locality through software techniques. Though classic blocking methods hardly allow the utilization of this locality (cf. section 5.2.1), it is possible to exploit it through blocking techniques that take into account the specifics of sparse matrices.

In section 2 of this paper, spatial and temporal locality of SpMxV is qualitatively evaluated and potential problems are identified. Then, in section 3, a meaningful share of the paper is devoted to modeling the non-regular references which appear in SpMxV, because an unavoidable preliminary to evaluating and optimizing a primitive is a good understanding of its behavior. Besides, the purpose of this section is to show that it is possible to predict the behavior and performance of sparse codes, and to actually *quantify* their impact on caches. In section 4, using the model and simulations, the behavior of SpMxV is characterized according to the values of the parameters. Finally, in section 5, software and hardware optimizations are discussed, and a blocking technique based on the observations of the previous section is presented.

2 A qualitative study of locality within SpMxV

Position of the problem The purpose of the paper is to analyze SpMxV on caches. *Storage-by-row* has been chosen because it is among the most commonly used storage techniques (see page 3 for more details). Otherwise, for sake of simplicity the cache is assumed to be direct-mapped. It can be seen in section 4.4 that this hypothesis is not very restrictive since set-associative and direct-mapped caches exhibit relatively similar behaviors with SpMxV, and that the model built can be extended to set-associative caches (cf. section A).

2.1 Data locality

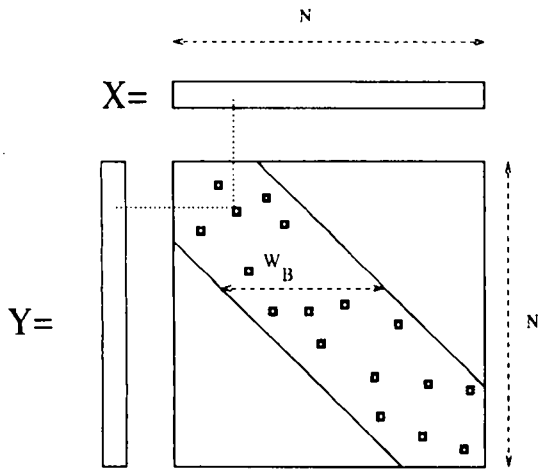
A first step to understanding the behavior of SpMxV on cache is to study the locality of the data used by this primitive. Since there are N_{nz} references to arrays X, *Index* and *Matrix*, and $2N$ references to arrays Y and D, the total number of references is $3 * N_{nz} + 4 * N$ (cf. figure 5).

- Arrays Y and D have very similar behaviors in the sense that they both exhibit flawless spatial and temporal locality. Most probably $D(I)$ and $D(I+1)$ will be stored in registers and therefore should not provoke a reference to memory on each iteration of loop J. So arrays Y and D (of size N) are mostly responsible for *intrinsic misses*, and therefore account for a small share of total cache misses (since, in general $N \ll N_{nz}$).

- Arrays *Matrix* and *Index* have no temporal locality and again exhibit flawless spatial locality. These two arrays account for $2N_{nz}$ references, that is, a major part of total references. Since no element is reused, cache misses due to these arrays are only *intrinsic misses*. Because of their size, they may also provoke important *cross-interferences* with other arrays (i.e. flush other arrays from cache).

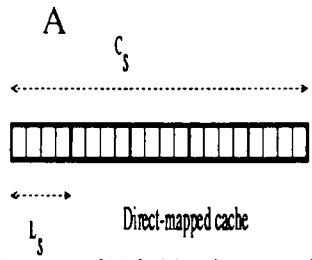
- Because of the indirect addressing through array *Index*, array X exhibits a complex behavior. If a uniform distribution of non-zero elements on a row within the band (of size W_B) is assumed, then the average distance between two columns with non-zero elements is $\frac{W_B}{n_{nz}}$. Therefore, in most cases there is some spatial locality if $\frac{W_B}{n_{nz}}$ is of the order of L_S. Actually, in usual *finite-element* matrices, the non-zero elements are sometimes grouped along specific diagonals, within the band (cf. [8] and figure 6). In that case, the spatial locality may not be negligeable even if $\frac{W_B}{n_{nz}} \gg L_S$.

  Array X is the only array which presents an unexploited temporal locality, and therefore from which significant gains can be expected; however the temporal locality of X is non-trivial and therefore hard to analyze and exploit. That is why our efforts will mainly focus on analyzing the behavior of array X. Due to the properties of sparse matrices (especially *finite-element* ones), if there is approximately n_{nz} non-zero elements per row, there is also about n_{nz} non-zero elements per column. A first consequence of that observation is that



N	:	Matrix dimension
N_{nz}	:	Total number of non-zero elements
$n_{nz} = \frac{N_{nz}}{N}$	:	Average number of non-zero elements per row
W_B	:	Matrix bandwidth

Figure 1: *Banded sparse matrix; matrix parameters.*



C_S	:	Cache size
L_S	:	Line size

Figure 2: *Direct-mapped cache; cache parameters, types of misses.*

Remark: *All cache dimensions are divided by the size of a single-precision floating-point element (4 bytes), so that a size S actually corresponds to $S \times 4$ bytes.*

Cross-interference misses (or conflict misses)	:	an array element flushed by an element of another array.
Self-interference misses (or capacity misses)	:	an array element flushed by an element of the same array.
Intrinsic misses (or compulsory misses)	:	an array element loaded for the first time.

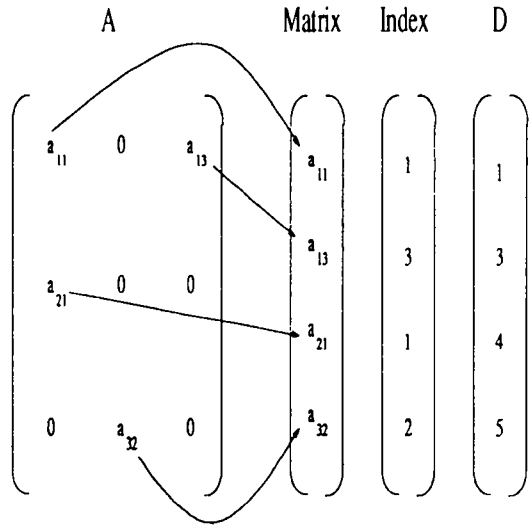


Figure 3: *Example of storage-by-row for a 3×3 sparse matrix*

```
DO I=1,N
   DO J=1,N
      Y(I) = Y(I)+A(I,J)*X(J)
   ENDDO
ENDDO
```

```
DO I=1,N
   REG = Y(I)
   DO J=D(I),D(I+1)-1
      REG = REG + Matrix(J)* X(Index(J))
   ENDDO
   Y(I) = REG
ENDDO
```

Figure 4: *Original loop nest; storage by row.*

Figure 5: *Problem parameters*

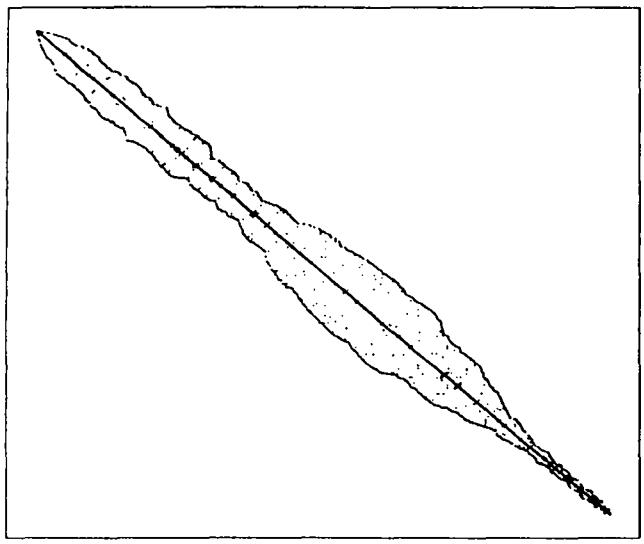


Figure 6: *Example of distribution of non-zero elements within finite-element matrices (matrix* 1138 BUS *of the Harwell-Boeing suite).*

each element of X may theoretically be reused n_{nz} times at best. Secondly, the average distance (in terms of iterations of loop I) between two reuses is approximately $\frac{W_B}{n_{nz}}$. Meanwhile, about $\frac{W_B}{n_{nz}} \times 3n_{nz}$ elements (from arrays X, *Matrix* and *Index*) are loaded into cache and may flush the elements to be reused.

The conclusion of the previous observations on X is that whether temporal locality and spatial locality of X are significant and can be exploited highly depends on W_B, C_S, L_S and n_{nz}.

3 Modeling: understanding and quantifying

From previous section, it appears that the main source of cache misses are misses of *Matrix* and *Index* which can be evaluated easily because they are intrinsic misses, and misses of X which are hard to estimate because addressing to array X is indirect and irregular. The misses of X are mainly *cross-interference* or *self-interference* misses. First of all, the simulations done (cf. section 4) show that the role of cross-interference and self-interference phenomenons on X is very similar. Second, both of the two kinds of misses can be modeled using techniques presented thereafter. Third, the purpose of our model is to provide a good understanding of the interactions between parameters rather than an accurate formula of the total number of cache misses. Therefore, for sake of simplicity, only self-interference misses are precisely modeled and quantified (cf. section 3.1), while only a gross estimate is given for cross-interference misses (cf. section 3.2.2).

3.1 Modeling self-interferences of array X

References to array X are highly irregular and consequently cannot be investigated through classic *deterministic* methods. Therefore, *probabilistic modeling* is being used. The main problem seems to choose a distribution which matches that of non-zero elements on a row within the band. Though it is for the least possible to find an approximate distribution for *finite-element* matrices, this yields formulas which are too complex to handle (cf. section A). Therefore, though most computations are conducted for any distribution $p(i,j)$ (probability that element (i,j) of A is non-zero), *uniform distribution* ($p(i,j) = p$) is employed for final calculi. The object of section 3.1 is to build the model of references to array X and compute the number of cache misses. This part needs not be read thoroughly by anyone not interested by model elaboration, though it provides an insight on the behavior of SpMxV.

3.1.1 Reducing problem P to problem P'

Let us consider original matrix A. All non-zero elements of A located on column j of this matrix breed a reference to element j of X. Now, let us consider the c^{th} cache location. All elements j of X such that $j \; mod \; C_S \in [c, c + L_S - 1]$ are mapped to the same cache line c. Therefore, all columns j of A such that $j \; mod \; C_S \in [c, c + L_S - 1]$ breed references to elements of X which are mapped to the same cache line.

Therefore, it is possible to divide the problem into $\frac{C_S}{L_S}$ sets of elements of X, all elements within a set being mapped to the same cache line. Similarly, A is divided into $\frac{C_S}{L_S}$ sets of columns, all breeding references to elements of X mapped to the same cache line (cf. figure 7). Since, the cache is direct-mapped, none of these sets interact with each other.

So, if the original problem can be formalized as follows

Problem P: *Compute an approximation of* $N_{cm}^{self} = N_{cm}(P)$*, the number of self-interference misses of array* X *in the sparse matrix-vector multiplication. The dimension of* X *is equal to* N*.* A *is an* $N \times N$ *matrix, and the cache is direct-mapped and of size* C_S*.*

Then P, is now equivalent to C_S subproblems $P_i, (1 \leq i \leq \frac{C_S}{L_S})$. Through simulations, it is possible to check that, for distributions occuring in *finite-element* matrices (non-zero elements are grouped along three diagonals) and even more for *uniform* distributions, it is a very fair approximation to assume that all subproblems P_i are equivalent.

Problem P': *Compute an approximation of* $N_{cm}(P')$*, the number of self-interference misses of array* X'*. The dimension of* X' *is equal to* N' *(where* $N' = \lfloor \frac{N}{C_S} \rfloor \times L_S$ *or* $N' = (\lfloor \frac{N}{C_S} \rfloor + 1) \times L_S$*).* A *is an* $N \times N'$ *matrix, and the cache is direct-mapped and of size* L_S*.*

Then, the number of cache misses of P is approximately equal to the number of cache misses of P' times C_S:

$$N_{cm}(P) \simeq \frac{C_S}{L_S} \times N_{cm}(P')$$

3.1.2 Reducing problem P' to problem P''

As it has been specified above, problem P' is composed of a cache of size L_S, an array X', and a $N \times N'$ matrix A'.

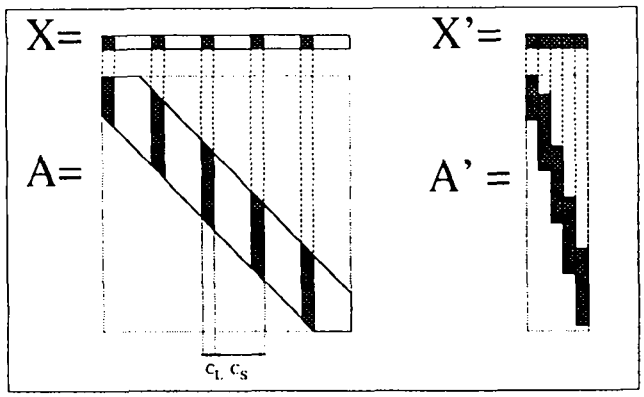


Figure 7: *Original problem P, and subproblem P'*

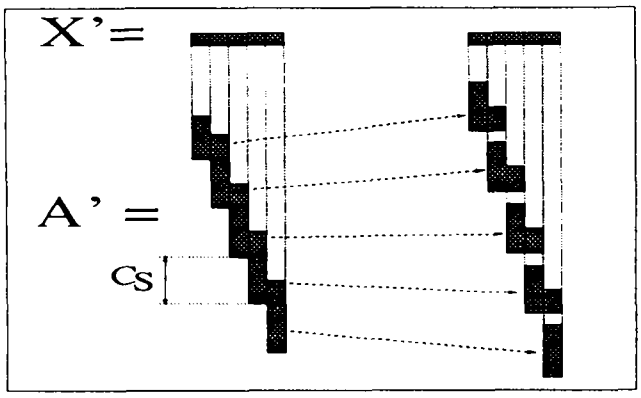


Figure 8: *Decomposition of subproblem P'*

Now, A' can be decomposed into sections of C_S rows, which would all have the same shape as shown on figure 8. Since, in general, $C_S < N$, there is a great number of such sections in A' (approximately $\frac{N}{C_S}$). Through experiments, it can be observed that the number of cache misses corresponding to the execution of each section becomes rapidly stable. Therefore, it is possible to restrict the study to only one section S' of A'. Then, the number of cache misses due to A' is approximately equal to that of one section times $\frac{N}{C_S}$.

Let us do an ultimate simplification. The number of columns in the sections described above is not constant, because of the banded shape of the matrix. In order to ease the computations even more, a section S' can be divided into two parts A''_1 and A''_2, each with a constant number of columns (cf. figure 9). The characteristics of each subsection are the following ones:

$$A''_1 \begin{cases} n_c^1 = \lfloor \frac{W_B}{C_S} \rfloor \\ n_l^1 = C_S - W_B \ mod \ C_S \end{cases}$$
$$A''_2 \begin{cases} n_c^2 = \lfloor \frac{W_B}{C_S} \rfloor + 1 \\ n_l^2 = W_B \ mod \ C_S \end{cases}$$

Problem P'' can now be defined as follows:

Problem P'': *Compute an approximation of $N_{cm}(P'')$, the number of self-interference misses of array X''. The dimension of X'' is equal to N'' (where $N'' = n_c$). A'' is an $n_l \times n_c$ matrix, and the cache is direct-mapped and of size L_S.*

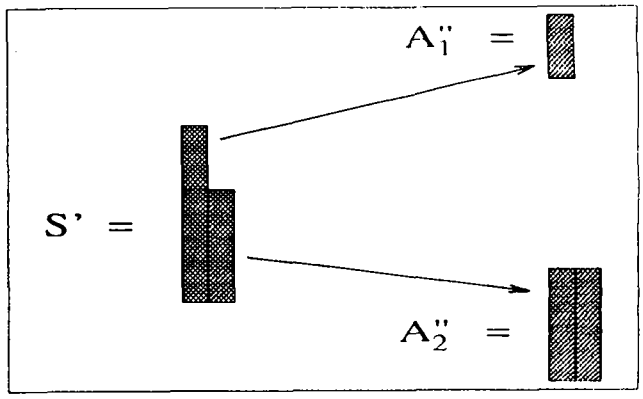


Figure 9: *Decomposition of subproblem P'*

3.1.3 Estimating the number of cache misses

Let us now formalize the notion of "cache miss" within the scope of problem P''. During execution of SpMxV, elements of a subsection A'' of problem P'' are referenced row-wise. Let us call $\pi^k_{out}(i,j)$ the probability that element k of X'' be out of cache right before element (i,j) of the subsection is referenced. Let us also call $p(i,j)$ the probability that element (i,j) of A'' be a non-zero element. Now, the probability that element j of X'' is not in cache, right before element (i,j) of A'' is being considered, is equal to $\pi^j_{out}(i,j)$. Therefore, the probability for a cache miss to occur at that moment is equal to $p(i,j) \times \pi^j_{out}(i,j)$.

Then, the number of cache misses due to A'' is given by the following expression:

$$N_{cm}(P'') \simeq \sum_{i=1}^{n_l} \sum_{j=1}^{n_c} \pi^j_{out}(i,j) \times p(i,j)$$

And, the total number of cache misses is equal to (cf. section A for more details):

$$\begin{aligned} N_{cm}(P) &\simeq \frac{C_S}{L_S} N_{cm}(P') \\ &\simeq \frac{C_S}{L_S} \times \frac{N}{C_S} N_{cm}(P''_1 \bigcup P''_2) \\ &\simeq \frac{N}{L_S} \times (N_{cm}(P''_1) + N_{cm}(P''_2)) \end{aligned}$$

An explicit and simple expression of $N_{cm}(P)$ can be derived easily only if $p(i,j)$ is constant, i.e if distribution is uniform. Nevertheless, since the goal is to get a hint at the interactions between parameters rather than a precise approximation of the number of cache misses, it is a good tradeoff to assume that distribution is uniform (cf. section 3.3 for more details).

So, if distribution is uniform, then $p(i,j) = p$ and p can be given as a function of problem parameters $p = 1-(1-\frac{N_{nz}}{NW_B})^{L_S}$ (cf. section 3.3 for more details). For this distribution of probability, the total number of self-interference misses on X is given by expression of figure 10.

3.2 Number of cache misses for each type of misses

3.2.1 Self-interferences of array X

$\frac{W_B}{C_S} > 1$: Since $N_{nz} \ll N \times W_B$ when W_B is sufficiently large, it can be assumed that $p \ll 1$ and that $p \simeq$

$$N_{cm}^{self} \simeq \frac{N}{L_S} \times \left(\frac{pn_c^1}{1-(1-p)^{n_c^1}} \times \left[n_l^1 (1-(1-p)^{n_c^1-1}) + p(1-p)^{n_c^1-1} \frac{1-(1-p)^{n_c^1 n_l^1}}{1-(1-p)^{n_c^1}} \right] + \frac{pn_c^2}{1-(1-p)^{n_c^2}} \times \left[n_l^2 (1-(1-p)^{n_c^2-1}) + p(1-p)^{n_c^2-1} \frac{1-(1-p)^{n_c^2 n_l^2}}{1-(1-p)^{n_c^2}} \right] \right)$$

Figure 10: *Expression of the total number of self-interference misses on array X*

$\frac{N_{nz}}{NW_B} \times L_S$. Therefore, a first order development of expression in figure 10 gives (cf. section A for more details)

$$N_{cm}^{self} \simeq N_{nz} - \frac{N_{nz}^2 \times C_S}{2 \times N \times L_S \times W_B} \quad (1)$$

$\frac{W_B}{C_S} < 1$: In that case, there are no self-interferences of array X because all active elements of X fit in the cache, therefore

$$N_{cm}^{self} = 0$$

3.2.2 Cross-interferences with array X

A second class of misses breeded by array X are the cross-interference misses between arrays *Matrix*, *Index* and array X.

Let us first consider the case where $W_B > C_S$ (which is the assumption made up to now). Because of their flawless spatial locality, the elements of *Matrix* and *Index* can be seen as two trains of references being translated along the cache. Each time one of these sets of references reaches a cache location, it flushes it. Therefore, considering that there are $\frac{N_{nz}}{N}$ non-zero elements per row, $2\frac{N_{nz}}{N}$ elements of *Matrix* and *Index* are loaded for each iteration of I, and consequently, each cache location is flushed $\frac{2N_{nz}}{C_S N}$ times per iteration of I.

Therefore, the probability that *Matrix* or *Index* flush a given cache line for one iteration of I is $p_{cross} = \frac{2N_{nz} \times L_S}{C_S}$.

Consequently, since there are N_{nz} references to array X, the number of cross-interference misses on X account for

$$N_c^{cross} m = p_{cross} \times N_{nz} = \frac{2N_{nz}^2 \times L_S}{C_S} \quad (2)$$

It must be noted that, when L_S is sufficiently large, the effect of self and cross-interference misses do not cumulate but become redundant.

The effect of cross-interference misses on other arrays is generally negligeable.

3.2.3 Intrinsic misses

There are N references to D, Y and X, and N_{nz} references to *Matrix* and *Index*. Therefore, the total number of intrinsic misses is given by:

$$N_{cm}^{int} = \frac{3N + 2N_{nz}}{L_S} \quad (3)$$

Since in general $N_{nz} \gg N$, arrays *Matrix* and *Index* represent the main source of intrinsic misses.

3.3 Model accuracy and validity

In order to understand the soundness and precision of the model, let us first analyze the approximations made, and then describe some experimental results obtained through simulation.

3.3.1 Discussing approximations

During the elaboration of the model, a number of approximations have been performed:

- The width of the band is considered to be equal to W_B on any row of the matrix, which is not true for $i \leq \frac{W_B}{2}$ and $i \geq N - \frac{W_B}{2}$. However, a good tradeoff for overcoming this problem is to consider that the actual number of rows is $N - \frac{W_B}{2}$ instead of N.

- The number of non-zero elements on a row is assumed to be constant. This is not an important approximation since this fact is relatively true for a great range of applications, especially for *finite-element* matrices.

- The number of cache misses is considered to be the same in all sections S' of A'. This approximation does not affect much the model either. All experiments done proved the good regularity of the number of cache misses across the different sections. The larger the matrix, the better is the approximation, because this number becomes stable after a few sections only. Most problems have very large dimensions which make this *start-up* effect negligeable.

- The main approximation probably lays in the distribution. Clearly, it appeared that *uniform distribution* does not always fit very precisely. This is especially true when cache line L_S is equal to 2 (*16 bytes*), because of the particular spatial locality of *finite-element* matrices (cf. figure 16). However, for most cases it provides a very good hint at the influence of each parameter. Besides, it is not too easy to understand the distributions of non-zero elements in real sparse matrices, and little litterature could be found on that subject. Therefore, approximate distributions had to be designed for some matrices, such as *finite-element* ones, and hopefully, these distributions proved to give a good description of real *finite-element* matrices. However these more complex distributions do not allow the computation of an analytical expression of the number of self-interference misses ($p(i,j)$ is not constant). Part of the formula obtained would have to be computed numerically (cf. section A).

3.3.2 Experimental testing of the model

In order to check the precison of the estimate, a cache simulator has been used.

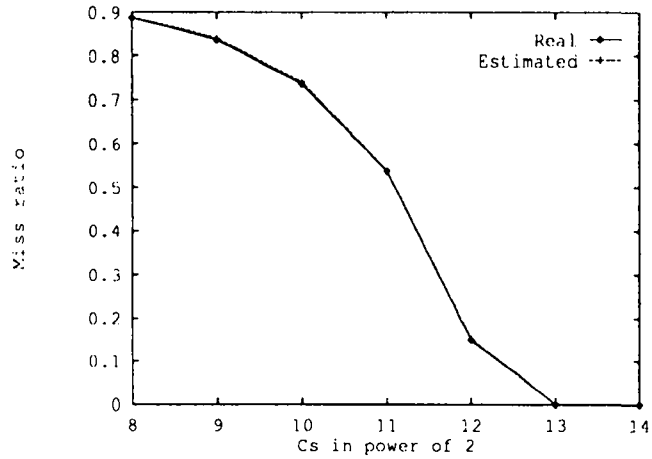


Figure 11: *Varying C_S: estimated and real number of cache misses*

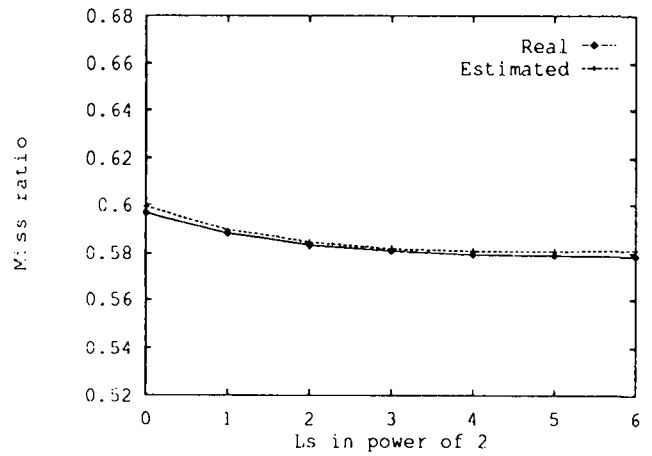


Figure 12: *Varying L_S: estimated and real number of cache misses*

Randomly generated matrices In addition to this simulator, a subroutine for generating sparse matrices with various distributions has been written. All parameters of cache and matrix can be varied in any domain. In order to get meaningful results, the number of cache misses presented is a mean value obtained through several experiments ($\simeq 100$) on different sparse matrices with identical characteristics (cf. figures 11, 12).

The parameters of the matrices used for the two graphs of this section are the following ones

N :	n_{nz}	W_B	C_S	L_S	D_A
10^5	10^6	5001	2048	2	$1 (or LRU)$

Real matrices A set of real matrices has also been used, namely the *Harwell-Boeing* [8] suite in order to validate the applicability of our model. It is this set of experiments which showed it was necessary to take into account more complex distributions than uniform ones, especially for small line sizes (cf. figure 16). Though some applications still present matrices with close to uniform distributions, especially after using renumbering techniques such as *minimum degree algorithm* (cf. figure 22).

4 Highlighting the role of the problem parameters

Of course, all problem parameters have an impact on SpMxV. However, through model analysis and experiments, three coefficients (L_S, $w = \frac{W_\mu}{C_S}$, $d = \frac{n_{nz}}{W_\mu}$) proved to have a major impact on the hit ratio, and are sufficient to characterize most phenomenons. L_S is a critical parameter, mainly because of its influence on *intrinsic misses*, but also on *cross-interference misses*. Parameter $w = \frac{W_\mu}{C_S}$, called *degree of interference*, it indicates how many elements of X conflict for the same cache line (cf. section 3.1.2), and therefore it reflects quite well the degree of self-interferences occuring on X. Finally, parameter $d = \frac{n_{nz}}{W_\mu}$, called *density*, corresponds to the average distance between two non-zero elements on a row and on a column of original matrix A (cf. section 2). In other terms, it is a measure of the degree of temporal and spatial locality of non-zero elements of matrix A, and consequently, of the references to array X.

Basing our analysis on the model obtained in section 3 and simulations, the role and importance of the above parameters is discussed in the following subsections. A small subsection is also devoted to discussing the difference between direct-mapped and set-associative caches. For sake of simplicity, the experiments used to make the graphs of this section are mainly based on uniformly distributed matrices, but account quite well for phenomenons occuring in real sparse matrices.

4.1 Line size L_S

Influence of L_S on the intrinsic misses of *Matrix* and *Index* The expression of the number of intrinsic misses (3) shows that this number decreases hyperbolically with L_S (cf. figure 13). Therefore, a small increase of L_S brings important reductions of the number of intrinsic misses.

Influence of L_S on self and cross-interference misses Let us assume that $\frac{W_B}{C_S} > 1$. In the case where non-zero elements are uniformly distributed across the sparse matrix, the approximate number of self-interference misses is given by (1). This expression is a function of L_S

$$N_{cm}^{self} = \alpha - \frac{\beta}{L_S}$$

Therefore, N_{cm}^{self} grows with L_S, though this increase is more or less moderate depending on β. Otherwise, $N_{cm}^{cross} = \frac{2N_{nz}^2 \times L_S}{C_S}$, therefore the number of cross-interference misses grows linearly with L_S. Consequently, an increase on L_S

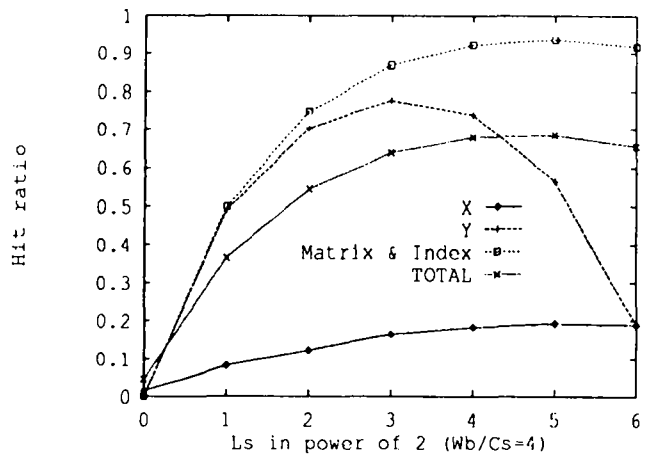


Figure 13: *Influence of L_S on the total hit ratio and the hit ratio of each array.*

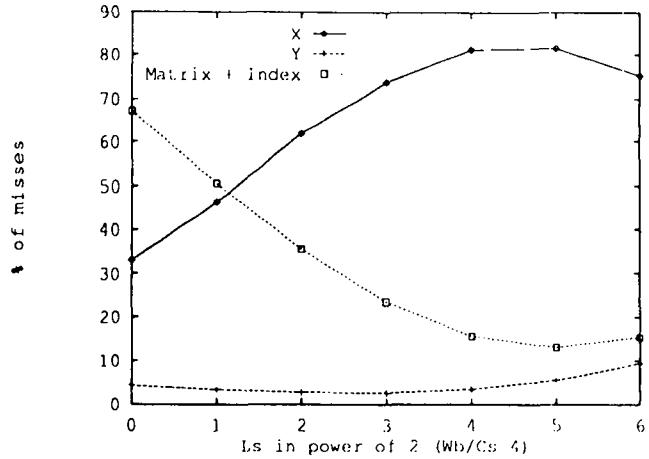


Figure 15: *% of misses of each array for different values of L_S.*

corresponds to an increase on both the number of self-interference and cross-interference misses (cf. figure 14).

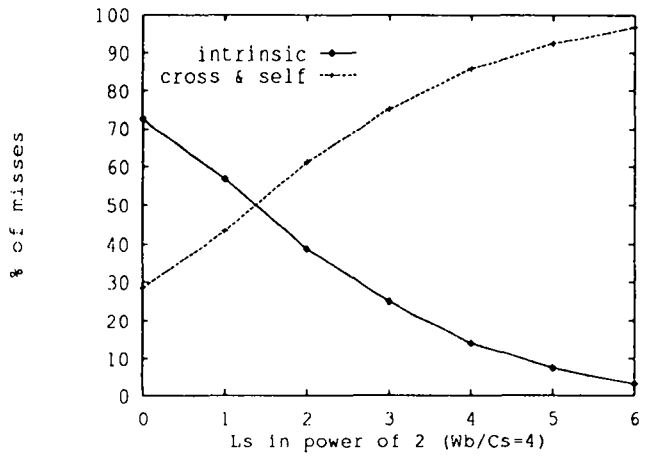


Figure 14: *% of* intrinsic misses *and* self + cross-interference misses *for different values of L_S.*

The consequence of the previous observations is that the hit ratio of all arrays but X increases very quickly (hyperbolically) with L_S. On the other hand, because of cross and self-interference misses, the hit ratio of X increases much more moderately (or sometimes even decreases) when L_S grows.

Therefore, for small values of L_S the main cause of cache misses are arrays Matrix and Index, while for high values of L_S, X accounts for the major part of cache misses (cf. figure 15).

Consequently, devoting important efforts to benefit from the temporal locality of X should be considered only when L_S is such that X becomes a major cause of cache misses, otherwise little improvements of total hit ratio can be expected (cf. figure 15).

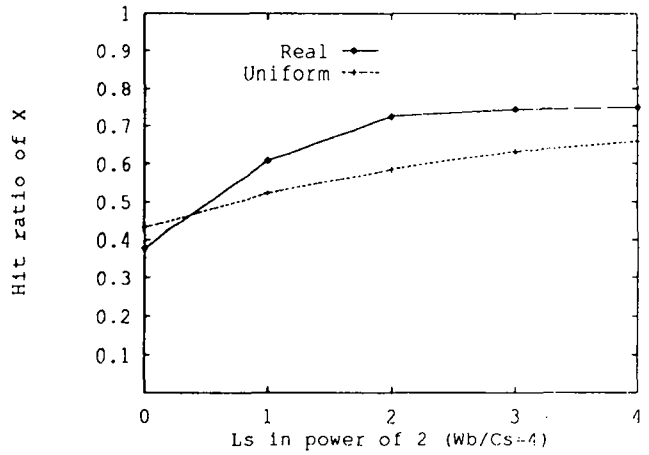


Figure 16: *Influence of the distribution of non-zero elements on the spatial locality of references to X*

Proper values of L_S for finite-element sparse matrices Due to the properties of mesh structures (where each node has a relatively constant number of neighbors) and the use of renumbering algorithms to minimize bandwidth, finite-element sparse matrices exhibit a non-uniform distribution of non-zero elements. They are grouped by packs of 2,3 or 4 elements depending on the mesh type. This spatial locality of non-zero elements induces a spatial locality of references to array X. Therefore, $L_S = 2$ *or* 4 is sufficient to make use of this locality, while little improvement can be expected for higher values of L_S.

As it can be seen on figure 16, the reduction of cache misses between $L_S = 1$ and $L_S = 2$ is impressive, while it steps down after $L_S = 4$. This phenomenon is characteristic of finite-element matrices. It can be noted on figure 16 that an increase of L_S breeds progressive instead of drastic improvements on the hit ratio of X, when the non-zero elements are uniformly distributed within the sparse matrix. The shape itself of finite-element sparse matrices (cf. fig-

ure 6) suggests a greater spatial and temporal locality than of uniformly distributed matrices. However, both uniform and finite-element distributions tend to behave similarly for large enough cache line sizes, i.e when the locality effect of finite-element sparse matrices does not show anymore.

4.2 Degree of interference $w = \frac{W_B}{C_S}$

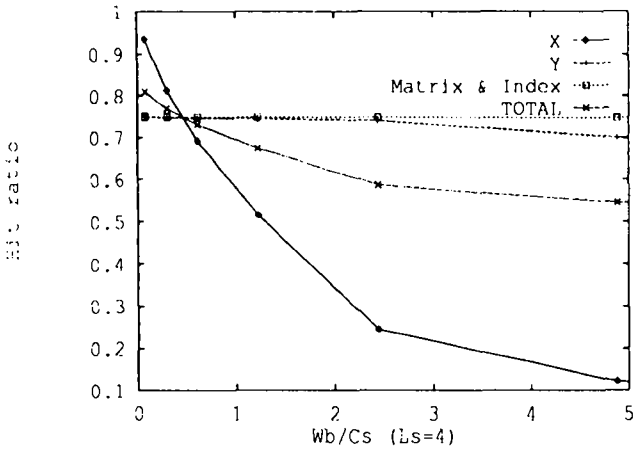


Figure 17: *Influence of $\frac{W_B}{C_S}$ on the total hit ratio and the hit ratio of each array.*

Self-interference misses The effect of W_B and C_S cannot be dissociated. For $W_B < C_S$, the number of self-interference misses due to X is equal to zero. This appears clearly when considering *simplified problem* P'': the number of columns of A'', i.e the number of interfering columns, is equal to $\lfloor \frac{W_B}{C_S} \rfloor$ or $\lfloor \frac{W_B}{C_S} \rfloor + 1$, that is, 0 *or* 1. Therefore, once an element is loaded into a cache line, it cannot be flushed by another element. So, *for $W_B < C_S$, the number of cache misses due to X is optimum.* Now, for $W_B > C_S$, model expression (1) shows that the number of self-interference misses increases hyperbolically with w

$$N_{cm}^{self} = \alpha - \frac{\gamma}{w}$$

Now, if the previous expression is considered as a function of w, and is differentiated it, then it appears that the increase of N_{cm}^{self} becomes small (i.e less than 10 %) whenever $w \geq \sqrt{10\alpha}$.

So, three cases can occur. First $w < 1$, the number of self-interference misses is negligeable and it is not useful to reduce W_B. Second $w \simeq 1$, in this interval the number of self-interference misses increases hyperbolically with w, and therefore very significant improvements can be obtained through slight bandwidth reduction. Third $w \gg 1$, and the number of self-interference misses is close to maximum (nearly no element of X is reused), and only a drastic bandwidth reduction may bring improvements.

Cross-interference misses When $w \gg 1$ is sufficiently large, because of self-interferences only, there is little reuse on X. Therefore, the effect of *Matrix* and *Index*, i.e cross-interferences, can only be redundant with that of X.

When $w \leq 1$, there are no self-interference misses. In that case, *alive* (i.e currently used) elements of X are located in an area of size W_B within cache. Now, as mentioned in section 3.2.2 *Matrix* and *Index* can be considered as two "trains" of references moving across the cache. Therefore, the larger W_B, the higher the probability that these "trains" meet the area of alive elements of X, i.e the higher the probability of cross-interferences. Still, these cross-interference misses account for a relatively small share of total misses, unless L_S is large, i.e there are few *intrinsic misses* (cf. figure 18).

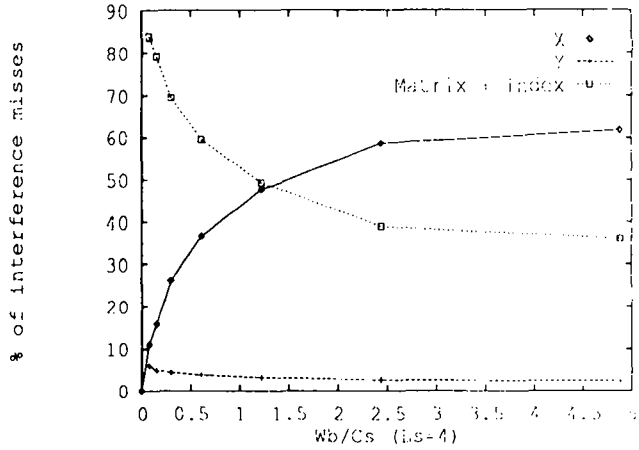


Figure 18: *% of interference misses for different values of $\frac{W_B}{C_S}$.*

Current values of $\frac{W_B}{C_S}$ Let us now try to see what are the values of $\frac{W_B}{C_S}$ currently found.

The size of numerical problems tends to grow, and consequently W_B grows accordingly. When *bandwidth reduction* renumbering algorithms are employed, $W_B \simeq \sqrt{N}$ *or* $W_B \simeq N^{\frac{2}{3}}$ *or* $W_B \simeq \frac{N}{10}$ according to the problem type, while $W_B \simeq N$ when *minimum-degree* renumbering algorithms are used [7]. Large current problem sizes range from $N = 10^5$ to $N = 10^6$, therefore W_B generally varies between $W_B = 300$ and $W_B = 10^6$ [7].

For single-cache machines, C_S clearly tends to grow. C_S values of more than *256 Kbytes* can now be found [17]. So, if cache sizes increase fast enough $\frac{W_B}{C_S}$ will soon be on the "safe zone" (i.e $\frac{W_B}{C_S} \ll 1$; *256 Kbytes* cache, $N = 10^5$, $W_B = N^{\frac{2}{3}}$, $\frac{W_B}{C_S} = 0.01$). On single-cache machines equipped with current-size caches (i.e, $C_S \simeq 4192$ or *16 Kbytes*), a large third-dimensional problem (i.e, $N \simeq 10^5$, $W_B \simeq N^{\frac{2}{3}}$ to $W_B = N$) exhibits a $\frac{W_B}{C_S}$ ratio of *0.5* to *3.5*.

The increasing popularity of multi-level caches makes small (primary) caches more frequent. The size of such caches is currently of the order of *4 Kbytes* [17]. What is more, N also tends to grow, and *minimum degree* is a rather popular algorithm (cf. [11]). So very large ratios $\frac{W_B}{C_S}$ may

become more frequent (*4 Kbytes* cache, $N = 10^5$, $W_B = N$, $\frac{W_B}{C_S} = 100$).

4.3 Density $d = \frac{n_{nz}}{W_B}$

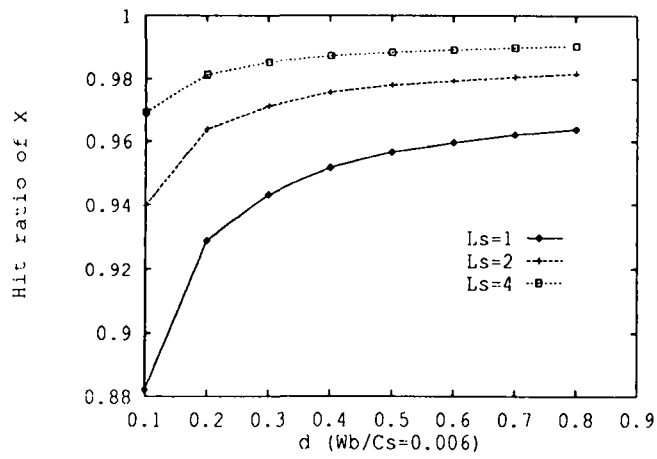


Figure 19: *Influence of the density on the hit ratio of X for* $\frac{W_B}{C_S} < 1$.

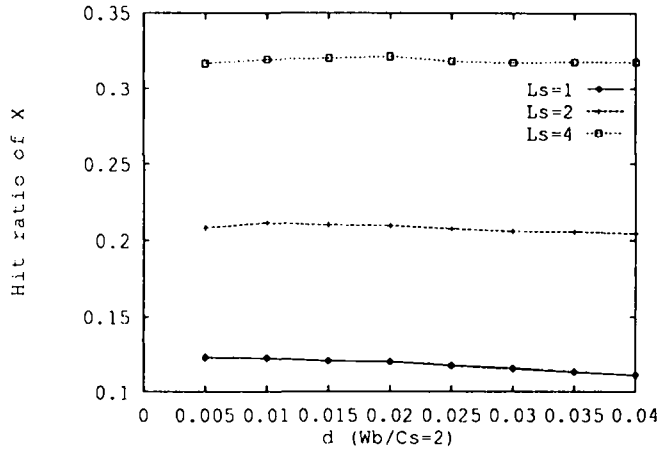


Figure 20: *Influence of the density on the hit ratio of X for* $\frac{W_B}{C_S} > 1$.

Let us now consider parameter $d = \frac{n_{nz}}{W_B}$. d can be considered as the "density" of non-zero elements on one row of matrix A. When d is very high matrix A looks very much like a banded *dense* matrix, while the matrix is "very sparse" when d is relatively small.

Depending on w, the density of non-zero elements induces two different phenomenons.

If $w < 1$, the smaller W_B (i.e the larger d) the less cross-interferences occur (cf. paragraph 4.2). Consequently, it is possible to benefit from the temporal locality of references to X (cf. graph $L_S = 1$ of figure 19). For the same reason, it is also possible to benefit from spatial locality. Indeed, once an element of X is loaded $L_S - 1$ consecutive elements of the same array are also loaded. Though they are not immediately referenced, they are not flushed from cache (as seen above), and therefore they stay into cache until they are referenced. That is why array X also benefits from spatial locality in this case, independently of the distribution of non-zero elements (cf. paragraph 4.1 and figure 19).

Now it can be observed that, for a fixed value of W_B, *when* n_{nz} *is large there exists an important potential reuse on* X. Since it is possible to benefit from temporal locality when $w < 1$, the higher n_{nz} (i.e the larger d) the higher the hit ratio of X (cf. figure 19). Nevertheless, a high value of n_{nz} slightly worsens cache pollution, though n_{nz} must be quite large for this phenomenon to counterbalance the benefits from spatial and temporal locality (cf. figure 19).

When $w > 1$, the behavior of X is not correlated to d anymore because interferences (cross and self) occur so often that benefiting from temporal and spatial locality becomes hypothetical (cf. figure 20).

Usual values of d For 2-dimensional finite-element problems, the average number of non-zero elements per row is of the order of *10*, while it is of the order of *100* for 3-dimensional problems [7]. *So essentially 3-dimensional problems are worth optimizing.* As it has been seen in paragraph 4.2, W_B ranges from $\sqrt{N}$ (2-dimensional) or $N^{\frac{2}{3}}$ (3-dimensional) to N. Typically, the density of a 3-dimensional problem may range from *0.001* to *1*.

4.4 Set-associative caches

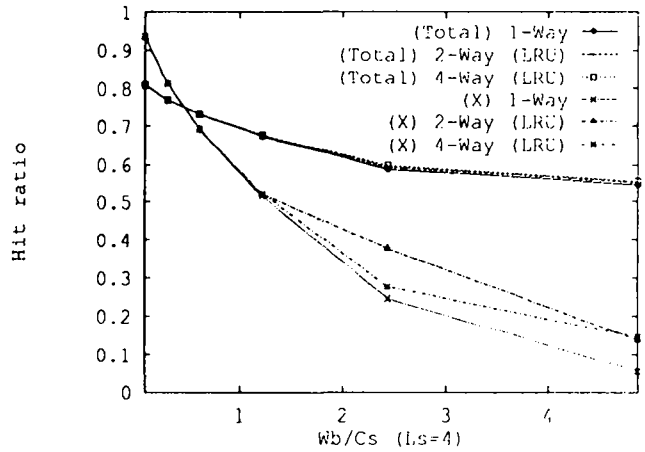


Figure 21: *Performance comparison of set-associative and direct-mapped caches for different values of* w

Though the model presented in section 3 corresponds to direct-mapped caches, it can be extended to set-associative caches (cf. section A). Nevertheless, simulations can already show that associativity brings little improvements on total hit ratio, and more particularly on the hit ratio of X. Basically associativity is helpful when interferences occur. Now, when $w < 1$, there are very few interferences, and when $w > 1$, interferences are so numerous (at least w elements of X conflict for the same cache location) that a

2-way or 4-way associativity brings little or no improvement (cf. figure 21).

4.5 Synthesizing the analysis

In most applications there is a tradeoff related to L_S. When L_S is too large, cache pollution becomes important and threatens further gains. For SpMxV again, the tradeoff exists. Increasing L_S worsens cross and self-interferences. What is more it has been noted that only small line sizes (typically $L_S = 2$ *or* 4) are necessary to take into account the particular locality of *finite-element* matrices. However, in SpMxV, intrinsic misses account for the major part of cache misses, and therefore even very large line sizes do bring substantial improvements. However, the reduction is hyperbolic, and therefore there is again a threshold beyond which little improvements can be expected (typically $L_S \geq 16$). Anyway, as long as intrinsic misses account for a large fraction of total misses, i.e as long as L_S is small (typically, $L_S \leq 2$), decreasing the miss ratio of X does not bring significant improvements on the total hit ratio.

The degree of interference $w = \frac{W_B}{C_S}$ strongly influences the hit ratio of X. When $w < 1$, there are no self-interference misses, and therefore high total hit ratio can be achieved if L_S is large. If L_S is small (typically $L_S \leq 4$), the percentage of cross-interference misses is far smaller than intrinsic misses, though may significantly increase for large values of L_S. Reducing W_B also decreases the number of cross-interferences. When $w \simeq 1$, the number of self-interferences grows hyperbolically with w, and therefore significant gains on the hit ratio of X can be expected through bandwidth reduction. When $w \gg 1$ (typically $w \geq 10$), the number of self and cross interferences are redundant and close to maximum. If L_S is large, misses of X account for the major part of total misses.

When $w < 1$, the higher the density of non-zero elements $d = \frac{n_{nz}}{W_B}$, the better the hit ratio of X. When $w > 1$, it is hardly possible to make use of the locality of X, and therefore the density does not influence much the hit ratio of X.

Finally, simulations tend to show that, in most cases, associativity brings little improvements on the hit ratio of X and the total hit ratio.

5 Improving the behavior of SpMxV

In this section, possible software and hardware optimizations are discussed. Two different approaches for software optimization of SpMxV are distinguished: an algorithmic approach which aims at reducing bandwidth using renumbering algorithms, and a software approach based on particular blocking techniques. For hardware optimizations, the ways to reduce misses corresponding to regular and non-regular references are considered.

5.1 Software optimization: *Bandwidth reduction*

Two main kinds of renumbering algorithms are employed: *bandwidth reduction* algorithms which are derivatives of that of Cuthill and McKee, and the *minimum-degree* algorithm [11]. According to George [11], reducing bandwidth is not closely related to minimizing arithmetic operations and storage. On the other hand, *minimum-degree* algorithm is efficient for finding low-fill orderings. Therefore, this second renumbering scheme tends to become popular. However, it must be noted that an effect of *minimum degree* is to scatter non-zero elements across the matrix (cf. figure 22), while bandwidth reduction algorithms are generally very efficient in grouping non-zero elements (cf. figure 22). So, if *minimum degree* is more efficient for LU factorization as shown by George, it is far less profitable for SpMxV in terms of locality, because it widens considerably matrix bandwidth.

Both LU factorization and SpMxV often occur in the same program. Since LU factorization is far more costly then SpMxV, optimizing renumbering for this operation would be reasonnable (i.e sacrificing bandwidth). However, for a given matrix, SpMxV is often executed several times, vector X being the only data changed on each iteration. Therefore, it is after all profitable to make a copy of the matrix and renumber it for SpMxV only, since the cost of the renumbering process is not prohibitive (of the order of one execution of SpMxV). It should also be kept in mind, that renumbering in order to minimize bandwidth is profitable in our case, only if the degree of interference w can be made smaller than 1.

5.2 Software optimization: *Blocking*

5.2.1 Classic blocking

The reason why sparse codes do not work on caches when matrix parameters are much larger than cache parameters is the same as for dense codes: because elements to be reused cannot be kept in cache. However, a solution valid for dense codes [5], i.e blocking, is not valid for sparse codes, because each element is not reused a sufficient number of times to override the overhead of blocking.

Moreover, sparse codes exhibit an *irregular locality* which cannot be forseen at compile-time (i.e *a priori*), while dense codes exhibit *regular locality*, which can be exploited at compile-time. If classic blocking techniques were to be used for SpMxV, part of its *irregular locality* would probably be lost, and large overhead data would be added.

Let us try to explain and formalize these latter notions. Consider original matrix A is blocked horizontally and vertically, using rectangular blocks. One block has B_Y rows and B_X columns. Therefore, if each block is stored row-wise, B_Y integers must be stored indicating the number of non-zero elements on each row. B_Y rows of the matrix are divided into $\frac{W_B}{B_X}$ verical blocks. Since, there are $\frac{N}{B_Y}$ blocks of size B_Y, $\frac{N \times W_B}{B_Y \times B_X}$ integer elements must be stored for this blocked algorithm (while only N elements are needed for the original non-blocked algorithm).

Now, there are two ways to execute this blocked SpMxV: row-wise or column-wise. If SpMxV is executed column-

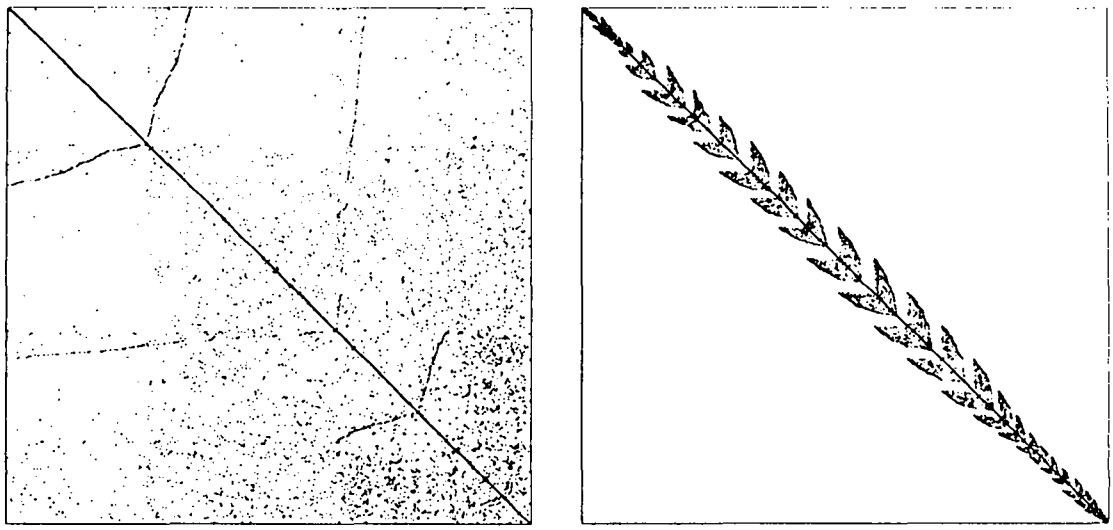


Figure 22: *Matrix BCSPWR09 of the Harwell-Boeing suite* after application of minimum degree algorithm *and* after application of a bandwidth reduction algorithm.

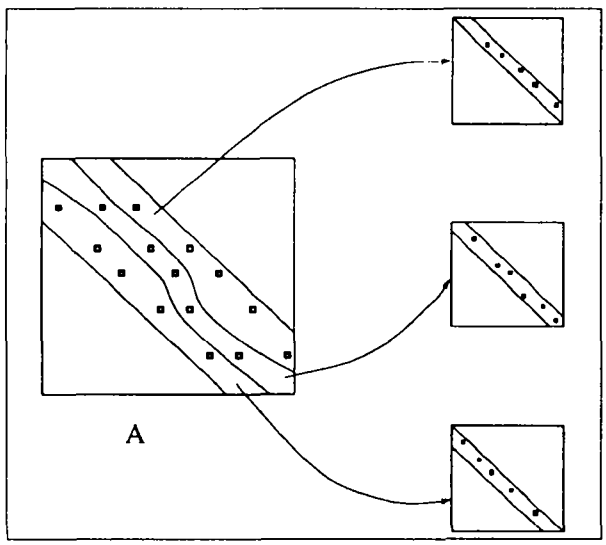


Figure 23: *Splitting a sparse matrix with a large band into several submatrices with a small band.*

wise, then elements of Y probably cannot be kept into cache from one column to another. Consequently, the number of cache misses due to Y is multiplied by $\frac{N}{B_X}$. If SpMxV is executed row-wise, then the reuse of X is degraded in turn. However, since X does not exhibit the same flawless spatial and temporal locality as Y, then it is preferable to degrade the reuse of X rather than that of Y, i.e to execute SpMxV row-wise.

Since on one column of a block $B_X \times B_Y$ there are averagely $d \times B_Y$ non-zero elements (where $d = \frac{n_{nz}}{W_B}$ is the density of the sparse matrix), then in the best case, the miss ratio of X can be divided by a factor of $d \times B_Y$. In order to have at least *2* non-zero elements per column of a block (i.e that an element of X is reused at least twice per block), it is necessary to have $\frac{n_{nz}B_Y}{W_B} \geq 2$ for blocking to be profitable. Since at most $B_Y = C_S$, the minimal condition for benefiting from rectangular blocking is $\frac{n_{nz}C_S}{W_B} \geq 2$. Actually, because of the $\frac{N^2}{B_Y \times B_X}$ new intrinsic misses, it is necessary to have $\frac{n_{nz}C_S}{W_B} \gg 2$ in order to get any improvement.

Therefore, in most cases classic blocking of sparse matrices degrades the hit ratio of Y, and brings little improvements on that of X (cf. figure 25; the optimal block sizes were found experimentally: $C_S = 1024$, $B_X = 100$, $B_Y \simeq 25$). Moreover, significant overhead data worsens the total hit ratio by addition of intrinsic misses (cf. figure 25). Only if the matrix is nearly dense ($d \simeq 1$), classic blocking is profitable due to the important potential reuse per element of X.

5.2.2 Blocking by diagonal

It is clear that any software optimization techniques should take into account the specifics of SpMxV. First of all, in opposition to most dense codes, it must be assimilated that one computation corresponds to one *non-zero* element and not to any element of the matrix. Therefore, any blocking technique should implicitly deal with *blocks of non-zero elements* rather than *blocks of matrix elements*.

Second of all, in dense matrices, the elements of symmetry are *columns and rows*, which explain why such matrices are blocked using rectangular shapes. The elements of symmetry of banded sparse matrices are *diagonals*. Therefore, it seems natural to block along diagonals, otherwise too many blocks would be half-empty and would degrade the efficiency of the blocking technique.

According to section 4.2, the original large band should be split into several small bands, such that their width is of the order of cache size (cf. figure 23). Now, these blocks should be based on non-zero elements and not on arbitrary geometric dissection. Moreover, having the number of non-zero elements on each of the different diagonals constant (except for the first and last diagonal blocks) would reduce the overhead data to be kept, as seen in above paragraph.

For that purpose, n_{nz} the average number of non-zero elements per row and the number of non-zero elements on each

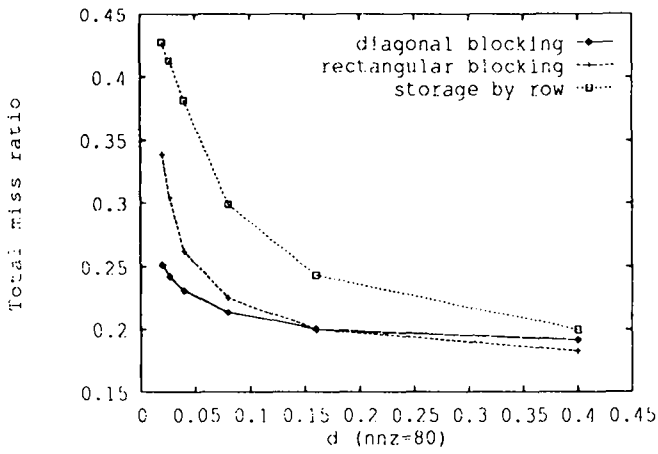


Figure 24: *Effect of diagonal blocking on the total miss ratio.*

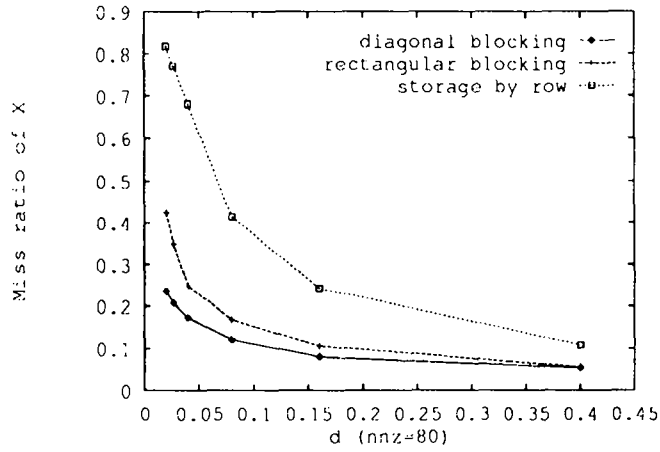


Figure 25: *Effect of diagonal blocking on the miss ratio of* X.

diagonal of the original matrix must be computed. Since the goal of the method is to obtain a collection of banded matrices which bandwith is smaller than cache size, the original matrix needs to be split into approximately $n_B = \lceil \frac{W_B}{C_S} \rceil$ submatrices. In all these submatrices except for the first and last one, the number of non-zero elements ought to be constant, approximately equal to $\frac{n_{nz}}{n_B}$. Additional data is necessary to store the number of non-zero elements on the rows of the first and last diagonal block. The corresponding code can be seen on figure 26.

This blocking method is interesting only when X is the major cause of cache misses, and the potential reuse is high (cf. sections 4.1 and 4.3). Though it increases the number of misses on Y (they are nearly multiplied by n_B), it decreases the miss ratio of X, so that the improvement of the total hit ratio may be quite important (cf. figure 25). The main asset of the method is to be applicable even when the density is low, where other classic blocking methods would fail.

Moreover, the shape of certain types of matrices such as finite-element matrices suggests that diagonal blocking could be further enhanced. Indeed, in such matrices, there are generally three diagonals along which non-zero elements are grouped (cf. figure 6). Therefore a first step to diagonal blocking would be to find the diagonals of "highest density" and block non-zero elements around them. Not only, nearly dense blocks of non-zero elements would be obtained, but the number of blocks itself (and the overhead) would be considerably reduced in many cases, since it would be fixed (and would not depend on W_B and C_S).

However, all blocking techniques introduce bounds of their own. In our case, the number of blocks determine the maximum reuse per element of X. If there are n_B blocks and n_{nz} non-zero elements per row, then *diagonal blocking* authorizes a maximum reuse of $\frac{n_{nz}}{n_B}$ per element of X, while the theoretical maximum is n_{nz}.

5.2.3 Parallelization

Rather than making use of data locality, supercomputers propose to speed up sparse computations by vectorizing accesses to data stored in compress format using hardware Scatter/Gather [14]. For shared-memory multiprocessors with a memory hierarchy, the issue is to parallelize sparse codes while optimizing load balancing and communications between global and local memories. Actually, little information is known on the behavior of such architectures under sparse workloads. Saad and al. [20, 21] proposed a benchmark suite for evaluating the performance of sparse codes, and experimentally analyzing their locality and inherent parallelism. Yang [26] presented a set of parallelized sparse primitives, where parallelism is extracted at the algorithmic level. Davis [1] also proposed a parallel implementation of sparse Gaussian elimination based on an algorithmic transformation, and studied slightly the impact of his scheme on communications between global and local memories.

However, little effort has been devoted to analyzing and modeling the impact of parallelization on the behavior of sparse codes on a memory hierarchy. Now, for SpMxV, distributing tasks to each processor is equivalent to splitting original matrix into blocks of certain size and shape. Consequently, each subproblem assigned to a local cache is equivalent to original problem though with smaller parameters, and therefore, the whole previous analysis can be applied to each subproblem separately.

This property can be used to determine the impact of blocking on communications. Actually, it may appear that the most natural and most commonly employed methods for blocking sparse matrices (e.g. blocking by rows) may not be the best regarding communications between the different levels of memory. For instance, the original *storage-by-row* scheme already allows an easy parallelization of SpMxV (cf. figure 5) on loop I. However, analysis of section 4 shows that a large bandwidth breeds an important miss ratio of X. Now, if SpMxV is blocked on loop I and the bandwith of original matrix is large, then the bandwidth of the resulting submatrices remains the same. Therefore, the miss ratio of X on each local cache may be high.

On the other hand, *blocking by diagonal* has several assets which can be profitable to parallelization *and* locality also, especially by improving data reuse without bringing significant overheads. Since the original problem (a banded sparse

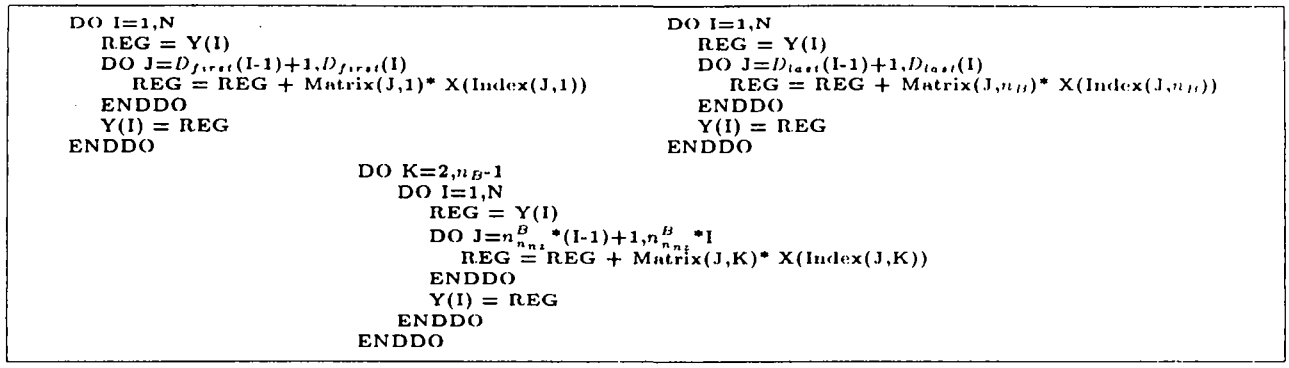


Figure 26: *Blocking by diagonal.*

matrix) is split diagonally, the submatrices bandwidth is smaller than that of original matrix (actually it is equal to that of original matrix divided by the number of blocks). Therefore, on each local cache the miss ratio of X may be far smaller than when blocking by row. Consequently, parallelizing this way naturally *decreases the burden on local caches* by reducing the bandwidth, and consequently the hit ratio.

Another asset of this blocking method is that, once the original problem has been divided into subproblems, it is possible to apply any storage method for these subproblems, not inevitably *storage by row*. Now, *jagged diagonal* [3] and *generalized columns* [9] are among the most efficient storage techniques for vector execution. Therefore, if a tradeoff between parallelization, vectorization and data locality must be found, blocking by diagonal may ensure non negligeable data locality while still authorizing vectorization (on each subproblem separately).

5.3 Hardware optimizations

5.3.1 Regular references

In a number of cases, *intrinsic misses* account for the major part of cache misses (cf. section 4.5). These misses are due to arrays exhibiting flawless spatial locality and deprived of temporal locality. Some processors are now equipped with *pipelined bypass* [13] which are well suited for loading arrays with high spatial locality, and which also avoid cache pollution with arrays deprived of temporal locality by not storing them into cache. Otherwise, this problem can also be solved with many techniques already proposed for regular codes. Large line sizes [23] provide a simple solution for reducing the impact of such misses. Jouppi also presented a *Multi-Way Streamed Buffer* [15] which aims at removing completely intrinsic misses. Inspite of all, whenever *intrinsic misses* account for a minor part of total cache misses, self-interference misses due to array X become the main issue.

5.3.2 Irregular references

Little solutions have been implemented in the industry for dealing with sparse codes. The most classic and efficient one is the *scatter-gather* implemented on many vector supercomputers [14]. However, this feature aims at vectorizing non-regular accesses to memory rather than optimizing the reuse of elements accessed. Other original solutions have been proposed, but up to now few resulted in actual machines or at least few published results for the moment. Seznec and al. [22] propose an interconnection network that would behave efficiently under sparse code workloads. Ibbet and al. [12] intend to implement the *Edimburgh Sparse Processor*, an architecture dedicated to *finite-element problems*. Wolfe and al. [25] work on a coprocessor *The White Dwarf* implementing specific sparse primitives. Finally, Amano and al. [2] presents the *Sparse Matrix Solving Machine* which is a parallel computer using a memory design adapted to sparse computations.

Most of the previous projects have not yet been implemented or at least few performance results have been published. Actually, the majority of these projects concern dedicated machines rather than solutions for optimizing and adaptating current memory hierarchies to sparse computations. Let us consider which design would be the most convenient for these applications. When $w \gg 1$, interferences are so numerous that exploiting locality within SpMxV becomes impossible in a usual cache (cf. section 4.5). However, if it were possible to determine placement of data within cache (as in local memory), many interferences would be avoided. First, only one memory location is actually required for arrays D, Y, *Index* and *Matrix*, and the rest can be devoted to array X (thereby removing cross-interferences). Second, the pattern of references to X is known *a priori*, so if data replacement can be controlled (as in local memory), optimal replacement could be achieved, as shown in [4] (thereby minimizing self-interference misses). On the other hand, when $w < 1$, cache makes a very good job at exploiting spatial and temporal locality without any optimization (i.e no overhead) required (cf. section 4.5). So depending on problem parameters SpMxV would require the upper-level memory to have the properties of a *local memory* or of a *cache*, in other terms of a hybrid memory system (cf. [6]).

6 Conclusions

This paper presents a methodology for modeling the irregular references of sparse codes using probabilistic methods. The model was shown to be very accurate for uniformly distributed matrices, *finite-element* matrices renumbered with *minimum degree algorithm*, and still reflects quite well the

behavior of *finite-element* matrices renumbered with *bandwidth reduction algorithm*.

The analysis of the model and simulations allowed to identify three main parameters and their impact on the behavior of SpMxV. First of all, cache size and bandwidth are closely dependent. When bandwidth is smaller than cache, spatial and temporal locality of sparse matrices is well exploited and SpMxV needs not be optimized. Second, when bandwidth is greater than cache size, self and cross-interferences degrade the reuse of vector X which cannot exploit its temporal and spatial locality. Moreover, in that case little optimizations can be expected if the line size is small because *intrinsic misses* are too numerous anyway. However, when line size is sufficiently large, then exploiting the potential locality of array X may yield significant improvements of the total hit ratio, especially in 3-dimensional *finite-element* problems where the potential reuse per element of X is important.

Little hardware or software techniques exist for making use of locality within sparse problems. First, *Bandwidth reduction renumbering algorithms* may significantly improve the exploitation of locality within SpMxV, but due to LU factorization *minimum degree renumbering algorithms* (which widen considerably bandwidth) are generally preferred. Therefore, there is a tradeoff between the two algorithms which can actually be resolved through copying. Second, blocking methods are considered. Classic rectangular blocking is proved to be efficient only when the matrix is nearly dense within its band, otherwise the method breeds too much overhead. A blocking technique *blocking by diagonal* that takes into account the specifics of sparse codes has been presented. It is shown to efficiently exploit locality where other blocking methods would fail, i.e when the matrix is "very sparse". Moreover, the parallel version of the diagonally blocked algorithm on a multi-cache system would naturally reduce the burden on local caches.

7 Acknowledgements

We would like to thank J. Erhel and A. Seznec for their helpful comments and many enlightening discussions.

References

[1] S. G. Abraham and T. A. Davis: *Blocking for parallel sparse linear system solvers*, Proc. of Int. Conf. on Parallel Processing, 1989.

[2] H. Amano, T. Yoshida and H. Aiso: $(SM)^2$: *Sparse matrix solving machine*, Proc. of Int. Symp. on Computer Architecture, 1983.

[3] E. Anderson and Y. Saad: *Solving sparse triangular systems on parallel computers*, Int. J. of High Speed Computing, 1 (1989), pp. 73-95.

[4] L. A. Belady: *A study of replacement algorithms for a virtual-storage computer*, IBM Systems Journal, Vol. 5, No. 2, pp. 78-101, 1966.

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

[6] G. D. McNiven and E. S. Davidson: *Analysis of memory referencing behavior for design of local memories*, Proc. of 14th Annual IEEE Int. Symp. on Computer Architecture, ACM, 1988.

[7] I. Duff, A. Erisman and J. K. Reid: *Direct methods for Sparse matrices*, Oxford University Press, Oxford, England, 1987.

[8] I. Duff, R. Grimes, J. Lewis: *Sparse matrix test problems*, ACM TOMS, 15 (1989), pp. 55-64.

[9] J. Erhel: *Sparse matrix multiplication on vector computers*, Int. J. of High Speed Computing, Vol. 2, No. 2, pp. 101-116, 1990.

[10] C. Fricker, P. Robert: *An analytical cache model*, INRIA Report, INRIA Rocquencourt, France, July 1991.

[11] A. George: *Direct solution of sparse positive definite systems: some basic ideas and open problems*, in Sparse Matrices and Their Uses, edited by I. S. Duff, Academic Press, 1981.

[12] R. N. Ibbet, T. M. Hopkins and K. I. M. McKinnon: *Architectural mechanisms to support sparse vector processing*, Proc. of Int. Symp. on Computer Architecture, 1989.

[13] Intel Corporation: *Intel i860 reference manual*, 1991.

[14] J. G. Lewis and H. D. Simon: *The impact of hardware Scatter/Gather on sparse Gaussian elimination*, Proc. of Int. Conf. on Parallel Processing, 1986.

[15] Norman P. Jouppi: *Improving cache replacement by the addition of a small fully-associative cache and prefetch buffers*, IEEE, 1990.

[16] Monica S. Lam, Edward E. Rothberg and Michael E. Wolf: *The cache performance and optimizations of blocked algorithms*, Proc. 4th International conference on Architectural Support fo Programming Languages and Operating Systems, pp. 63-74, April 1991.

[17] D. A. Patterson and J. L. Hennessy: *Computer architecture: a software approach*, Morgan Kaufmann Publishers, Palo Alto, 1990.

[18] S. Przybylski, M. Horowitz and J. Henessy: *Performances tradeoffs in cache design*, Proc. of 15th Annual IEEE Int. Symp. on Computer Architecture, ACM, 1989.

[19] E. Rothberg and A. Gupta: *Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations*, Proc. of ACM Supercomputing 90.

[20] Y. Saad and H. A. G. Wijshoff: *A benchmark package for sparse matrix computations*, Proc. of Int. Conf. on Supercomputing, 1988.

[21] Y. Saad and H. A. G. Wijshoff: *SPARK: A benchmark package for sparse computations*, Proc. of Int. Conf. on Supercomputing, 1990.

[22] A. Seznec and Y. Jegou: *Synchronizing processors through memory requests in a tightly coupled multiprocessor*, Proc. of Int. Symp. on Computer Architecture, 1988.

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

[24] H.(Harry) A.(Arnold) G.(Gwendoline) Wijshoff: *Implementing Sparse BLAS primitives on concurrent/vector processors: a case study* CSRD Report No. 843, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, January 1989.

[25] A. Wolfe, M. Breternitz Jr., C. Stephens, A. L. Ting, D. B. Kirk, R. P. Bianchini Jr., J. P. Shen: *The White Dwarf: A high-performance application-specific processor*, Proc. of 14th Annual IEEE Int. Symp. on Computer Architecture, ACM, 1988.

[26] G. C. Yang: *DSPACK: A parallel direct sparse matrix package for shared-memory multiprocessors*, Proc. of Int. Conf. on Parallel Processing, 1990.

A Appendix: Computing the number of cache misses due to array X

A.1 Methodology of resolution of the basic problem

In the following paragraphs, we indicate the method for computing the expression of $\pi^k_{out}(i,j)$

State vector For each element k of X'', we define the *state vector* $\pi^k(i,j)$ as follows:

$$\pi^k(i,j) = \begin{pmatrix} \pi^k_{in}(i,j) \\ \pi^k_{out}(i,j) \end{pmatrix}$$

where $\pi^k_{in}(i,j)$ is the *probability that element k of X'' is in the 1-size cache on row i of the computation, right before column k is considered.* And, $\pi^k_{out}(i,j) = 1 - \pi^k_{in}(i,j)$.

Probability of transition Let us now try to express $\pi^k(i,j+1)$ as a function of $\pi^k(i,j)$, that is, find a probability of transition matrix that could help describing this event.

Case k=j: $\pi^j(i,j)$ describes the probability of presence in cache of element j of X'', on row i of the computation, before column j is considered. Now, when column j is considered, element j of X'' has a probability $p(i,j)$ of being referenced, that is a probability $p(i,j)$ of being loaded into the cache if it is not already there. This notion can be formalized by the following probability of transition matrix:

$$p_{load}(i,j) = \begin{array}{c} \\ in \\ out \end{array} \begin{array}{c} \begin{array}{cc} in & out \end{array} \\ \begin{pmatrix} 1 & p(i,j) \\ 0 & 1-p(i,j) \end{pmatrix} \end{array}$$

Therefore, we can now write formally that

$$\pi^k(i,j+1) = p_{load}(i,j) \times \pi^k(i,j)$$

Case $k \neq j$: if column $k \neq j$, then column k has a probability $p(i,k)$ of breeding a reference to element k of X, which would then flush element j of X from the cache. This notion can be formalized by the following probability of transition matrix:

$$P_{flush}(i,j) = \begin{array}{c} \\ in \\ out \end{array} \begin{array}{c} \begin{array}{cc} in & out \end{array} \\ \begin{pmatrix} 1-p(i,j) & 0 \\ p(i,j) & 1 \end{pmatrix} \end{array}$$

Transition from row $i-1$ to row i: Therefore, we can now express $\pi^j(i,j)$ as a function of $\pi^j(i-1,j)$:

$$\begin{aligned} &\pi^j(i,j) = \\ &P_{flush}(i,j-1) \times \ldots \times P_{flush}(i,1) \\ &\times P_{flush}(i-1,n_c) \ldots \times P_{flush}(i-1,j+1) \\ &\times P_{load}(i-1,j)\pi^j(i-1,j) \end{aligned}$$

Let us define $P_{(i-1,j)\rightarrow(i,j)}$ as follows:

$$\begin{aligned} &P_{(i-1,j)\rightarrow(i,j)} = \\ &P_{flush}(i,j-1) \times \ldots \times P_{flush}(i,1) \\ &\times P_{flush}(i-1,n_c) \ldots \times P_{flush}(i-1,j+1) \\ &\times P_{load}(i-1,j) \end{aligned}$$

then $\pi^j(i,j) = P_{(i-1,j)\rightarrow(i,j)}\pi^j(i-1,j)$. Recursively, we can then write that $\pi^j(i,j) = P_{(0,j)\rightarrow(i-1,j)}\pi^j(0,j) = \prod_{l=1}^{i-1} P_{(l-1,j)\rightarrow(l,j)}\pi^j(0,j)$.

A.2 Computing the number of cache misses

It is a fair approximation to assume that, at the beginning of each subsection, no element is in cache. Therefore, we write that

$$\pi^j(0,j) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \forall j \in [1, n_c]$$

therefore, $\pi^j_{out}(i,j) = P_{(0,j)\rightarrow(i-1,j)}(2,2)$, and it is now possible to compute $N_{cm}(P'')$:

$$N_{cm}(P'') = \sum_{i=1}^{n_l} \sum_{j=1}^{n_c} \pi^j_{out}(i,j) \times p(i,j)$$

It is now possible to give an approximation of the total number of cache misses yielded by A. The number of cache misses due to a section S' of A' is given by $N_{cm}(S') = N_{cm}(P''_1) + N_{cm}(P''_2)$. Since there are $\frac{N}{C_S}$ $N_{cm}(S') = N \times N_{cm}(S')$. Now, since there are C_S such submatrices in A, the total number of cache misses is equal to $N_{cm}(P) = C_S N_{cm}(P')$.

$$N_{cm}(P) \simeq \frac{N}{L_S}(n_c^1 \times n_l^1 + n_c^2 \times n_l^2)p - \frac{1}{2}\frac{N}{L_S} \times \left(n_c^1 \times n_l^1 \times (n_l^1 - 1) + n_c^2 \times n_l^2 \times (n_l^2 - 1)\right) \times p^2$$

Figure 27: *Expression of the number of self-interference misses.*

$$N_{cm}(P) \simeq n_{nz} - \frac{1}{2}\frac{L_S \times n_{nz}^2 \times C_S^2}{N \times W_B^2} \times \left(\lfloor\frac{W_B}{C_S}\rfloor - 4\frac{W_B}{C_S}\lfloor\frac{W_B}{C_S}\rfloor + 3(\lfloor\frac{W_B}{C_S}\rfloor)^2 + 2\lfloor\frac{W_B}{C_S}\rfloor(\frac{W_B}{C_S})^2 - 4\frac{W_B}{C_S}(\lfloor\frac{W_B}{C_S}\rfloor)^2 + 2(\lfloor\frac{W_B}{C_S}\rfloor)^3 + (\frac{W_B}{C_S})^2 - \frac{1}{C_S}\frac{W_B}{C_S}\right)$$

Figure 28: *Expression of the number of self-interference misses as a function of the problem parameters.*

A.3 Distributions of probability

Up to now we have considered that each element (i, j) of A had a specific probability $p(i, j)$ of being a non-zero element. This distribution can be non-uniform, i.e p depends effectively on (i, j), or uniform, i.e $p(i, j) = p$. The non-uniform distribution allows a better description of the number of cache misses, but prevents, in most cases, the computation of an analytical expression of this number. The uniform distribution, though less precise, allows to obtain such an expression.

Non uniform distribution Certain types of sparse matrices cannot be considered to have a uniform distribution within the band. However, in many applications, specifically *finite element methods* it is a fair approximation to consider that the distribution of probability is constant across the diagonals, that is $p(i, j) = p(i-1, j-1)$. This is due to a good repartition of the non-zero elements among the rows. Indeed, most of the time, the number of non-zero elements per row is constant. And, what is more the distribution of the non-zero elements is relatively similar on all the rows.

Let us now give the expression of the total number of cache misses when the distribution is non-uniform:

$$\begin{aligned} &N_{cm}(P) = \\ &N \times \left[\sum_{i=1}^{W_B \bmod C_S} \sum_{j=1}^{\lfloor\frac{W_B}{C_S}\rfloor+1} \pi_{out}^j(i,j) \times p(i,j)\right. \\ &\left. + \sum_{i=1}^{W_B - W_B \bmod C_S} \sum_{j=1}^{\lfloor\frac{W_B}{C_S}\rfloor} \pi_{out}^j(i,j) \times p(i,j)\right] \end{aligned}$$

Therefore, when the distribution is not uniform, the two sums

$$\sum_{i=1}^{W_B \bmod C_S} \sum_{j=1}^{\lfloor\frac{W_B}{C_S}\rfloor+1} \pi_{out}^j(i,j) \times p(j)$$

and

$$\sum_{i=1}^{W_B - W_B \bmod C_S} \sum_{j=1}^{\lfloor\frac{W_B}{C_S}\rfloor} \pi_{out}^j(i,j) \times p(j)$$

must still be computed numerically (this computation is not too costly considering the boundaries of the sums).

Uniform distribution For some sparse applications, it is a reasonnable approximation to consider that the distribution of the non-zero elements is uniform. In that case an analytical expression of the number of cache misses can be derived from the previous formula. Let us show how.

First of all, we compute the uniform probability p that an element be non-zero. There are n_{nz} non-zero elements in the matrix. Since the number of non-zero elements on each row is constant, we can consider that there is an average number of $\frac{n_{nz}}{N}$ elements per row. Therefore each element has a probability $p = \frac{\frac{n_{nz}}{N}}{W_B}$ of being referenced. When L_S is taken into account, the probability that a group of L_S elements of X be referenced is $p = 1 - (1 - \frac{\frac{n_{nz}}{N}}{W_B})^{L_S}$.

Let us now give the expression of $P_{(i-1,j)\to(i,j)}$:

$$\begin{aligned} p(l,k) &= p \\ \Rightarrow P_{flush}(l,k) &= \begin{pmatrix} 1-p & 0 \\ p & 1 \end{pmatrix} \end{aligned}$$

and,

$$P_{load}(l,k) = \begin{pmatrix} 1 & p \\ 0 & 1-p \end{pmatrix}$$

Then,

$$\begin{aligned} P_{(i-1,j)\to(i,j)} &= (P_{flush})^{n_c-1} \times P_{load} \\ \Rightarrow P_{(0,j)\to(i,j)} &= ((P_{flush})^{n_c-1} \times P_{load})^i \end{aligned}$$

And,

$$\pi^j(i,j) = ((P_{flush})^{n_c-1} \times P_{load})^i \pi^j(0,j)$$

The explicit expression of $P_{(0,j)\to(i,j)}$ is the following:

$$\begin{pmatrix} \cdot & \cdot \\ \frac{1}{1-(1-p)^{n_c}} & [1-(1-p)^{n_c-1} \\ \cdot & + p(1-p)^{n_c-1}(1-p)^{n_c(i-1)}] \end{pmatrix}$$

We can note that $\pi^j(i,j)$ is now independent of j. What is more, we will consider, without a great loss of precision that:

$$\pi^j(0,j) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Therefore, using notations of section 3.1.2, we can deduce that the total number of cache misses is given by expression of figure A or figure 10.

Now, since $\frac{n_{nz}}{N \times W_B} \ll 1$, we can write that

$p \simeq L_S \times \frac{n_{nz}}{N \times W_B} - L_S(L_S - 1) \times \frac{n_{nz}^2}{N^2 \times W_B^2} + o(\frac{n_{nz}}{N \times W_B})$

Therefore, a first order development of expression of $N_{cm}(P)$ (as a function of the problem parameters) is given by expression of figure A.

For sake of simplicity, we can assume that $\lfloor\frac{W_B}{C_S}\rfloor \simeq \frac{W_B}{C_S}$. and therefore:

$$N_{cm}(P) \simeq n_{nz} - \frac{n_{nz}^2 \times C_S}{2 \times N \times L_S \times W_B}$$

A.4 Extending the model to set-associative caches

The problem formulation for set-associative caches is very similar to the one of direct-mapped caches except that, in the simplified problem, the cache size is now equal to $D_A \times L_S$ (D_A is the degree of associativity) instead of L_S. Therefore, the methodology used for direct-mapped caches can be employed for set-associative caches with some adjustments. The complexity of these modifications depends on the type of *replacement policy* used inside the cache sets. Considering actual architectures, two replacement policies can be considered: *Random Replacement* policy or *Least Recently Used* policy.

Random Replacement For this replacement policy, the main difference lays in the probability of transition matrices. P_{load} is unchanged but P_{flush} must be modified. The main difference lays in the fact that a reference is not sufficient for flushing the cache. Several such references should occur before the cache is flushed. This notion can be simply formalized by modifying the probability that one element be flushed, the expression of which appears in $P_{flush}(1,1)$ and $P_{flush}(1,3)$.

Least Recently Used In this case, the method previously used changes slightly. More precisely, it is necessary to modify the state vector. Instead of two states only (in, out), we have to consider $D_A + 1$ states where D_A is the degree of associativity. Then, new *probability of transition matrices* must be computed to describe the probability for an element to shift from one position to another in the LRU queue. This methodology is the direct extension of the one used for direct-mapped cache where $D_A = 1$.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 642 ARCHE : UN LANGAGE PARALLELE A OBJETS FORTEMENT TYPES
Marc BENVENISTE, Valérie ISSARNY
Mars 1992, 132 pages.

PI 643 CARTESIAN AND SATISTICAL APPROACHES OF THE SATISFIABILITY PROBLEM
Israël-César LERMAN
Mars 1992, 58 pages.

PI 644 PRIME MEMORY SYSTEMS DO NOT REQUIRE EUCLIDEAN DIVISION BY A PRIME NUMBER
André SEZNEC, Yvon JEGOU, Jacques LENFANT
Mars 1992, 10 pages.

PI 645 SKEWED-ASSOCIATIVE CACHES
André SEZNEC, François BODIN
Mars 1992, 20 pages.

PI 646 INTERLEAVED PARALLEL SCHEMES : IMPROVING MEMORY THROUGHPUT ON SUPERCOMPUTERS
André SEZNEC, Jacques LENFANT
Mars 1992, 14 pages.

PI 647 COMMUNICATING PROCESSES AND FAULT TOLERANCE : A SHARED MEMORY MULTIPROCESSOR EXPERIENCE
Michel BANATRE, Maurice JEGADO, Philippe JOUBERT, Christine MORIN
Mars 1992, 40 pages.

PI 648 SET-THEORETIC GRAPH REWRITING
Jean-Claude RAOULT, Frédéric VOISIN
Mars 1992, 18 pages.

PI 649 UNE STRUCTURE D'INFORMATION POUR LES ALGORITHMES D'EXCLUSION MUTUELLE FONDES SUR UNE ARBORESCENCE
Jean-Michel HELARY, Achour MOSTEFAOUI, Michel RAYNAL
Mars 1992, 18 pages.

PI 650 BLOCK-ARNOLDI AND DAVIDSON METHODS FOR UNSYMMETRIC LARGE EIGENVALUE PROBLEMS
Miloud SADKANE
Avril 1992, 24 pages.

PI 651 COMPILING SEQUENTIAL PROGRAMS FOR DISTRIBUTED MEMORY PARALLEL COMPUTERS WITH PANDORE II
Françoise ANDRE, Olivier CHERON, Jean-Louis PAZAT
Avril 1992, 18 pages.

PI 652 CHARACTERIZING THE BEHAVIOR OF SPARSE ALGORITHMS ON CACHES
Olivier TEMAM, William JALBY
Avril 1992, 20 pages.

PI 653 MADMACS : UN OUTIL DE PLACEMENT ET ROUTAGE POUR LE DESSIN DE MASQUES DE RESEAUX REGULIERS
Eric GAUTRIN, Laurent PERRAUDEAU, Oumarou SIE
Avril 1992, 16 pages.

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique.

ISSN 0249 - 6399