MEMORY HIERARCHY PERFORMANCE PREDICTION FOR BLOCKED SPARSE ALGORITHMS*

BASILIO B. FRAGUELA, RAMÓN DOALLO
Dept. Electrónica e Sistemas. Univ. da Coruña, Campus de Elviña s/n
A Coruña, 15071 (Spain)
{*basilio,doallo*}*@udc.es*

and

EMILIO L. ZAPATA
Dept. de Arquitectura de Computadores. Univ. de Málaga,
Complejo Tecnológico Campus de Teatinos Málaga, 29080 (Spain)
ezapata@ac.uma.es

ABSTRACT

Nowadays the performance gap between processors and main memory makes an efficient usage of the memory hierarchy necessary for good program performance. Several techniques have been proposed for this purpose. Nevertheless most of them consider only regular access patterns, while many scientific and numerical applications give place to irregular patterns. A typical case is that of indirect accesses due to the use of compressed storage formats for sparse matrices. This paper describes an analytic approach to model both regular and irregular access patterns. The application modeled is an optimized sparse matrix-dense matrix product algorithm with several levels of blocking. Our model can be directly applied to any memory hierarchy consisting of K-way associative caches. Results are shown for several current microprocessor architectures.

Keywords: Sparse matrix, irregular computation, cache performance, memory hierarchy, probabilistic analytical model.

1 Introduction

Despite the fact that several hardware and software techniques have been developed and successfully applied in order to improve the memory hierarchy behavior, memory accesses continue to be a bottleneck for system performance. This is particularly true in the case of applications characterized by irregular access patterns, as they limit the spatial and temporal locality that caches try to exploit. It is the case of codes that deal with sparse matrices, whose compressed storage formats [1] generate indirect accesses. Our interest in sparse codes is justified by the broad spectrum of engineering and scientific computing applications where such codes arise.

*This work was supported by the Ministry of Education and Science (CICYT) of Spain under project TIC96-1125-C03, Xunta de Galicia under Project XUGA20605B96, and E.U. Brite-Euram Project BE95-1564

Cache performance evaluation tools are required both to provide quantitative predictions of miss ratio and information to guide cache optimization. The traditional approach has been the software simulation of the cache effect for every memory access [2], which is very accurate but gives little insight into the reasons for the cache behavior and requires very long computation times. Some modern microprocessors overcome this problem by providing performance monitoring tools such as built-in counters. Still they present the limitations of requiring program compilation and execution and being restricted to a specific platform. A more general approach is that of analytical models, which, in addition to requiring short computation times, make more flexible the parametric study of the cache. Although many models extract input parameters from address traces [3], [4], [5] and combine them with cache definition parameters, more general purpose models have been developed taking as input the code to analyze [6], [7], [8]. Nevertheless these models only consider dense algebra codes, with regular access patterns. There are very few works related to the analytic modeling of sparse codes, and their scope is very limited compared to ours. For example, [9] is restricted to the partial modeling (only taking into account self interferences) of the behavior of one vector in a very simple code such as the sparse matrix-vector product, and only on direct-mapped caches. Besides until now no attempts have been made to build a general framework for the modeling of these kinds of codes.

On the other hand, several studies have been carried out on sparse algebra kernels in order to improve their performance applying software techniques such as blocking, loop unrolling, loop interchanging or software pipelining. This work demonstrates the feasibility of modeling such types of complex algorithms that fit hardware improvements of current high performance microprocessors. As an example, an optimized version of the sparse matrix-dense matrix product described in [10] is modeled. In this work we apply the probabilistic model for K-way associative caches with LRU replacement introduced in [11], where only simple algebra kernels such as sparse matrix-vector product and sparse matrix transposition were considered. Optimum block sizes for the memory hierarchy of an arbitrary system can be predicted by means of this model. A uniform distribution of the non zero elements is considered, but the model proposed can be extended to different types of distributions as we demonstrate in [12].

The algorithm to model is described in the next section. Basic model concepts and an example of the modeling process are presented in Section 3. Section 4 validates the model and applies it in order to propose block sizes for obtaining minimum execution times on several platforms. A brief study of the cache behavior as a function of several paratemers is also carried out. Section 5 concludes the paper.

2 Optimized Sparse Matrix-Dense Matrix Product

The algorithm modeled is displayed in Figure 1. It stores the sparse matrix using the Compressed Row Storage (CRS) format [1], which uses three vectors: vector `A` contains the sparse matrix non zero elements or entries, vector `C` stores the column

```
 1 DO J2=1, H, BJ
 2   LIMJ=J2+MIN(BJ, H-J2+1)-1
 3   DO I=1, M+1
 4     R2(I)=R(I)
 5   ENDDO
 6   DO K2=1, N, BK
 7     LIMK=K2+MIN(BK, N-K2+1)
 8     DO J=1, LIMJ-J2+1
 9       DO K=1, LIMK-K2
10         WB(J,K)=B(K2+K-1,J2+J-1)
11       ENDDO
12     ENDDO
13     DO I=1, M
14       K=R2(I)
15       LK=R(I+1)

16       d1=D(I,J2)
17       d2=D(I,J2+1)
         ...
18       dbj=D(I,J2+BJ-1)
                 ↪
19       DO WHILE (K<LK AND C(K)<LIMK)
20         a=A(K)
21         ind=C(K)
22         d1=d1+a*WB(1,ind-j2+1)
23         d2=d2+a*WB(2,ind-j2+1)
           ...
24         dbj=dbj+a*W(BJ, ind-j2+1)
25         K=K+1
26       ENDDO
27       D(I,J2)=d1
28       D(I,J2+1)=d2
         ...
29       D(I,J2+BJ-1)=dbj
30       R2(I)=K
31     ENDDO
32   ENDDO
33 ENDDO
```

Fig. 1: A sparse matrix-dense matrix product with blocking at the memory and register levels.

of each entry, and vector R points to the location in vectors A and C where a new row of the sparse matrix starts. This format forces a row-wise access of the sparse matrix. The dense matrix is B, while D stores the product matrix.

In this code the outer loop on I (line 13) of the product traverses the rows of the sparse matrix and matrix D (dimension I); then the loop on K (line 19) processes the entries of a given row (so the K dimension is the one common to the sparse and the dense matrices), and finally an inner loop, which has been completely unrolled due to a register blocking, considers the columns of matrices B and D. This is what we call the J dimension. In addition, this optimized version comprises one level of cache blocking using a block of BK rows and BJ columns of the dense matrix. When the processing of a new block begins, a transposed copy is stored in a temporal matrix WB which is the one the inner loop uses. The reason is that matrix B is accessed by rows in this loop, which does not favour the exploitation of locality, as matrices are stored by columns in FORTRAN. The use of a transposed copy permits taking advantage of both temporal and spatial locality. Vector R2 simplifies the access to the subrows of the sparse matrix associated to a given block of B by pointing to the location in vectors A and C where these subrows start.

Blocking at the register level is also considered to optimize the algorithm performance; for this purpose techniques such as strip mining, loop interchanging, full loop unrolling and the elimination of redundant load and store operations were applied. As mentioned above, these techniques have been applied to the inner loop, taking the load and stores for matrix D out of the loop on K. This modification requires BJ registers (d1, d2, . . . , dbj in Figure 1) to store these values. The resulting algorithm has a twodimensional blocking for registers, resulting in fewer loads and stores per arithmetic operation. Besides, the number of independent floating point operations in the loop body is increased.

3 Analytic Model

For our modeling purposes we distinguish two kinds of misses: the intrinsic ones take place the first time a memory block corresponding to a cache line is accessed. The remaining misses affecting the same line are called interference misses, as they are due to interferences with other lines that are placed in the same set. When the interfering lines belong to the same program vector, we say that they are self interferences, whereas when they belong to another vector, they are considered cross interferences.

In our model and the platforms we have studied (see Section 4) the K-way cache sets are managed following an LRU replacement policy (pseudo-LRU in real systems, actually). In this case K or more different lines mapped to the set where a given line resides must be accessed between two consecutive accesses to this line in order to obtain an interference miss in the second access. The probability that an access to a line results in an interference miss corresponds to the likelihood of having placed at least K lines in its set since the last time it was accessed. We use the concept of area vector to compute this probability, being $S_{\mathrm{V}} = S_{\mathrm{V}_0}, S_{\mathrm{V}_1}, \ldots, S_{\mathrm{V}_K}$ the area vector corresponding to a given program vector `V`. The element in the i-th position stands for the ratio of sets that have received $K - i$ lines of this program vector during the execution of a given portion of code. The exception is S_{V_0}, which is the ratio of sets that have received K or more lines. Different expressions have been developed to calculate the area vectors associated to typical access patterns [11]. Next, we describe those ones that arise in our particular code in Figure 1.

3.1 Access Patterns Modeled

One of the most common patterns is the sequential access to n consecutive words, whose area vector $S_{\mathrm{s}}(n)$ is

$$\begin{array}{ll} S_{\mathrm{s}_{(K-\lfloor l \rfloor)}}(n) = 1 - (l - \lfloor l \rfloor) & \\ S_{\mathrm{s}_{(K-\lfloor l \rfloor - 1)}}(n) = l - \lfloor l \rfloor & \\ S_{\mathrm{s}_i}(n) = 0 & 0 \leq i < K - \lfloor l \rfloor - 1, K - \lfloor l \rfloor < i \leq K \end{array} \quad (1)$$

where $l = \max\{K, (n + L_{\mathrm{s}} - 1)/(L_{\mathrm{s}} N_K)\}$ is the maximum of K and the average number of lines placed in each set. The term $L_{\mathrm{s}} + 1$ added to n stands for the average extra words brought to the cache in the first and last accessed lines.

Another useful expression is the one corresponding to an access to an n word vector in which any cache line has the same probability P_l of being accessed. We denote the area vector in this case as $S_{\mathrm{l}}(n, P_l)$

$$\begin{array}{ll} S_{\mathrm{l}_i}(n, P_l) = P(X = K - i), \;\; X \in B(n/(L_{\mathrm{s}} N_K), P_l) & m < i \leq K \\ S_{\mathrm{l}_m}(n, P_l) = P(X \geq K - m), \;\; X \in B(n/(L_{\mathrm{s}} N_K), P_l) & \\ S_{\mathrm{l}_i}(n, P_l) = 0 & 0 \leq i < m \end{array} \quad (2)$$

where $m = \max\{0, K - \lceil n/(L_{\mathrm{s}} N_K) \rceil\}$ and B(n,p) is the binomial distribution[a]. In the case of k accesses of this type, the area vector is $S_{\mathrm{l}}^k(n, P_l) = S_{\mathrm{l}}(n, 1 - (1 - P_l)^k)$.

[a]We define the binomial distribution on a non integer number of elements n as $P(X = x), X \in B(n, p) = (P(X = x), X \in B(\lfloor n \rfloor, p))(1 - (n - \lfloor n \rfloor)) + (P(X = x), X \in B(\lceil n \rceil, p))(n - \lfloor n \rfloor)$

3.2 *Other Important Issues and Input Parameters*

When calculating self interference area vectors we will need to compute the average number of lines that compete with a given line in an n word vector. Function $C(n)$ calculates it as

$$C(n) = \begin{cases} 0 & \text{if } z \leq 1 \\ \lfloor z \rfloor / z(2z - \lfloor z \rfloor - 1) & \text{if } z > 1 \end{cases} \quad (z = n/(L_\mathrm{s} \cdot N_K)) \tag{3}$$

where z is the average number of lines of the vector mapped to a set.

As between consecutive accesses to a given line several program vectors may be accessed, and some of them may be referenced different access patterns, a mechanism is needed to add area vectors. Given two area vectors $S_\mathrm{U} = (S_{\mathrm{U}_0}, S_{\mathrm{U}_1}, \ldots, S_{\mathrm{U}_K})$ and $S_\mathrm{V} = (S_{\mathrm{V}_0}, S_{\mathrm{V}_1}, \ldots, S_{\mathrm{V}_K})$, the union area vector $S_\mathrm{U} \cup S_\mathrm{V}$ that comprises the accesses corresponding to both area vectors is defined as

$$\begin{array}{ll} (S_\mathrm{U} \cup S_\mathrm{V})_0 = \sum_{j=0}^{K} \left(S_{\mathrm{U}_j} \sum_{i=0}^{K-j} S_{\mathrm{V}_i} \right) & \\ (S_\mathrm{U} \cup S_\mathrm{V})_i = \sum_{j=i}^{K} S_{\mathrm{U}_j} S_{\mathrm{V}_{(K+i-j)}} & 0 < i \leq K \end{array} \tag{4}$$

From now on the symbol $\cup$ will be used to denote the vector union operation. This method makes no assumptions on the relative positions of the program vectors in memory, as it is based on the addition, as independent probabilities, of the area ratios.

Table 1: Input parameters.

C_s	Cache size in words
L_s	Line size in words
K	Associativity
Nk	Number of cache sets $(C_\mathrm{s}/(L_\mathrm{s} \cdot K))$
M	Number of rows of the sparse matrix
N	Number of columns of the sparse matrix
H	Number of columns of the dense matrix
BJ	Block size in the `J` dimension
BK	Block size in the `K` dimension
N_{BJ}	Number of blocks in the J dimension (H/BJ)
N_{BK}	Number of blocks in the K dimension (N/BK)
N_nz	Number of entries of the sparse matrix
β	Average number of entries per row (N_nz/M)
p_n	Probability that a position in the sparse matrix contains an entry $\left(\frac{N_\mathrm{nz}}{M \cdot N}\right)$
r	size of an integer/size of a floating point

Table 1 displays the main input parameters for our model. Word stands in our model for the size of the data elements the algorithm handles. The size of a floating point number has been chosen as word, but the model is totally scalable, as integers are considered through the use of parameter r.

3.3 *Modeling Cache Behavior of Algorithm Arrays*

The modeling of the proposed code has been performed through the modeling of the behavior of each one of the vectors and matrices involved. For each array, each point where it is referenced is considered separately. As an example, and due to space limitations, only the modeling of the behavior of matrix `WB` will be detailed here, as it usually accounts for the greatest portion of the misses and its access pattern is the most complex. We will first calculate the number of misses on this matrix in the inner loop; later we will study matrix `WB` behaviour during the storage in it of the transposed blocks of matrix `B`.

3.3.1 Misses on Matrix `WB` in the Inner Loop

The probability of a hit in an access to a given line of this matrix during the processing of the j-th row is estimated as

$$\begin{aligned} P_{\text{hit WB}}(j) = &\sum_{i=1}^{j-1} P(1-P)^{i-1}(1-S_{\text{intf WB}_0}(i)) \\ &+ (1-P)^{j-1}(1-S_{\text{intf WB}_0}(j))(1-S_{\text{intf init WB}_0}) \end{aligned} \tag{5}$$

where $P = 1-(1-p_n)^{Av(BJ,L_\text{s})}$ is the probability of accessing a line of matrix `WB` during the processing of a subrow of the sparse matrix, being $Av(a,b)$ the average number of different groups of a consecutive elements into which we divide an abstract infinite vector that fall in a group of b consecutive elements of that vector. The formula is given by

$$Av(a,b) = \begin{cases} 1 & \text{if } a \bmod b = 0 \\ (b+a-\gcd(b, a \bmod b))/a & \text{otherwise} \end{cases} \tag{6}$$

This way $P(1-P)^{i-1}$ in (5) is the probability that the last access to a line of matrix `WB` has taken place during the processing of row $j-i$. On the other hand, $S_{\text{intf WB}}(i)$ is the area vector corresponding to the accesses to all of the vectors involved in the processing of i subrows of the sparse matrix along their corresponding i iterations of the loop on `I`. As explained in Section 3, its first element ($S_{\text{intf WB}_0}$) gives the probability that a line accessed just before those processes take place suffers an interference miss if it is rerefenced just after they have finished. This vector is calculated adding with the $\cup$ operation the area vectors for these accesses:

- Sequential access to i elements of vectors `R` and `R2` ($S_\text{s}(i \cdot r)$, see Section 3.1).
- Access to a portion of $\beta \cdot i + L_\text{s} - 1$ elements of vector `A` with an uniform access probability per line (see Section 3.1)

  $$p_\text{A} = (p_n \cdot BK + (L_\text{s}-1)(1-P_{\text{share A}})(1-(1-p_n)^{BK}))/\beta \tag{7}$$

  where

  $$P_{\text{share A}} = \frac{1}{L_\text{s}} \sum_{i=0}^{L_\text{s}-2} \sum_{j=0}^{i} \binom{N-BK}{j} p_n^j (1-p_n)^{N-BK-j} \tag{8}$$

  is the probability that the elements belonging to a group of BK consecutive columns in two consecutive rows of the sparse matrix share a line in vector `A`. The term $1-(1-p_n)^{BK}$ is used to take into account the probability that the sublock is empty in a row, as in this case no access to vector `A` takes place.

- Idem for vector C, being the access probability per line

$$p_{\mathrm{C}} = (p_n \cdot BK \cdot r + (L_{\mathrm{s}} - r)(1 - P_{\mathrm{share\ C}}))/(\beta \cdot r) \tag{9}$$

  where $P_{\mathrm{share\ C}}$ is calculated using expression (8) replacing L_{s} by L_{s}/r because vector C is made up of integers. For the same reason, the number of words on which the access may take place is $\beta \cdot i \cdot r + L_{\mathrm{s}} - 1$.
- If $BJ \cdot BK > N_K \cdot L_{\mathrm{s}}$ self interferences may occur. The associated area vector is estimated as the one corresponding to a vector of $C(BJ \cdot BK)N_K \cdot L_{\mathrm{s}}$ words with uniform access probability per line $1 - (1 - P)^i$ (see Section 3.1).
- Finally, the area vector for the access to i subrows of BJ elements of matrix D is calculated using a deterministic algorithm described in [13].

The last term in expression (5) stands for the probability of a hit in a line that has not been accessed in the inner loop yet (probability $(1 - P)^{j-1}$) , but that is still resident in the cache after the copying of the present block of matrix B to WB. This will happen if it has not been affected by interferences both in the iterations of the loop on I and in the copying process $(1 - S_{\mathrm{intf\ init\ WB}_0})$. The calculation of this last probability is not included here due to space limitations.

The number of misses on matrix WB in this loop, $M_{\mathrm{inner\ WB}}$, is obtained as the product of three terms: the average miss probability, the average number of lines accessed when a column of this matrix is read, and the number of times that read takes place (once per entry and block in the J dimension):

$$M_{\mathrm{inner\ WB}} = \left(1 - \frac{\sum_{j=1}^{m} P_{\mathrm{hit\ WB}}(j)}{M}\right) Av(L_{\mathrm{s}}, BJ) N_{\mathrm{nz}} N_{BJ} \tag{10}$$

3.3.2 Misses on Matrix WB in the Copying

This value, $M_{\mathrm{copy\ WB}}$, is calculated as the product of the average number of misses per copy process by the number of blocks, $N_{BJ} N_{BK}$. This average is estimated as

$$M_{\mathrm{copy\ WB}} = \sum_{j=1}^{BJ} \sum_{i=1}^{\frac{BJ \cdot BK}{L_{\mathrm{s}}}} A \cdot S_{\mathrm{s}_0}(1) + \left(\frac{L_{\mathrm{s}}}{BJ} - A\right) P_{\mathrm{miss\ first}}(i, j) \tag{11}$$

where $A = \max\{0, \frac{L_{\mathrm{s}}}{BJ} - 1\}$ is the average number of accesses after the first one to a given line of matrix WB during an iteration of the loop on line 8. As the equation shows, the miss probability for these accesses is that associated to an access to an element of matrix B. The probability of a miss for the first access to line i during iteration j of the loop, $P_{\mathrm{miss\ first}}(i, j)$, is calculated as

$$\begin{aligned} P_{\mathrm{miss\ first}}(i, j) = & P_{\mathrm{acc}}(j - 1)(S_{\mathrm{self}}(BJ, BK) \cup S_{\mathrm{s}}(BK))_0 \\ & + (1 - P_{\mathrm{acc}}(j - 1))(1 - P_{\mathrm{hit\ WB}}(M)(1 - P_{\mathrm{exp\ WB}}(i, j))) \end{aligned} \tag{12}$$

where $P_{\mathrm{acc}}(j)$ is the probability of the line having been accessed in the j previous iterations, which is $\min\{1, (L_{\mathrm{s}} + j - 2)/BJ\}$ but for $j = 0$, for which the probability is null. The first access to a line results in a miss unless it is in the cache when the copying begins (probability $P_{\text{hit WB}}(M)$) and it has not been replaced during the copy of the elements previously accessed, $(1 - P_{\text{exp WB}}(i, j))$. On the other hand, if the line has been accessed in the previous iteration, only the accesses to BK elements of matrix `B` located in two consecutive columns (whose area we approach by a completely sequential access) and the accesses to a row of matrix `WB` can have replaced the considered line. This self interference area vector $(S_{\text{self}}(BJ, BK))$ is calculated using the deterministic algorithm mentioned above. As for $P_{\text{exp WB}}(i, j)$, it is estimated as the first element of the area vector resulting from the union of the area vectors associated to the following patterns:

- $i \cdot L_{\text{s}}/BJ$ consecutive elements of the subcolumn of matrix `B` that is being copied, $S_{\text{s}}(i \cdot L_{\text{s}}/BJ)$.
- $j - 1$ subcolumns of BK elements of `B` corresponding to the previous copied subcolumns. It is calculated using the deterministic algorithm.
- $S_{\text{l}}(\lfloor (i \cdot L_{\text{s}} - 1)/(N_K L_{\text{s}}) \rfloor N_K L_{\text{s}}, P(j))$, which is the self interference area vector for the lines of `WB` with index smaller than i.
- $S_{\text{l}}(\lfloor (BJ \cdot BK - i \cdot L_{\text{s}})/(N_K L_{\text{s}}) \rfloor N_K L_{\text{s}}, P(j - 1))$, which stands for the self interferences with the lines of the matrix with index greater than i.

4 Validation and Application

The code shown in Figure 1 was rewritten replacing the references to memory by functions that calculate the position to be accessed and write it to a trace file. Later this trace file was fed to the dineroIII cache simulator, integrated into the WARTS toolset [14]. Table 2 displays the prediction deviation Δ for several combinations of the input parameters obtained from the execution on synthetic matrices. For each combination several simulations were made changing the data structures starting addresses. In the table σ is the average deviation of the number of misses measured in the simulations. The average error obtained in the trial set was under 1.9%, and the maximum error was under 15%. The errors have never given place to deviations in the prediction of the miss ratio above 1.5%. The largest deviations were obtained for the combinations in which σ is large or BJ and BK are not dividers of H and N respectively and there are few blocks in any of these dimensions. The latter is the case for the deviations over 10% in Table 2: the use of blocks of size 2100 in a dimension with 5000 elements gives place to two blocks with 2100 elements and another one with only 800, instead of the three completely equal blocks of 2100 elements the model considers to simplify the problem. This leads to an overestimation of the number of misses.

In order to prove the usefulness of the model we have used it to derive the optimum block sizes for the execution of this code on several platforms. We have taken into account parameters such as the cache size, line size, associativity and miss penalty of each cache level. The policy followed to compare the different blocks

Table 2: Deviation of the model for optimized sparse matrix-dense matrix product : r=1; Order $M = N$, N_{nz} and H in thousands and C_{s} in Kwords. σ is the average deviation of the number of misses measured in the simulations; Δ is the deviation of the model prediction with respect to the average number of measured misses.

Order	N_{nz}	p_n	H	BJ	BK	C_s	L_{s}	K	σ	Δ
1.5	125	0.055	1.5	25	500	8	4	1	0.55%	2.76%
1.5	125	0.055	1.5	25	500	8	4	2	0.30%	3.89%
1.5	125	0.055	1.5	25	500	8	4	4	0.26%	4.13%
1.5	125	0.055	1.5	25	500	16	4	2	0.46%	0.15%
1.5	125	0.055	1.5	25	500	16	4	4	1.09%	-2.09%
5	500	0.020	0.1	10	500	2	4	2	0.20%	2.57%
5	500	0.020	0.1	10	500	4	8	2	0.45%	2.36%
5	500	0.020	0.1	10	1000	4	8	2	0.25%	3.13%
5	500	0.020	0.1	24	2100	16	8	1	0.30%	12.79%
5	500	0.020	0.1	28	2100	16	8	1	0.12%	14.50%
5	500	0.020	0.1	24	2100	16	8	2	0.08%	8.45%
5	500	0.020	0.1	28	2100	16	8	2	0.10%	10.11%
10	100	0.001	10	20	1000	16	4	1	0.16%	2.39%
10	100	0.001	10	20	1000	16	4	2	0.02%	2.45%
10	100	0.001	10	10	1000	16	4	2	0.15%	1.36%
10	100	0.001	10	10	1000	16	4	4	0.08%	1.20%
10	100	0.001	10	20	1000	32	4	1	0.34%	1.87%
10	100	0.001	10	20	1000	32	4	4	0.07%	0.69%

consisted in calculating the cost of the block as

$$\mathrm{Cost}(BJ, BK) = \sum_{i=1}^{N_{\mathrm{Levels}}} \mathrm{Cost}_i \cdot M(BJ, BK, C_{\mathrm{s}_i}, L_{\mathrm{s}_i}, K_i) \tag{13}$$

where N_{Levels} is the number of cache levels in the considered system; C_{s_i}, L_{s_i}, and K_i are the cache size, line size and associativity of level i, and Cost_i provides the relative cost of a miss at this level. $M(BJ, BK, C_{\mathrm{s}}, L_{\mathrm{s}}, K)$ is the number of misses predicted by the model for this block and cache. The parameters describing the matrices involved have been removed to simplify the expression.

The platforms used to carry out our study were the following ones: a DEC 433au Personal Workstation, which is based on a 433MHz 21164 Alpha processor; a SGI Origin 200 server with R10000 processors at 180 MHz and a SUN Ultra 1 using a 167 MHz UltraSPARC-I processor. We think they represent a wide range of present-day architectures in order to validate our experiments. The results obtained for several matrices are shown in Table 3. The trial set for BJ (block size in the `J` dimension) was 8, 10, 12, 16 and 20, while the values tested for BK (block size in the `K` dimension) were 50, 100, 200, 250, 500, 1000, 1250, 2500 and 5000 (these last ones only when applicable). The table displays the difference between the execution times obtained using the block proposed by the model, that is, the one that minimizes $\mathrm{Cost}(BJ, BK)$, and the optimum block, that is, the block that leads to the minimum execution time. This difference is expressed as a percentage of the time obtained for the optimum block.

The model was initially designed for write-back caches, considering the same cost and behavior for read and write misses. A variation has been developed to adapt it

Table 3: Deviation in the execution time using the block proposed by the model as a percentage of the optimum execution time: M and N in thousands, $H = 1000$.

Matrix			System		
M	N	p_n	433au	Origin 200	Ultra 1
1	1	0.05	0.00%	11.46%	0.00%
1	1	0.10	0.00%	10.64%	0.00%
2.5	2.5	0.08	2.10%	3.28%	0.19%
2.5	2.5	0.16	0.00%	2.75%	0.00%
5	2.5	0.01	6.83%	0.00%	1.96%
5	5	0.04	0.00%	0.00%	9.98%

to write-through non-allocating caches, as the first level caches of the 21164 and the UltraSPARC I processors follow this policy. It has consisted in considering any write on this caches as a miss, but giving it a weight proportional to the time required to solve it. Besides the write accesses to these caches produce no interference area vectors, as the affected data are not placed in the cache.

We see that the worst estimations have taken place for the case of the Origin server using a small matrix of order 1000. Although the percentage difference is noticeable, the execution time of the block proposed by the model is only less than 0.1 seconds larger than the optimum time for $p_n = 0.05$ and 0.18 seconds for $p_n = 0.1$. Anyway, we have used the `perfex` utility of the operating system in order to access the internal counters available in the R10000, which can provide information such as the number of data misses in each of the two levels of the hierarchy. The number of misses reported is not close to that predicted by the model, which is not surprising, since there are several effects not considered by it: the sharing of the second level cache with the code; the access patterns are not exactly those expected due to the use of intermediate variables and compiler optimizations, and the measurement of data misses that the model does not take into account such as those that take place when the sparse matrix is read from disk. Besides the CPU is always shared with other processes, which causes misses in the context switches. Anyway, we expect the most important patterns to be those reflected in the model, being our purpose not to give an accurate quantitative estimation of the real number of misses, but better give a fair idea of the evolution of the cache behavior with respect to the variations in the model input parameters, using the predicted number of misses as an indication of this behavior. The measurements obtained using the processor counters show that this goal has been achieved, as there is almost always a proportionality between the number of real misses measured in each cache level and the number of misses the model predicts. This fact is highlighted in Figure 2. The block size proposed by the model is really more effective than the optimum block size from the point of view of the memory hierarchy. What happens is that the total execution time depends on more factors: the block proposed by the model fits better in the caches, as it is smaller than the optimum block, but its use has more overhead, as it requires more iterations in the loops that control the blocks. On the other hand, we are talking about modern superscalar processors capable of issuing multiple instructions and hiding memory latencies in several ways, which makes the number of misses only a rough approximation of the program performance.

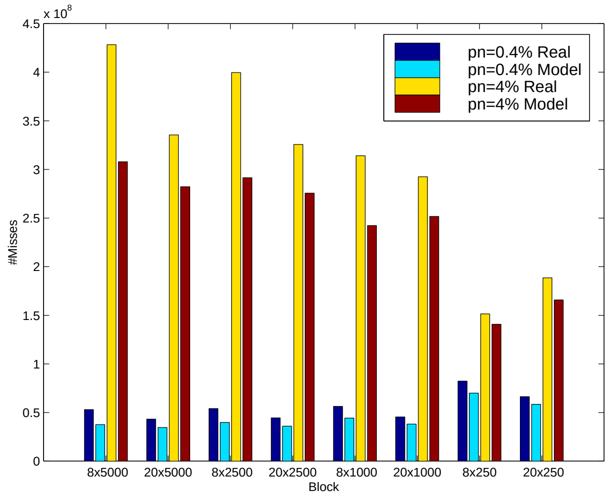

Fig. 2: Measured and predicted number of misses in the first level data cache of the R10000 processor during the product of a 5000x5000 sparse matrix with $p_n = 0.004$ and $p_n = 0.04$ using $H = 1000$ for different blocks.

Similar reasons can be given for the deviation observed in Table 3 for the largest matrix in the Ultra 1 system. Besides in this case we have the drawback that in the first level data cache each line is divided into two sub-blocks and our model does not support sub-block placement. The model could be easily extended in order to support this feature. Nevertheless we think subblock placement has not reached a wide implementation in current caches by now.

The maximum BJ value in our trials has been 20 because there must be BJ free floating point registers in order to implement the register blocking and the FORTRAN compiler of the 433au personal workstation uses up to 19 registers for this purpose. The optimum block sizes have almost always used the maximum value of BJ in the three machines tested. This makes sense, since it leads to the minimum number of blocks in the `J` dimension. The number of accesses is reduced and the exploitation of the spatial locality is favoured. Access to matrix `WB` in the inner loop takes place sequentially in this dimension, whose components are stored in consecutive memory positions. Nevertheless there have been strong variations in the optimum values of BK for the different architectures and matrices, which has moved us to make a study of the influence of different parameters on the cache behavior.

In Figures 3 to 5 we have tested the cache behavior during the product of a 5000x5000 sparse matrix as a function of BK keeping BJ constant and equal to 20. Each one of the three figures focuses on different degrees of associativity, line and cache sizes, respectively. Typical values of second level caches have been chosen for the cache parameters. In the following, cache behavior is explained based only on matrix `WB`, as it usually accounts for the greatest amount of misses, as said in

Section 3.3.

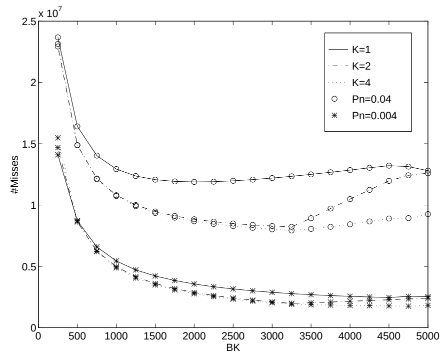

Fig. 3: Number of misses during the product of a 5000x5000 sparse matrix for different degrees of associativity on a 128Kw cache with a line size of 16 words; $p_n = 0.004$ and $p_n = 0.04$ using $H = 1000$.

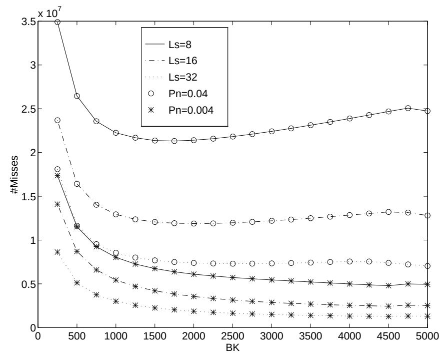

Fig. 4: Number of misses during the product of a 5000x5000 sparse matrix for different line sizes on a direct-mapped 128Kw cache; $p_n = 0.004$ and $p_n = 0.04$ using $H = 1000$.

Figure 3 shows that small values of p_n favour the use of large blocks no matter the size of the cache sets because all of the block sizes considered fit in the cache and there are few cross interferences due to the small number of entries. Nevertheless as p_n grows the optimum block decreases its size because of the increase in the number of cross interferences. Besides, as expected, the smaller the value of K the smaller the optimum block, as the interferences effect is greater.

Line size (L_s) influence on the number of cache misses is shown in Figure 4. The references inside the inner loop are basically 20 read accesses to consecutive memory positions, so for typical second level cache line sizes, the larger the line the better. Only for extremely large line sizes (≥ 256) an increase in the number of misses arises, due to their negative effect on the interference probability. On the other hand, the optimum block size experiences little variations with L_s. Although it is not shown in the graph, the increase of the associativity gives place to an increase of the optimum block size with the line size, as it balances the interference growth that takes place with larger line sizes. It must be taken into account that the increase of the line size can lead to a longer execution time despite the reduction in the number of misses because of the longer miss time.

The behavior of the optimum block size as a function of the cache size is scalable for the two different values of p_n in Figure 5. For $C_s \geq 128$Kw any block fits in the cache, so the largest block is the best for large caches. Nevertheless we see the effect observed in Figure 3 of reducing the optimum block size as p_n grows because of the increase in the cross interference probability. The behavior of the only second level cache considered in Figure 5 for which some blocks do not fit, that is, the 64Kw cache, limits the optimum block size to be the largest below C_s in the case of $p_n = 0.004$ in order to avoid self interferences and exploit the cache. As p_n grows this limit is reduced because of the added effect of self and cross interferences.

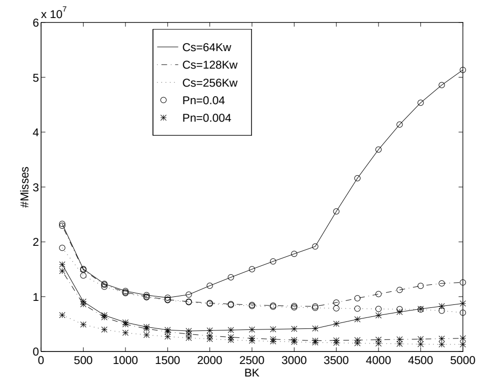

Fig. 5: Number of misses during the product of a 5000x5000 sparse matrix for different cache sizes on a 2-way associative cache with a line of 16 words; $p_n = 0.004$ and $p_n = 0.04$ using $H = 1000$.

The shape of the curve for the 64Kw cache and the matrix with $p_n = 0.04$ shows clearly the relative importance of cross and self interferences: these latter are much more important.

5 Conclusions

A cache behavior model developed by our group has been applied in a systematic way to a remarkably complex code in comparison with those in the previous literature. The model has demonstrated its usability for proposing code parameters, such as block sizes, that lead to minimum execution times on real platforms.

The model was first validated by simulations showing a good degree of accuracy. Later it was used to derive optimum block sizes for different architectures and matrices taking into account the different cache levels of each system. The blocks proposed by the model were almost always the optimum ones or provided very similar results. Finally, a comparison of the predicted number of misses with the real values measured using the internal built-in counters of the R10000 microprocessor shown that, although the model is far from being quantitatively accurate due to several factors it does not take into account (context switches, sharing of certain cache levels with code, etc.), its predictions do follow the real cache behavior qualitatively.

Besides, the model presents the advantage of its high execution speed in comparison to simulations and even to real executions providing a flexible architecture-independent tool for the analysis of the cache hierarchy behavior and the proposal of optimum blocks. As an example, let us consider the product of a 5000x5000 sparse matrix with 500K entries by a dense matrix with $H = 100$ using a 20x1000 block on a direct mapped 128Kw cache with $L_s = 16$. These are the parameters of the second level cache of the Ultra 1 system mentioned above. The time required to generate the trace and process it in this system was 690 seconds using dineroIII. The real execution time of this product consumed 7 seconds of CPU time, while the model required 0.64 seconds. Although two executions of the model are needed to reflect the behavior of this two-level hierarchy, the model is still competitive with respect to the real execution. This is specially true if we take into account compilation times. The difference grows with the dimensions of the matrices and the number of entries (as the simulation and execution times are proportional to the number of accesses) and with large miss rates, because of optimizations in the model implementation.

References

1. R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* (SIAM Press, 1994).
2. R.A. Uhlig and T.N. Mudge, Trace-Driven Memory Simulation: A Survey, *ACM Computing Surveys* **29** (1997) 128–170.
3. A. Agarwal, Analysis of Cache Performance for Operating Systems and Multiprogramming, Ph. D. Thesis, University of Stanford, 1987.
4. D. Buck and M. Singhal, An Analytic Study of Caching in Computer Systems, *J. of Parallel and Distributed Computing* **32** (1996) 205–214.
5. B.L. Jacob, P.M. Chen, S.R. Silverman and T.N. Mudge, An Analytical Model for Designing Memory Hierarchies, *IEEE Transactions on Computers* **45** (1996) 1180–1194.
6. S. Ghosh, M. Martonosi and S. Malik, Cache Miss Equations: An Analytical Representation of Cache Misses, in *Proc. 11th ACM Int'l. Conf. on Supercomputing (ICS'97)*, Vienna, Austria, July 1997, 317–324.
7. O. Temam, C. Fricker and W. Jalby, Cache Interference Phenomena, in *Proc. ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, Nashville, TN, May 1994, 261–271.
8. M.S. Lam, E.E. Rothberg and M.E. Wolf, The cache Performance and Optimizations of Blocked Algorithms, in *Proc. ASPLOS IV*, Santa Clara, CA, April 1991, 63–74.
9. O. Temam and W. Jalby, Characterizing the Behaviour of Sparse Algorithms on Caches, in *Proc. IEEE Int'l. Conf. on Supercomputing'92* Minneapolis, MN, Nov. 1992, 578-587.
10. J.J. Navarro, E. García, J.L. Larriba-Pey and T. Juan, Block Algorithms for Sparse Matrix Computations on High Performance Workstations, in *Proc. 10th ACM Int'l. Conf. on Supercomputing (ICS'96)*, Philadelphia, May 1996, 301–308.
11. B.B. Fraguela, R. Doallo and E.L. Zapata, Modeling Set Associative Caches Behavior for Irregular Computations, in *Proc. ACM Sigmetrics/Performance Joint International Conf. on Measurement and Modeling of Computer Systems*, Madison, Wisconsin, June 1998, 192–201.
12. R. Doallo, B.B. Fraguela and E.L. Zapata, Cache Probabilistic Modeling for Basic Sparse Algebra Kernels involving Matrices with a Non Uniform Distribution, in *Proc. EUROMICRO'98*, Vasteras, Sweden, Aug. 1998, 345–348.
13. B.B. Fraguela, Analytical Modeling of the Cache Memories Behavior, Ph. D. Thesis, Universidade da Coruña, 1999 (in Spanish).
14. A.R. Lebeck and D.A. Wood, Cache Profiling and the SPEC Benchmarks: A Case Study, *IEEE Computer* **27** (1994) 15–26.