Precise Automatable Analytical Modeling of the Cache Behavior of Codes with Indirections

DIEGO ANDRADE, BASILIO B. FRAGUELA, and RAMÓN DOALLO
Universidade da Coruña

16

The performance of memory hierarchies, in which caches play an essential role, is critical in nowadays general-purpose and embedded computing systems because of the growing memory bottleneck problem. Unfortunately, cache behavior is very unstable and difficult to predict. This is particularly true in the presence of irregular access patterns, which exhibit little locality. Such patterns are very common, for example, in applications in which pointers or compressed sparse matrices give place to indirections. Nevertheless, cache behavior in the presence of irregular access patterns has not been widely studied. In this paper we present an extension of a systematic analytical modeling technique based on PMEs (probabilistic miss equations), previously developed by the authors, that allows the automated analysis of the cache behavior for codes with irregular access patterns resulting from indirections. The model generates very accurate predictions despite the irregularities and has very low computing requirements, being the first model that gathers these desirable characteristics that can automatically analyze this kind of codes. These properties enable this model to help drive compiler optimizations, as we show with an example.

Categories and Subject Descriptors: B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.4 [**Performance of Systems**]: Modeling techniques

General Terms: Measurement, Performance

Additional Key Words and Phrases: Analytical modeling, memory hierarchy, irregular access patterns, performance prediction

ACM Reference Format:

Andrade, D., Fraguela, B. B., and Doallo, R. 2007. Precise automatable analytical modeling of the cache behavior of codes with indirections. ACM Trans. Archit. Code Optim. 4, 3, Article 16 (September 2007), 34 pages. DOI = 10.1145/1275937.1275940 http://doi.acm.org/10.1145/1275937.1275940

This work has been supported in part by the Ministry of Science and Technology of Spain under contract TIN2004-07797-C02-02, and by the Xunta de Galicia under contract PGIDIT03-TIC10502PR.
Author's address: Diego Andrade, Basilio B. Fraguela and Ramón Doallo, Departamento de Electrónica e Sistemas, Universidade da Coruña, Facultade de Informática, Campus de Elviña, 15071. A Coruña, Spain; email: dcanosa@udc.es, basilio@udc.es, doallo@udc.es.

1. INTRODUCTION

The importance of the role of the memory hierarchy in current general-purpose and embedded systems grows in parallel to the impact of the memory bottleneck problem. Unfortunately, cache behavior is unstable and depends on many parameters [Lam et al. 1991; Temam et al. 1994]. As a result, cache performance is difficult to predict, analyze, and understand. Trace-driven simulations [Uhlig and Mudge 1997] and profiling with built-in hardware counters [Ammons et al. 1997] can provide accurate estimations, but these approaches have relatively high computing requirements, they are restricted to a given architecture, in the case of the counters, and, most important, they give little insight on the reasons for the observed behavior. Analytical models, on the other hand, can help explain such behavior and be fast enough even to be used in a production compiler. In the past few years, accurate and automatable analytical models able to analyze whole programs with regular access patterns have appeared [Ghosh et al. 1999; Chatterjee et al. 2001; Vera and Xue 2002; Fraguela et al. 2003]. Nevertheless, none of the approaches to model the cache behavior of codes with irregular access patterns because of indirections proposed so far [Fraguela et al. 1998; Ladner et al. 1999; Cascaval et al. 2000; Mitchell et al. 2001] is both automatable and reasonably accurate. This is despite the fact that the analysis of the cache behavior in the presence of irregular access patterns is very important, since many relevant applications exhibit them, and cache performance usually drops in their presence as a result of their lack of locality.

In this paper, we extend the PME (probabilistic miss equations) model [Fraguela et al. 2003], which was restricted to codes with regular access patterns, so that it is now able to automatically analyze codes with indirections. The ability of our model to analyze these codes in a totally automatic way has enabled its integration in the XARK compiler [Arenaz et al. 2003], which is built on top of the Polaris compiler framework [Blume et al. 1996]. Such integration has been described in Andrade et al. [2007].

Our modeling considers indirections in which all the elements of the array accessed by means of the indirection have the same probability of being accessed, i.e., where the irregular access is uniformly distributed on the referenced array. This restriction eases the treatment of the problem in this first attempt to model automatically the cache behavior of codes with indirections, while allowing to represent the most important problems that irregular access patterns pose for their modeling. We acknowledge, though, that most indirections in real programs do not follow an uniform distribution; thus we are currently extending our model to consider other distributions. As a first step in this direction, we present a simple extension that allows to model the behavior of indirections generated by banded matrices, which we validate successfully with matrices from the Harwell–Boeing collection [Duff et al. 1992]. Still, our experiments show that our current model accurately predicts the behavior of several codes when matrices with nonuniform distributions are used. More interestingly, we have performed a successful preliminary experiment driving compiler optimizations that involve codes for which the model predictions for real nonuniform sparse matrices must be considered as fair approximations, rather than as accurate

estimations. The results point out that the model is good enough qualitatively to drive optimizations in the presence of typical real matrices.

We will see that the modular nature of the PME model facilitates the extension, and that its probabilistic nature provides the right tools for the modeling of codes with irregular access patterns. The validations reveal that the resulting model is both accurate and fast. A previous work in this line [Andrade et al. 2006] considered the extension of the model to analyze codes with irregular access patterns because of conditional statements.

The rest of the paper is organized as follows. An overview of the PME model and the scope of our extension is provided in the following section. Section 3 describes both the formulas that estimate the number of misses generated by regular access patterns of the existing PME model, as well as the new ones we introduce to support the modeling of irregular access patterns. Both kinds of formulas depend on miss probabilities that represent the impact on the cache of each given access pattern found in the code. Section 4 explains the procedure followed by the model to estimate these probabilities. The extension is validated in Section 5 using typical kernels that contain indirections. Section 6 discusses related work. Finally, Section 7 is devoted to our conclusions.

2. THE PROBABILISTIC MISS EQUATIONS (PME) MODEL

The PME model [Fraguela et al. 2003] classifies misses in two groups. Compulsory or cold misses take place the first time a given memory line is accessed, since lines are loaded in the cache on demand. Interference misses take place when a line that has been accessed previously is not found in the cache in a new access. This category includes all the conflict and capacity misses. An attempt to reuse a line results in a miss with a probability that depends on the cache footprint of the data accessed since the previous reference to the considered line. Thus, the PME model estimates the number of misses generated by each static reference found in a code by means of a formula, called probabilistic miss equation, which includes the number of different lines it accesses (compulsory misses), the number of line reuses it generates, and the interference probability for such accesses (interference misses) during the execution of the program.

Normally, each given line can be reused with different reuse distances, that is, different portions of code are executed in between different attempts to reuse the line. In the case of references found in loop nests, which is the scope of the PME model, each loop enclosing a reference gives place to a different reuse distance, which can be measured in terms of loop iterations, that (possibly) characterizes some of the reuses not captured by the inner loops. This way, our model estimates the number of misses generated by a reference by exploring the loops that enclose it from the innermost to the outermost one. In each loop the model builds a partial PME that adds information about the reuses whose reuse distance is associated with that loop. Specifically, each partial PME estimates the number of accesses generated by the reference that cannot exploit reuse in the considered loop, the number of accesses whose reuse distance is associated with this loop, and the associated miss probability for such reuses. The PME for each loop and static reference is expressed recursively in terms of the PME

for the same reference in the immediately inner loop, so that it contains all the information for the behavior of the reference within the loop. Thus, the PME associated with the outermost loop in a nest takes into account all the reuses and its evaluation yields the number of misses generated by the reference during the execution of the loop nest.

In our extended model, PMEs continue to be built in this recursive way, but besides the PMEs associated with regular access patterns introduced in Fraguela et al. [2003], which we describe in Section 3.1 for completeness, new kinds of PMEs associated with irregular access patterns are proposed in Sections 3.2 and 3.3.

2.1 Miss Probability Estimation

Our model is probabilistic because PMEs estimate the probability that each given access results in a miss in order to calculate the number of misses. In a K-way set associative cache with LRU replacement policy, an attempt to reuse a line results in a miss if K or more different lines accessed since the last reference to the considered line are mapped to its cache set. As a result, the probability of miss in a nonfirst access is equal to the probability that a cache set has received K or more lines during the reuse distance, that is, the portion of code executed since the immediately previous access to the line. The PME model follows three steps to estimate the interference probability associated with a reuse distance:

1. Access pattern identification: the access patterns followed by the references involved in the reuse distance and the parameters that characterize them are inferred from the references indexing functions and the shape of the loops that enclose them. The PME model represents each pattern as a function whose output is a mathematical representation of the footprint of the access pattern on the cache. There is one function per each typical access pattern (sequential access, access with constant stride, etc.) and its arguments provide the quantitative characterization of the access pattern.
2. Cache impact quantification: the functions identified in the preceding step are evaluated, yielding vectors of probabilities that we call *area vectors*, that represent how each access pattern has impacted on the cache.
3. Area vectors addition: once the area vectors for the different access patterns have been estimated, they must be added in order to calculate a global area vector that represents their total impact on the cache.

Section 4 explains how our model follows these steps. Once they are completed, the final interference probability is estimated as component 0 of the global area vector associated with the analyzed reuse distance, since, as we will see in Section 4.2, it contains the ratio of sets that received K or more lines during the reuse distance, which is conversely the probability a given set has received K or more lines.

2.2 Scope of Application

Figure 1 depicts the scope of application of our extended model. It shows a set of normalized perfectly or nonperfectly nested loops in which the number of

```
DO I0 =1, N0
  DO I1 =1, N1
  ...
    DO IZ =1, NZ
     ...
      A(fA1(IA1), ..., fAj(B(fB1(IB1))), ...)
     ...
    END DO
  ...
  END DO
END DO
```

Fig. 1. Nested loops with structures accessed using indirections.

Table I. Notation Used

C_s	Cache size
L_s	Line size
K	Associativity of the cache
D_A	# of dimensions of array A
D_{Aj}	size of the jth dimension of array A
d_{Aj}	cumulative size of the jth dimension of array A, $d_{Aj} = \prod_{k=1}^{j-1} D_{Ak}$
α_{Rj}	constant that multiplies either a loop index or a value read from an index array in the jth subscript of reference R
δ_{Rj}	constant added either to a loop index or to a value read from an index array in the jth subscript of reference R
N_i	# of iterations of loop at nesting level i, whose index is I_i
S_{Ri}	stride of reference R with respect to the loop at nesting level i, $S_{Ri} = \alpha_{Rj} \cdot d_{Aj}$, where j is the dimension of array A referenced by R indexed by I_i
L_{Ri}	# of different sets of lines (SOLs) accessed by reference R during the execution of the loop at nesting level i
L_{Ri_1}	# of different sets of lines (SOLs) accessed by reference R during the execution of one iteration of the loop at nesting level i
D_{Ri}	# of different sets of lines (SOLs) that reference R can potentially access during the execution of the loop at nesting level i
p_{Ri}	probability each one of the D_{Ri} sets of lines (SOL) R can access is actually accessed during a given iteration of the loop at nesting level i, $p_{Ri} = L_{Ri_1}/D_{Ri}$

iterations of every loop must be the same in every execution of the loop. The reference indexes are affine functions f_i either of the loops control variables I_i or of values read from arrays. We call index or indirection array the one whose values are used to index another array, which we call the base array of the indirection. Index arrays can be themselves indexed by other arrays, which gives place to several levels of indirection. In the codes considered in this work the probability that a component of the base array of an indirection is accessed is uniformly distributed. This means they all have the same probability of being accessed. An extension to consider the situation when the probability of access is concentrated, still uniformly, on a band of the base array, is also presented.

As for the hardware, our model considers set-associative caches of an arbitrary size C_s, line size L_s and associativity K with LRU replacement policy, which is the most common situation. Table I depicts these and other parameters we will make reference to during the explanation of our model. For simplicity,

in all our terms and formulas, sizes and strides are expressed in elements of the array whose access is being analyzed rather than in bytes.

3. BUILDING PROBABILISTIC MISS EQUATIONS

As explained in Section 2, a partial PME F_{Ri} is built for each static reference R in the code and loop at nesting level i that encloses such reference. This PME estimates the number of misses that R generates during a complete execution of this loop as a summatory of the number of accesses that enjoy each possible reuse distance associated with this loop multiplied by the miss probability that the memory regions accessed during that reuse distance generate. Of course, every access that is the first one to a line in this loop, cannot result in reuses of lines already accessed in the current execution of the loop; thus, their miss probability cannot be associated to reuse distances within the loop. The miss probabilities for those accesses correspond either to (a) reuse distances that are associated with outer loops; or (b) reuse distances with respect to accesses to the same data in previous loops in the same nesting level, when we consider nonperfectly nested loops; or (c) when the loop is the outermost one ($i = 0$) and there are no preceding loops that could give place to reuses, the miss probability is simply one, since every first access to a line in this loop is, indeed, a first access to the line, unable to exploit any reuse, which results in a compulsory miss. Since PMEs are built beginning in the innermost loop and proceeding outward and their evaluation depends on memory regions associated with reuses that are calculated in outer or previous loops, the general expression of a PME is F_{Ri}(RegIn), where RegIn stands for the memory regions accessed during the reuse distance for what in this level of the nest happen to be first accesses. The exception are the PMEs for outermost loops F_{R0} in which no reuse from previous accesses is possible, for whose evaluation we use as RegIn a memory region whose associated miss probability is one, so that the first-time accesses to a line in the nest are predicted as misses. In general, we can define the input parameter RegIn of a PME F_{Ri} as the memory region accessed since the immediately previous access to any of the lines that R references in loop i in the moment the execution of the loop begins.

The construction of F_{Ri} depends on whether the control variable for loop i, $\mathtt{I}_i$, is used in the indexes of index arrays found in the reference or not. If $\mathtt{I}_i$ does not appear in R, or if it only appears in the indexes that do not depend on indirections, the access pattern of R is regular with respect to loop i, so the PME for this loop is built as in Fraguela et al. [2003]. This kind of PME is explained in Section 3.1 below for the sake of completeness. If, on the contrary, $\mathtt{I}_i$ participates in the indexing of an index array in R, thus giving place to an indirection, the access pattern of R is irregular with respect to loop i. Modeling the behavior of irregular access patterns requires new kinds of PMEs. We have identified two kinds of irregular PMEs in the presence of accesses with an uniform distribution. Monotonic irregular access PMEs, explained in Section 3.2, model the situation when the accesses generated by the indirection are ordered, i.e., when the values read from the index array are monotonically increasing or decreasing. When this condition does not hold or we simply do not have

information about the indexing values, nonmonotonic irregular access PMEs are to be applied (see Section 3.3). We now explain the three kinds of PMEs, in turn.

3.1 Regular Access PME

If the variable $\mathtt{I}_i$ associated with loop i does not index any structure that is used in the indexing of the base array A of the studied reference R, the access pattern of R is regular with respect to loop i. Thus, the behavior of R in this nesting level is modeled by the regular access PME, explained in Fraguela et al. [2003]:

$$F_{Ri}(\text{RegIn}) = L_{Ri} \cdot F_{R(i+1)}(\text{RegIn}) + (N_i - L_{Ri}) \cdot F_{R(i+1)}(\text{Reg}_{Ri}(1)) \qquad (1)$$

where N_i is the number of iterations of the loop at the nesting level i and L_{Ri} is the number of iterations in which there is no possible reuse for the lines referenced by R from the point of view of this loop. $\text{Reg}_{Ri}(j)$ stands for the memory region accessed during j iterations of the loop in the nesting level i that can interfere with the accesses of R in the cache.

The formula calculates the total number of misses for reference R in nesting level i as the sum of two values. The first one is the number of misses produced by the L_{Ri} iterations in which the accesses of R cannot exploit reuse in this loop. The miss probability for these iterations depends on reuse distances generated in outer or preceding loops. Thus, the number of misses generated in these iterations is obtained evaluating $F_{R(i+1)}$, the PME for the immediately inner loop, passing as parameter for the calculation of the miss probability of its first accesses the value RegIn provided by those external loops. The second value corresponds to the iterations in which there can be reuse with respect to the accesses in the previous iteration in this loop. The miss probability for the first accesses in the evaluation of the PME for the immediately inner level depends, in this case, on the memory regions accessed during one iteration of loop i.

When this formula is applied to the innermost loop containing reference R, these L_{Ri} iterations correspond to lines, meaning that during one complete execution of the N_i iterations of the innermost loop, R really accesses L_{Ri} different lines, the other accesses being thus reuses. When the loop analyzed is not the innermost one, the iterations of the loop define sets of lines (SOLs) accessed by R in the inner loops. For example, if a bidimensional $M \times N$ FORTRAN array is accessed row by row (that is, the innermost loop of the access sweeps through the N columns of a given row), in the analysis of the outer loop that controls the row index of the reference, each iteration of this loop is associated to the access to the set of lines that hold the elements of a row of the matrix. As FORTRAN stores the arrays by columns, if $M \geq L_s$, where L_s is the cache line size measured in elements, which is the most usual situation, each set of lines will be made up of N different lines. In this case, L_{Ri} iterations of this outer loop give place to accesses to new sets of lines (SOLs), while the other $N_i - L_{Ri}$ iterations generate reuses of the SOLs accessed in the previous iteration. In what follows, we will talk in general about sets of lines (SOLs), in the understanding that in the innermost loop each one of these sets consists of a single line.

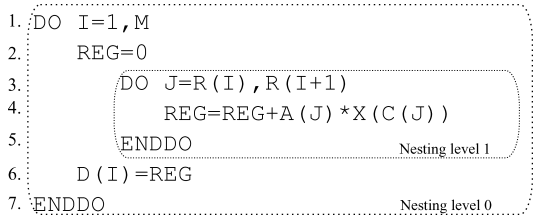


Fig. 2. Sparse matrix–vector product.

The number of iterations of loop i that cannot exploit either spatial or temporal locality is given by

$$L_{Ri} = 1 + \left\lfloor \frac{N_i - 1}{\max\{L_s/S_{Ri}, 1\}} \right\rfloor \tag{2}$$

where L_s is the line size measured in elements of the array referenced by R and S_{Ri} is the stride that reference R has with respect to loop i. This stride is a constant, since either I_i does not index reference R or the index we are considering is an affine function of $\mathtt{I}_i$. In the former case, trivially $S_{Ri} = 0$. In the latter case $S_{Ri} = \alpha_{Rj} d_{Aj}$, where j is the dimension whose index depends on $\mathtt{I}_i$; α_{Rj} is the scalar that multiplies the loop variable in the affine function, and d_{Aj} is the cumulative size[1] of the jth dimension of the array A referenced by R.

Example 1. We will use as ongoing example to illustrate the construction of the different kinds of PMEs the sparse matrix–vector product code in Figure 2, where the matrix is stored in CRS (compressed row storage) format [Barrett et al. 1994]. As we can see, this storage gives place to irregular accesses on the vector X by which the matrix is multiplied.

If we analyze the reference $R =$ D(I) in the context of loop I, at nesting level 0, we see that the variable that controls the loop indexes this reference by means of the affine function $1 \times \mathtt{I} + 0$. Thus, the regular access PME of Eq. (1) can model the behavior of the reference in this loop. In order to apply it, we must first calculate the number L_{R0} of different sets of lines this reference accesses during the execution of this loop by means of Eq. (2). Since the number of iterations of this loop is $N_0 = M$ and the stride S_{R0} of D(I) with respect to loop I is 1, we get $L_{R0} = 1 + \lfloor (M-1)/L_s \rfloor$, which matches our intuitive calculation of the number of different lines that the reference accesses during the M iterations of the loop. If we now replace this L_{R0} in Eq. (1) we get

$$\begin{aligned} F_{R0}(\text{RegIn}) \;=\; & (1 + \lfloor (M-1)/L_s \rfloor) F_{R1}(\text{RegIn}) \\ & + (M - (1 + \lfloor (M-1)/L_s \rfloor)) F_{R1}(\text{Reg}_{R0}(1)) \end{aligned} \tag{3}$$

that is, the probability of miss in the first access to each one of the L_{R0} different lines of D accessed in this loop depends on the impact on the cache of RegIn regions accessed in outer or previous loops that could generate interferences

[1]Let A be an N-dimensional array of size $D_{A1} \times D_{A2} \times \ldots D_{AN}$. We define the cumulative size for its jth dimension as $d_{Aj} = \prod_{i=1}^{j-1} D_{Ai}$.

with our attempt to reuse these lines of D, if such lines have been already accessed at some point in the program. If our analyzer does not find a previous access to these lines, it will evaluate the PME using as RegIn a region such that it yields a 100% miss probability for these first accesses to the lines of D. The probability of miss for the remaining $M - L_{R0}$ accesses depends on the regions accessed during one iteration of this loop, since the attempts to reuse each line happen with a reuse distance of one iteration.

3.2 Monotonic Irregular Access PME

When the control variable for loop i, $\mathtt{I}_i$, indexes an index array in an indirection, the access pattern on the base array of our reference R is irregular with respect to loop i. The reason is that the position accessed by R no longer depends directly on $\mathtt{I}_i$, but on the value read from the array that $\mathtt{I}_i$ indexes either directly or through more levels of indirection.

The distribution of the values read from the index arrays on the dimension of the base array they index determines the accesses, the reuses, and thus the PME that models the reference–cache interaction. In our modeling, we assume that this distribution is uniform, that is, all the elements of the base array have the same probability of being accessed in each iteration of the considered loop.

We have found that two classes of irregular access patterns arise depending on whether the values of the considered index array are ordered or not. When they are ordered, the sequence of accesses produced by the indirection can be characterized as a monotonically increasing or decreasing function. In this case, the reuses in the considered loop i can only take place with respect to the line referenced in the immediately previous iteration. This way, the PME for regular access patterns explained in the preceding section (Eq. 1) can be used in this situation, the difference being that L_{Ri}, the number of iterations of this loop that cannot exploit reuse, or conversely, the number of different sets of lines (SOLs) that R accesses during the execution of the loop, cannot be estimated as in the regular access pattern case. In a monotonic irregular access pattern,

$$L_{Ri} = D_{Ri}\left(1 - (1 - L_{Ri_1}/D_{Ri})^{N_i}\right) \tag{4}$$

where D_{Ri} is the number of different SOLs that R can potentially access during the execution of the loop i and L_{Ri_1} is the number of SOLs accessed during one iteration of loop i. The rationale for Eq. (4) is that if in each iteration of the loop i, on average L_{Ri_1} different SOLs are accessed out of the D_{Ri} ones that R could access, then each one of them has the same uniform probability $p_{Ri} = L_{Ri_1}/D_{Ri}$ of being accessed in each iteration of the loop. Thus, the probability that a SOL has been accessed at least once during the N_i iterations of the loop is $1 - (1 - p_{Ri})^{N_i}$. Multiplying this probability by the number of SOLs yields the average number L_{Ri} of different SOLs that are actually referenced. Thus, this is the number of iterations of the loop in which no reuse is possible. Because the values in the index array are monotonically increasing (or decreasing), the other $N_i - L_{Ri}$ iterations of the loop attempt to reuse the SOL accessed in the immediately previous iteration, with a reuse distance of one iteration of the loop, as PME (1) reflects.

The number L_{Ri_1} of different SOLs accessed during one iteration of loop i is trivially one in the innermost loop z that contains R. For any other loop i, L_{Ri_1} is L_{Rk}, with $k = \min\{v / i < v \leq z \wedge \text{DimInd}(v) = \text{DimInd}(i)\}$, i.e., it is the L_R for the outermost loop k nested inside loop i such that its index variable $\texttt{I}_k$ indexes (indirectly) the same dimension of the base array `A` referenced by our reference R as the variable $\texttt{I}_i$ of the considered loop i. If no such loop exists, then, again, $L_{Ri_1} = 1$. Another way to express it is that the number of different SOLs accessed in one iteration of loop i is the number of SOLs accessed during the complete execution of the outermost loop nested inside loop i that indexes, indirectly, the same dimension of the affected base array as $\texttt{I}_i$. This definition allows to handle correctly those cases in which, for example, the indirection for a given dimension in R depends on several loop index variables, e.g., in `A(B(I,J))` both `I` and `J` index indirectly the only dimension of vector `A`. Another example for this situation is often found in the codes in which indirections are generated by sparse matrices because of the formats used to store them.

Example 2. If we analyze the sparse matrix–vector product code in Figure 2, we see that vector `X` is accessed indirectly through `C(J)` in the innermost loop, whose index variable is precisely `J`. In that loop, trivially, $L_{R1_1} = 1$ for reference `X(C(J))`. If we analyze the outer loop on `I`, we can see that this variable indexes `R(I)`, which defines the values for `J`. As a result, the indexes of both loops indirectly index the only dimension of vector `X` and, thus, for the outermost loop 0, $L_{R0_1} = L_{R1}$.

We complete our modeling for this access pattern with the expression of D_{Ri}:

$$D_{Ri} = \left\lceil \frac{D_{\mathrm{A}j} d_{\mathrm{A}j}}{\max\{S_{Ri}, L_s\}} \right\rceil \tag{5}$$

where $S_{Ri} = \alpha_{Rj} \cdot d_{\mathrm{A}j}$ and $d_{\mathrm{A}j}$ are defined as in the preceding section, and $D_{\mathrm{A}j}$ is the size or number of elements along the jth dimension of the array `A` referenced by R. Let us remember that j is the dimension that is indexed, in this case indirectly, by $\texttt{I}_i$. This also means that in this case the constant α_{Rj} is multiplying the indirection indexed by $\texttt{I}_i$ rather than the variable $\texttt{I}_i$ itself.

Example 3. From the shape of the loops displayed in Figure 2, a compiler can speculate that `R` stores the indices for the beginning of the data of each row of the sparse matrix in `A` and `C`, which hold the nonzeros and their corresponding columns, respectively. Another possibility to extract this information would be to include a directive to the compiler in the code reporting which is the role of each array in the storage of the sparse matrix. With this knowledge we can also infer that the sparse matrix has M rows and we can speculate that the values in `C` are ordered for each row. If this were the case we could conclude that the values read in Figure 2 by `C(J)` are monotonically increasing during each whole execution of the loop `J`, at nesting level 1. As a result, the access pattern of `X(C(J))` in this loop can be modeled by a monotonic irregular access PME. This PME has the form of Eq. (1), with its L_{Ri} calculated according to Eq. (4). The latter expression is a function of D_{R1}, the number of different SOLs that R

can potentially access during the execution of the loop J, and L_{R1_1}, the number of SOLs accessed during each iteration of this loop.

Equation (5) allows to calculate D_{R1} knowing that (a) the indirection takes place in the first dimension of the base array X ($j = 1$), (b) the cumulative size for the first dimension of any array is always one ($d_{\mathrm{X1}} = 1$), (c) the stride S_{R1} of our reference with respect to its indirection is one ($S_{R1} = \alpha_{R1} \cdot d_{\mathrm{X1}} = 1 \cdot 1$), and (d) the size of the first (and only) dimension of X is a value D_{X1} our compiler extracts from the definition of the vector in the code. With these data, we evaluate Eq. (5) as $D_{R1} = \lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil$. This means that during each iteration of the loop J, X(C(J)) could potentially access any of the $\lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil$ lines that constitute X.

Both in our general explanation about the calculation of L_{Ri_1} and in our preceding example, we explained that trivially, in the innermost loop that contains a reference R with an indirection, $L_{Ri_1} = 1$, which is the case for X(C(J)) in loop J.

With these two pieces of data, we can evaluate Eq. (4):

$$L_{R1} = \lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil \left(1 - \left(1 - \frac{1}{\lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil} \right)^{N_1} \right) \tag{6}$$

This expression assumes that each one of the $\lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil$ lines of X has the same uniform probability of being accessed during each one of the N_1 iterations of loop J. As a result, after the N_1 iterations, each line has a probability $1 - (1 - 1/\lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil)^{N_1}$ of having been accessed at least once. Thus, multiplying this probability by the number of lines, we get the number of different lines that were actually accessed on average. As for the average number of iterations of this loop N_1, since it sweeps along the elements of a row of the sparse matrix, its value is $N_i = Nnz/M$, where Nnz is the number of nonzeros in the sparse matrix and M its number of rows. The number of nonzeros can be assumed from the size declared for the arrays A and C, or be part of a directive to the compiler or be extracted from a profiling of the input data.

Once we have calculated the number L_{R1} of different SOLs accessed in each execution of the loop (with each SOL consisting of a single line in this case), we can replace it in Eq. (1). This equation will consider the L_{R1} first accesses to a different line with a miss probability that depends on reuses that take place with respect to accesses outside the loop, while the remaining $N_1 - L_{R1}$ accesses necessarily try to reuse the line accessed in the immediately preceding iteration. As a result, the miss probability for them is associated to the regions accessed during one iteration of this loop.

3.3 Nonmonotonic Irregular Access PME

When the indexing values are not monotonic, or we have no information about their ordering, the last access of a reference to a given line, or, in general, set of lines (SOL), in the considered nesting level i may have happened an indeterminate number of iterations ago. The number of loop iterations between two accesses of the reference to the same SOL is not a fixed value, since every SOL can be accessed with a given probability in each iteration of the loop. Thus,

a probabilistic approach must be followed to estimate the number of misses taking into account that each potential reuse distance happens now with a different probability.

In the presence of uniform probabilities, each one of the D_{Ri} different SOLs that R can potentially access during each execution of the loop has the same probability $p_{Ri} = L_{Ri_1}/D_{Ri}$ of being accessed in a given iteration, no matter whether the accesses are monotonic or not. Also, every SOL has this probability of access in each one of the N_i iterations of the loop. As a result, the number of misses generated by a nonmonotonic irregular access pattern during the execution of loop i can be estimated by means of a summatory in which each term estimates the number of misses that the accesses of R can generate in the jth iteration of the loop:

$$F_{Ri}(\text{RegIn}) = \sum_{j=1}^{N_i} WM_{Ri}(\text{RegIn}, j) \tag{7}$$

where $WM_{Ri}(\text{RegIn}, j)$ yields the weighted number of misses generated in the jth potential access of R to the SOLs it defines in loop i. In this expression, RegIn stands for the region accessed since the last reference to the SOLs that R accesses in this loop when the execution of this loop begins, as usual. This number of misses is calculated as

$$\begin{aligned} WM_{Ri}(\text{RegIn}, j) \;=\; & (1 - p_{Ri})^{j-1} \cdot F_{R(i+1)}(\text{RegIn} \cup \text{Reg}_{Ri}(j-1)) \\ & + \sum_{h=1}^{j-1} p_{Ri} \cdot (1 - p_{Ri})^{h-1} \cdot F_{R(i+1)}(\text{Reg}_{Ri}(h)) \end{aligned} \tag{8}$$

where $p_{Ri} = L_{Ri_1}/D_{Ri}$, as explained in the previous section, yields the probability that a given SOL of the base array that R can potentially access during the execution of loop i is indeed accessed during one iteration of that loop.

The first term in Eq. (8) considers the case that the SOL has not been accessed in any of the previous $j-1$ iterations. This happens with a probability which is $(1 - p_{Ri})^{j-1}$ given that p_{Ri} is the probability of access in each iteration. In this case, the RegIn region that could generate interference with the new access to the line when the execution of the loop begins must be added to the $\text{Reg}_{Ri}(j-1)$ regions accessed during these $j-1$ previous iterations of the loop in order to account for the complete interference region. This addition is represented by means of the $\cup$ operator. The second term weights the probability that the last access took place in each one of the $j-1$ previous iterations of loop i. The probability that the last access to a given SOL was exactly h iterations before the current iteration is $p_{Ri}(1 - p_{Ri})^{h-1}$, that is, the probability there was an access to the SOL h iterations ago, but there were no accesses to it during the last $h-1$ iterations. In this case, the regions that can generate interferences with the attempt to reuse the SOL in the current iteration are those accessed during those h intermediate iterations, $\text{Reg}_{Ri}(h)$.

Example 4. When the reference `X(C(J))` in our example code of Figure 2 is analyzed in the context of the outer loop `I` at nesting level 0, the values read from the indirection are no longer guaranteed to be ordered throughout the execution

of the loop. That is, the values read from C during a single iteration of the loop I correspond to the column indexes of the elements of a single row, which we can assume that have been stored in a given order; but when the whole loop I is taken into account, the values read from C are not ordered among different iterations of loop I. As a result, the nonmonotonic irregular access PME of Eq. (7) characterizes the access to X in this loop. In that formula, the number of iterations of the loop is $N_0 = M$, in our case, and WM_{R0}(RegIn, j) is calculated following Eq. (8). In order to evaluate the latter formula, we must calculate p_{R0}, that is, the individual probability each SOL of X is accessed in each iteration of loop I. As we have explained, this value is derived as $p_{R0} = L_{R0_1}/D_{R0}$, where L_{R0_1} is the number of different SOLs our reference accesses, on average, in each iteration of the loop and D_{R0} is the number of different SOLs it could actually access. In example 2, we explained and calculated that for this reference $L_{R0_1} = L_{R1}$, and the value of L_{R1} was estimated in Eq. (6) in example 3. Regarding D_{R0}, it is calculated, according to Eq. (5). As we explained in example 2, while the variable I that controls the loop we are analyzing does not appear in the expression of our reference X(C(J)), this variable indexes R(I), which defines the values for J. This way, I indexes indirectly the indirection we are analyzing in the first (and only) dimension of array X and Eq. (5) can be evaluated using the same parameters used in example 3, which results in $D_{R0} = D_{R1} = \lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil$. That is, any of the $\lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil$ lines of X can be accessed during the execution of loop I, where we remind the reader that D_{X1} is the length of vector X and L_{s} is the number of elements of vector X a cache line can hold.

This example helps us also to illustrate the meaning and usage of the RegIn input for the PMEs. The PME F_{R0} for reference R =X(C(J)) we have just built is based on Eq. (7). In its development in Eq. (8), we can see how, as always, this PME is expressed in terms of the PME for the same reference in the inmediately inner loop. In our case, this PME is F_{R1}, built in example 3, which models the behavior of the accesses to X during the product by a row of the sparse matrix. The evaluations of $F_{R(i+1)}$ in F_{Ri} receive as RegIn the set of regions accessed during the reuse distance associated to that evaluation. In our example, attending to Eq. (8), F_{R0} evaluates F_{R1} through WM_{R0}(RegIn, j) with two kinds of reuse distances. The input for the first appearance of F_{R1} in this expression depends on the RegIn for F_{R0} itself, because it is not associated to reuses within the loop. Rather, it corresponds to the first accesses to lines of X during the execution of the loop, which will result in cold misses. The model predicts this correctly because (a) RegIn for outer loops with no preceding accesses is a region with an associated miss probability 1 and (b) as we can see the model propagates this region down to the PME F_{R1} for the innermost loop for the evaluation of the misses generated in the very first accesses to these lines.

The remaining evaluations of F_{R1} in Eq. (8) correspond to reuses within loop 0 with a reuse distance of exactly h iterations of this loop each. Such evaluations are multiplied by the probability this situation actually takes place in order to predict correctly the number of misses they generate. Their RegIn is $\mathrm{Reg}_{R0}(h)$, i.e., the interference region generated during those h iterations in which a line of X has not been accessed. In our example, this corresponds to the acesses that take place during the product of h rows of our sparse matrix

by the vector. The RegIn of F_{R1} determines the miss probability for the first accesses to the lines of X during an isolated iteration of the innermost loop. Assigning this value to RegIn ensures such probability depends, in fact, on the cache footprint of the accesses performed since the inmediately preceding access to those lines, which took place exactly h iterations of the loop on I ago.

The calculation of regions of interference and the quatitative evaluation of PMEs are considered in Section 4.

3.4 PME for Uniform Band Distribution

Until now, we have considered the case in which all the elements of the base array have the same probability of being accessed, but our model can be extended to cover situations in which the distribution is not uniform. For example, a very common source of indirections are accesses generated by sparse matrices that are stored in some compressed format like CRS [Barrett et al. 1994]. One of the most usual situations, by far, is that such matrices are banded,[2] so it is valuable to extend our model to consider irregular accesses that are restricted to a limited band or area of the base array, even if the probability of access is still uniform inside such band. In this case, the formulas described in the Sections 3.1, 3.2, and 3.3 can be used making two small changes to adapt them to this new situation:

- when PMEs for the indirect accesses generated by the column indices of a banded matrix are built, the term $D_{\mathrm{A}j}$ in Eq. (5) must be replaced by the size of the band of the studied matrix, since the accesses are not uniformly distributed on the whole jth dimension of the base array, but only of the region associated with the band B of the matrix.
- since the nonzeros are only distributed along B rows in each column and B columns in each row, when the probability of reuse of a group of SOLs with respect to the preceding iterations is considered in Eq. (7), the upper bound of the summatory is not N_i, the size of the sparse matrix along the considered dimension that gives place to the attempts of reuse, but B, since only along B rows/columns can be the same SOL of the base array be reused.

Example 5. The model derived for matrices with an uniform distribution for our example code in Figure 2 is applicable to banded matrices except in two points. First, in the calculation of D_{R0} and D_{R1} for reference X(C(J)), we must substitute the value of $D_{\mathrm{X}1}$ with the band size. In the expression F_{R0} that characterizes the behavior of this reference in the outer loop at nesting level 0 (the one indexed by I), which has the shape of Eq. (7), the upper bound of the summatory must also no longer be M, the total number of rows of the sparse matrix, but its band size B, since only along the processing of B different rows of the input matrix can we exploit reuse of a given line of the base array X of this reference. The size of the band would have to be provided by a directive to the compiler or be extracted by an analysis of the input data.

[2] A is banded with bandwidth $B = 2p + 1$ if all the nonzeros are contained within the first p super and first p subdiagonals ($A_{ij} = 0, |i - j| > p$).

```
function Reg_Ri(n) {
   RegSet = ∅
   foreach array A involved in the code {
      R_A = ∅
      foreach reference R' to array A in loop i or loops nested inside it {
         R_A = merge(region_accessed(R, i, n), Q_R, R_A)
      }
      reg_A.region_func = access_pattern(R_A)
      reg_A.is_self_interference = (A == array(R))
      RegSet = RegSet ∪ reg_A
   }
   return RegSet
}
```

Fig. 3. Calculation of $Reg_{Ri}(n)$, the set of regions that can interfere with the attempts of reuse of reference R generated during n iterations of the loop at nesting level i.

4. ESTIMATION OF MISS PROBABILITIES

The formulas described in the preceding section collect the number of accesses that can exploit each potential reuse distance as well as the set of memory regions that have been accessed during such reuse distance. This way, during the construction of each partial PME, the first stage of the process to estimate miss probabilities introduced in Section 2.1, access pattern identification, is performed. These memory regions, or, conversely, the access patterns that reference them, generate a miss probability for the attempts to reuse lines by the analyzed reference R that is calculated by means of the last two steps of the procedure described in Section 2.1 when the evaluation of its PME recursively reaches the innermost loop i that contains the reference. In this loop, the PME recurrence finishes defining $F_{R(i+1)}$(RegIn) as the miss probability that the impact on the cache of the memory region(s) RegIn generate on the reuses of reference R. To estimate this probability the different access patterns collected in RegIn are mapped in the cache impact quantification step to area vectors, which are vectors of probabilities. Finally, these vectors are added to generate a global area vector in the area vectors addition step. The global area vector gathers the combined effect on the cache of the accesses to all of the regions in RegIn and its component 0 is the miss probability we are estimating, as Section 2.1 explains. We now describe, in more detail, the three steps of the miss probability estimation process.

4.1 Access Pattern Identification

Every PME F_{Ri} requires the estimation of the set of regions $Reg_{Ri}(n)$ accessed during n iterations of the considered loop that can interfere with the reuses of R. Figure 3 shows the pseudocode for its calculation: the access pattern of the references to each array A found within the loop is identified, in turn, and added to the set of regions accessed. The memory region associated to the same array R accesses is marked, because its cache impact quantification step is different, as we will see in the next section.

Figure 4 shows the two steps involved in the identification of the access pattern that references follow during a reuse distance consisting of n iterations of the loop at nesting level h. First, for each reference R the indexes of each dimension and the number of iterations of each loop during this reuse distance

```
function region_accessed(R, h, n) {
   A = array(R)
   foreach dimension j of A {
      i = loop_that_indexes_dimension(R, j)
      if not indexed_by_indirection(R, j) {
         R_Rj(h, n) = (iters_regular(i, h, n), S_Ri, 1)
      } else {
         R_Rj(h, n) = (iters_irregular(i, h, n), S_Ri,
                                        prob(R, i, h, n))
      }
   }
   return simplify(R_R(h, n))
}
```

(a) Calculation of $\mathcal{R}_R(h,n)$, the tuple-based representation of the region accessed by reference R during n iterations of the loop at nesting level h.

```
function access_pattern(R) {
   if R == (M, S, P) {
      if P == 1.0
         if S == 1
            return Reg_s(M)
         else
            return Reg_r(M, 1, S)
      else
         if S == 1
            return Reg_sp(M, P)
         else
            return Reg_rp(M, 1, S, P)
   } else {
      if R == ((M_1, 1, P_1), (M_2, S_2, P_2))
         return Reg_rp(M_2, M_1, S_2, P_1 · P_2)
      else
         return error(UNKOWN)
   }
}
```

(b) Identification of the access pattern associated to the memory region described by the tuple(s) $\mathcal{R}$.

Fig. 4. Identification of the access pattern followed by the references during a reuse distance.

are examined. The output of this analysis is a D_{A}-tuple $\mathcal{R}_R(h,n)$, where D_{A} is the number of dimensions of the array `A` referenced by R. Each element of this tuple consists, in its turn, of a 3-tuple $\mathcal{R}_{Rj}(h,n) = (M_j, S_j, P_j)$, where the M_j is the number of different points accessed along dimension j, S_j the constant stride between two consecutive points, and P_j the probability each one of these points is actually accessed by R. The algorithm followed to calculate the 3-tuple associated to dimension j of reference R during n iterations of the loop at nesting level h is the following:

- if no indirections are involved in the indexing of the dimension, its index is an affine function $\alpha_{Rj}\ \mathtt{I}_i + \delta_{Rj}$ of some loop index $\mathtt{I}_i$. In this case, the set of points accessed in this dimension by R can be represented as the tuple $(\mathrm{Iters}_i(h,n), S_{Ri}, 1)$, where $\mathrm{Iters}_i(h,n)$ is the number of different values that $\mathtt{I}_i$ takes during n iterations of the loop in nesting level h. This value is calculated as

$$\mathrm{Iters}_i(h,n) = \begin{cases} 1 & \text{if } i < h \\ n & \text{if } i = h \\ N_i & \text{if } i > h \end{cases}$$

- if the indexing of dimension j depends on an indirection, that is, the index has a shape $\alpha_{Rj}\ \mathtt{B}(f(\mathtt{I}_i)) + \delta_{Rj}$, we assume that the accesses may be spread uniformly on the affected dimension of the array. Since our indirection is multiplied by some constant α_{Rj} (usually one), there are $\lfloor D_{\mathrm{A}j}/\alpha_{Rj} \rfloor$ different points in the dimension that can be actually accessed (e.g., reference `A(2*B(I))` can only access the even elements of array `A`). Each point has an uniform probability $1/\lfloor D_{\mathrm{A}j}/\alpha_{Rj} \rfloor$ of being the one accessed because of each given value read from the index array. As a result, if $Nindex_i(h,n)$ different values have been read from the index array `B` during n iterations of the loop at nesting level h, the average probability each that each one of the points that R can access in the jth dimension of its base array has been accessed

at least once is $1-(1-1/\lfloor D_{\mathrm{A}j}/\alpha_{Rj}\rfloor)^{Nindex_i(h,n)}$. Thus

$$\mathcal{R}_{Rj}(h,n)=\left(\left\lfloor\frac{D_{\mathrm{A}j}}{\alpha_{Rj}}\right\rfloor,\quad S_{Ri},1-\left(1-\frac{1}{\lfloor D_{\mathrm{A}j}/\alpha_{Rj}\rfloor}\right)^{Nindex_i(h,n)}\right) \tag{9}$$

As for $Nindex_i(h,n)$, it can be calculated from the analysis of the access pattern for the index array of the reference we are considering: once $\mathcal{R}_R(h,n)$ has been calculated for the reference that reads the indexing array `B`, it is straightforward that the number of different points the reference has accessed is $\prod_{k=1}^{D_{\mathrm{B}}} M_k P_k$, i.e., the product of the number of different points it may access in each dimension multiplied by the probability such access actually takes place.

Once the D_{A}-tuple $\mathcal{R}_R(h,n)$ that represents the region of array `A` accessed by R during n iterations of the loop at nesting level h has been calculated, some simplifications may be applied between pairs of 3-tuples $\mathcal{R}_{Rj}(h,n)$ that describe the access pattern in different dimensions of the array:

$$\begin{aligned}((1,S_j,P_j),(M_k,S_k,P_k)) &= (M_k,S_k,P_j\cdot P_k)\\ ((M_j,S_j,P_j),(M_k,M_j\cdot S_j,P_k)) &= (M_j\cdot M_k,S_j,P_j\cdot P_k)\end{aligned}$$

After these simplifications, a single 3-tuple (M,S,P) that describes the region accessed by the reference is typically obtained.

The notation described above suffices for the representation of memory regions in codes in which there is a single reference per data structure. In codes in which several references access the same data structure, the regions they access will often overlap or be adjacent, so we have developed simple algorithms to merge the descriptors for overlapping or adjacent regions. This way, lines that are accessed by different references are not taken into account several times as source of interferences. In order to perform this merging, one more parameter is used to describe the region affected by a given reference R: the position Q_R with respect to the beginning of the array of the first element it contains. The merging algorithm is applied in function *merge* in Figure 3 and it is not shown here because of space limitations.

As Section 2.1 explains, rather than this description of the memory region accessed, the output of the access pattern identification step is a function that characterizes the access pattern whose output is the area vector associated to it. Depending on the values of S and P in a tuple $\mathcal{R}$, four kinds of access pattern functions can be identified (see Figure 4b):

1. When $P=1$, the access pattern is regular, so it is already covered in Fraguela et al. [2003]. Depending on the value of S we distinguish:
   a. If $S=1$, it is an access to M consecutive elements. We denote the function that calculates the area vector associated to a region of M consecutive elements as $\mathrm{Reg_s}(M)$.
   b. Otherwise, it is an access to a set of M regions of one element separated by a constant stride S. Such access pattern is represented by the function $\mathrm{Reg_r}(M,1,S)$.

2. When $P < 1$, the access pattern is irregular, as each point involved in the pattern has only a certain probability of being actually accessed. As a result, we define probabilistic counterparts for Reg_s and Reg_r:
   a. If $S = 1$, it is an access to M consecutive elements in which each element is accessed with a probability P. The function that calculates the area vector for this access is $\text{Reg}_\text{sp}(M, P)$.
   b. Otherwise, the access affects M different points separated by a constant stride S, which each element is accessed with a probability P. The area vector associated to this access pattern is estimated by function $\text{Reg}_\text{rp}(M, 1, S, P)$.

The most general function in this classification is Reg_rp, all the other ones being specializations of this one. Similarly Reg_s functions are specializations for $S = 1$ of their Reg_r counterparts and the area vector functions that depend on a probability of access P yield the same output as their regular counterparts for $P = 1$. Still, we find this classification useful because regular access patterns enable simpler and faster algorithms for the calculation of their associated area vector than irregular access patterns, and the same happens with the Reg_s functions with respect to their Reg_r counterparts with input stride one.

Example 6. In our example 1 in Section 3.1, the PME F_{R0} that models the behavior of the reference `D(I)` in the loop at nesting level 0 in the sparse matrix–vector product code of Figure 2, that is, the outermost loop of the loop nest, was built in Eq. (3). One of the terms of this PME evaluates the PME F_{R1} for this reference in the immediately inner loop passing as RegIn parameter $\text{Reg}_{R0}(1)$, i.e., the set of memory regions accessed during one iteration of this loop that may generate interferences with the reuses of this reference. Let us identify these regions:

- In each iteration of loop `I`, a whole execution of the loop `J` takes place. As we reasoned in Example 3, the average number of iterations of this loop is $N_1 = Nnz/M$, where Nnz is the number of nonzeros in the sparse matrix and M is its number of rows. Reference `A(J)` is indexed by the variable that controls this loop, so it sweeps along N_1 different elements with stride 1 with probability one. Thus, its $\mathcal{R}_{R1}(0, 1) = (N_1, 1, 1)$, whose area vector can be estimated by $\text{Reg}_\text{s}(N_1)$
- Reference `C(J)` follows exactly the same access pattern; thus, it also accesses a region $(N_1, 1, 1)$, whose associated area vector is estimated by $\text{Reg}_\text{s}(N_1)$.
- Reference `X(C(J))` is indexed by the variable of the loop indirectly, through a read from vector `C`. This way, applying Eq. (9), its $\mathcal{R}_{R1}(0, 1)$ is estimated as $(D_{\text{X1}}, 1, 1 - (1 - 1/D_{\text{X1}})^{N_1})$. The simplest function that can estimate the area vector for this access pattern is $\text{Reg}_\text{sp}(D_{\text{X1}}, 1 - (1 - 1/D_{\text{X1}})^{N_1})$.
- During one iteration of loop `I`, reference `D(I)` also accesses a single element of vector `D`; thus, its $\mathcal{R}_{R1}(0, 1) = (1, 1, 1)$, whose area vector is given by $\text{Reg}_\text{sself}(1)$. The reason for the "self" subindex in the function is that a self-interference area vector will be calculated for this access rather than a cross-interference area vector, as `D(I)` is the very reference whose behavior we

are studying. Section 4.2 explains the difference between both kinds of area vectors.

Finally, although the access pattern functions have been presented based on the values of a single tuple, it is not always possible to reduce the region accessed in an array to a single tuple. All the cases of this kind we have found in the codes we have analyzed had the form $\mathcal{R} = ((M_1, 1, P_1), (M_2, S_2, P_2))$, which can be represented by function $\text{Reg}_{\text{rp}}(M_2, M_1, S_2, P_1 \cdot P_2)$, as they are an access to M_2 separate groups of M_1 consecutive elements each that are separated by a constant stride S_2, having each individual element of the region a probability $P_1 \cdot P_2$ of being accessed.

4.2 Cache Impact Quantification

The functions identified in the previous step are evaluated in order to yield vectors of probabilities called area vectors that represent the impact on the cache of the access they represent. The area vector V associated with a given set of accesses on a cache with associativity K consists of $K + 1$ probabilities $V_0, V_1, \ldots, V_K$. The PME model considers two kinds of area vectors:

- *Cross-interference area vectors* represent the impact on the cache of the considered access pattern as viewed by lines not involved in the access. In these vectors, $V_i, i > 0$ is the ratio of sets that hold $K - i$ lines of the accessed region and V_0 is the ratio of sets that hold K or more lines. These ratios are also conversely the probabilities. For example V_0, is the probability that a set in the cache has received K or more lines accessed by the pattern, V_1 is the probability a cache set has received $K - 1$ lines, and so on.
- *Self-interference area vectors* represent the impact of the footprint on the probability of reuse for the lines it involves. In these vectors, V_0 is the probability that a line of the footprint is competing in its cache set with other K or more lines of the footprint. For $i > 0$, V_i is the probability a line of the footprint shares its cache set with other $K - i$ lines of the access.

Example 7. Let us consider a two-way associative cache with four sets and a reference that has just accessed seven consecutive lines. As a result, three of the four sets contain two of the lines referenced, while the other set contains just one line. The cross-interference area vector generated by this access is $(3/4, 1/4, 0)$, as 3 out of the 4 sets have received two or more lines from the access; only one set received a single line and no sets received zero lines. These ratios are conversely the probabilities a randomly chosen set has two or more, one, or zero lines in it, respectively.

The self-interference area vector for this access is $(0, 6/7, 1/7)$. The first component is zero, as none of the lines involved in the access has to compete for its cache set with other two or more other lines from the footprint. The second component is the ratio of lines of the footprint that share their cache set with exactly one line (6 out of 7). Finally, according to the third component, only one of the seven lines of the footprint does not share its set with any other line of the footprint. These ratios are conversely the probabilities a randomly chosen

line of the footprint has to compete in its set with two or more, one, or no lines, respectively.

The evaluation of access pattern functions for regular access patterns has already been covered in Fraguela et al. [2003]. As we saw in the preceding step, the two most important irregular access patterns found in the presence of uniform probabilities of access are the access to a group of consecutive elements in which each element has an uniform probability of begin accessed (Reg_{sp}) and the access to several groups of elements of this kind that are separated by a constant stride (Reg_{rp}). The mapping of both patterns into area vectors is described in [Andrade et al. 2006]. For the sake of completeness, we will explain the quantification into cross-interference area vector of the first access pattern, which is simpler. For the second one, a high-level description of the algorithm to estimate its cross-interference area vector is provided; further details can be found in Andrade et al. [2006]. The calculation of self-interference area vectors follows very similar steps.

4.2.1 *Sequential Access with Uniform Probability of Access per Element.* Function $\text{Reg}_{sp}(n, p)$ yields the cross-interference area vector $AV_{\text{sp}}(n, p)$ associated with an access to n consecutive elements in which each one of them has an uniform probability p of being referenced. In order to map an access pattern into an area vector, the parameters of the cache must be taken into account. From the point of view of the PME model, a cache is fully characterized by its total cache size C_{s}, its line size L_{s}, and its associativity K. In order to simplify our explanation, both C_{s} and L_{s} are measured not in bytes, but in elements or words of the access we are considering. Given these input parameters, the $K+1$ elements of the area vector $AV_{\text{sp}}(n, p)$ (see Section 2.1 for an explanation of the meaning of each term within an area vector) are calculated as

$$\begin{aligned} &AV_{\text{sp}_i}(n, p) = P(X = K - i) && m < i \leq K \\ &AV_{\text{sp}_m}(n, p) = P(X \geq K - m) && \\ &AV_{\text{sp}_i}(n, p) = 0 && 0 \leq i < m \end{aligned}$$

where $X \in B(n/C_{\text{sk}}, 1 - (1 - p)^{L_{\text{s}}})$, being $B(n, p)$ the binomial distribution,[3] $C_{\text{sk}} = C_{\text{s}}/K$ is the cache size devoted to each level of associativity, and $m = \max\{0, K - \lceil n/C_{\text{sk}} \rceil\}$. The formula is based on the fact that, on average, there are n/C_{sk} lines of the footprint associated to each cache set. Since this is a consecutive memory region, the maximum number of lines a cache set can receive is $\lceil n/C_{\text{sk}} \rceil$, so the area vector elements $AV_{\text{sp}_i}(n, p)$ for $0 \leq i < m$ must be zero. Also, because of the uniform distribution of the accesses, we know that the number of cache lines per set belongs to a binomial $B(n/C_{\text{sk}}, 1 - (1 - p)^{L_{\text{s}}})$. The probability of access per line of this binomial is easy to calculate, since each individual element in a cache line has a probability p of being accessed, and a line holds L_{s} elements, then the probability that at least one of the elements of the line receives a reference is $1 - (1 - p)^{L_{\text{s}}}$. Since position i, $i > 0$, in the area vector represents the ratio of sets that receive $K - i$ lines in the access, its

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

value will be the probability the variable associated to this binomial takes the value $K - i$. The lowest element in the area vector with nonzero probability, m, is the probability the number of lines accessed is $K - m$ or more.

4.2.2 *Access to Groups of Elements Separated by a Constant Stride with Uniform Probability of Access per Element.* Function $\mathrm{Reg}_{rp}(N_\mathrm{r}, T_\mathrm{r}, L_\mathrm{r}, p)$ estimates the cross-interference area vector associated with an access to N_r regions of T_r consecutive elements each and separated by a constant stride of L_r elements, in which each individual element has a probability p of being referenced. The area vector for such region is calculated in two phases:

- In a first phase, the region potentially affected by the references is considered. This region allows to measure the impact of the access on the cache by calculating the number of lines that are mapped to each cache set.
- Since accesses really accur with a given probability p, a second phase is needed where the different combinations of accesses are weighted with the probability that they happen.

A detailed explanation of how both phases are carried out in our model can be found in Andrade et al. [2006].

4.3 Area Vectors Addition

The preceding step generates an area vector per data structure accessed during a reuse distance. Each component of one of these area vectors V yields the probability a given cache set will hold K or more (V_0), or $K - 1$ (V_1), etc., lines because of the accesses to the corresponding data structure that can interfere with the reuses of the reference whose behavior is being analyzed. In this final step of the process these area vectors are added in order to get a global interference area vector that represents the total impact on the cache of all the accesses that take place during the considered reuse distance. The component 0 of this global area vector is the miss probability we are trying to estimate. Given two area vectors V_A and V_B, their addition, represented by the operator $\cup$, is calculated as

$$\begin{aligned}(V_\mathrm{A} \cup V_\mathrm{B})_0 &= \sum_{j=0}^{K} \left(V_{\mathrm{A}_j} \sum_{i=0}^{K-j} V_{\mathrm{B}_i} \right) \\ (V_\mathrm{A} \cup V_\mathrm{B})_i &= \sum_{j=i}^{K} V_{\mathrm{A}_j} V_{\mathrm{B}_{(K+i-j)}} \qquad 0 < i \leq K\end{aligned} \tag{10}$$

This method is based on the addition as independent probabilities of the area ratios, which means that it does not take into account the relative positions of the program data structures in memory. This approach allows our model to provide reasonable estimations in many situations in which the base addresses of the data structures are not known at compile time (e.g., physically addressed caches, dynamically allocated data structures, ...), something that, as far as we know, no other model supports. When those base addresses are known at compile time, each area vector is scaled before its addition by means

of a coefficient that represents the amount of overlapping between the region it represents and the data structure associated to the reference whose PME is being calculated in the cache. [See Fraguela et al. [2003] for more details.]

Example 8. Now that all the steps of the model have been introduced, we are in a position to make a final example that illustrates it quantitatively. Namely, we will compute the number of misses of reference $R =$ `D(I)` during the execution of the code in Figure 2. Our example cache will be a two-way associative cache ($K = 2$) with capacity for a total of 512 array elements ($C_s = 512$), each line being able to hold four elements ($L_s = 4$). The code will operate on a 1000×1000 sparse matrix with $Nnz = 10,000$ nonzeros. Equation (3), developed in example 1 in Section 3.1, estimates the number of misses generated by this reference. If we substitute $M = 1000$ and $L_s = 4$ in this equation we get

$$F_{R0}(\text{RegIn}) = 250 \cdot F_{R1}(\text{RegIn}) + 750 \cdot F_{R1}(\text{Reg}_{R0}(1))$$

We see that out of the 1000 accesses generated by this reference, 250 are first-time accesses to new lines, so their miss probability depends on reuse distances external to this loop, that is, on RegIn. The other 750 accesses reuse the line accessed in the previous iteration, so their miss probability depends on the $\text{Reg}_{R0}(1)$ regions accessed during one iteration of the loop.

Loop 0 is the innermost loop that contains this reference. Thus, the evaluation of F_{R1} for both kinds of accesses consists in computing the global area vector associated to its input region and taking its first component. This component is the miss probability that this region generates on the reuse attempts of R that depend on this evaluation of the PME. Regarding the 250 potential cold misses, their RegIn for F_{R1} is the same as for F_{R0}. Let us remember that RegIn for F_{Ri} is defined as the set of memory regions accessed since the immediately previous access to the lines accessed by R in loop i that could interfere with the potential reuse that can take place when these lines are accessed for the first time during the execution of this loop. As level 0 is the outermost loop of the code, and the code does not include any previous nest, such reuse is impossible. As a result, RegIn for F_{R0} is any memory region such that the first component of its area vector is 1, which implies, that all first-time accesses to lines within the loop result in cold misses. For example, we could use $\text{RegIn} = \text{Reg}_s(C_s)$, a sequential access to as many elements as the cache can hold. The area vector for this region is (1,0,0), so $F_{R1}(\text{RegIn}) = 1$.

As for the 750 potential reuses, the RegIn for their evaluation of F_{R1} is $\text{Reg}_{R0}(1)$, as their attempt of reuse has a reuse distance of one iteration. The expression of this set of regions was developed in example 6 in Section 4.1:

$$\text{Reg}_{R0}(1) = \text{Reg}_s(N_1) \cup \text{Reg}_s(N_1) \cup \text{Reg}_{sp}(D_{X1}, 1 - (1 - 1/D_{X1})^{N_1}) \cup \text{Reg}_{sself}(1)$$

where the four regions correspond to the accesses of references `A(J)`, `C(J)`, `X(C(J))` and `D(I)` itself, respectively, just in the same order they were analyzed in example 6. The average number of iterations of loop 1, N_1 can be estimated as Nnz/M, the average number of nonzeros in a row of the sparse matrix. As for the size of vector `X`, D_{X1}, it is 1000 in our example. Substituting these values in the expressions above, the computation of the corresponding area vectors

```
DO I= 1,M
  DO K= R(I), R(I+1) - 1
    DO J= 1,H
      D(I,J)=D(I,J)+A(K)*B(C(K),J)
    ENDDO
  ENDDO
ENDDO
```

Fig. 5. Sparse matrix–dense matrix product with IKJ order.

yields:

$$\text{Reg}_{\text{s}}(10) = (0, 0.0508, 0.9492) \qquad \text{Reg}_{\text{sself}}(1) = (0, 0, 1)$$
$$\text{Reg}_{\text{sp}}(1000, 0.00996) = (0.0006, 0.0378, 0.9616)$$

Let us remember that in all these vectors the first value is the probability the access pattern puts two or more lines in the cache set in which our reference `D(I)` will try to reuse a line, the second value is the probability they contribute just one line, and the third value is the probability they do not generate accesses to any line that interferes with the reuse. As a result, the addition of the three probabilities must always yield 1. The calculation of the area vector for Reg_{s} and $\text{Reg}_{\text{sself}}$ has been made according to Fraguela et al. [2003]. Anyway it is straightforward that an access to a single element cannot interfere with the attempts to reuse that same line. Thus, it is intuitive that $\text{Reg}_{\text{sself}}(1) = (0, 0, 1)$. As for the area vector calculation for Reg_{sp}, it is reviewed in Section 4.2.1. The addition of the four area vectors applying Eq. (10) yields the global area vector (0.0068, 0.1267, 0.8665). This way, $F_{R1}(\text{Reg}_{R0}(1)) = 0.0068$, that is, the miss probability for each one of the 750 attempts at reuse of this reference is 0.68%.

Altogether, PME F_{R0} estimates that during the execution of this code reference `D(I)` will generate a total of $250 \times 1 = 250$ cold misses and $750 \times 0.0068 = 5.1$ interference misses resulting in a total of 255.1 misses, on average.

5. VALIDATION

Our validation relies on five kernels of increasing complexity that contain indirections derived from the manipulation of sparse matrices stored in the CRS (compressed row storage) format [Barrett et al. 1994]. The first code is the sparse matrix–vector product (SPMXV) shown in Figure 2. The next three codes are the sparse matrix–dense matrix product (SPMXDM) with the three different loop orderings this operation allows: IJK, JIK, and IKJ, where the first index is the one for the outermost loop and the last index the one for the innermost loop in the nest. In the three orderings, I indexes the rows of the sparse matrix, K its columns, and J the columns of the dense matrix. As an example, the IKJ loop ordering is shown in Figure 5. Finally, Figure 6 shows a sparse matrix transposition (TRANSPOSE) where both the original and the transposed matrix are stored using the CRS format. This code is particularly complex, as it contains four loop nests, there are accesses with several levels of indirection in loop 4 and it involves much more data structures than the other examples (six). Besides, some structures appear in several loop nests, so there may be reuses between the access to a line in one loop nest and another access in another loop nest.

```
1  DO I=2,N+1
     RT(I)=0
   END DO

2  DO I=1, R(M+1)-1
     J=C(I)+2
     RT(J)=RT(J)+1
   END DO

   RT(1)=1
   RT(2)=1

3  DO I=3, N+1
     RT(I)=RT(I)+RT(I-1)
   END DO
           ↪

4  DO I=1, M
     DO K=R(I), R(I+1)-1
       J=C(K)
       P=RT(J)
       CT(P)=I
       AT(P)=A(K)
       RT(J)=P+1
     END DO
   END DO
```

Fig. 6. Transposition of a sparse matrix.

Table II. Overall Model Validation Data[a]

Code	$\overline{MR_{\text{Sim}}}$	$\overline{MR_{\text{Mod}}}$	$\overline{\Delta_{MR}}$	$\max(\Delta_{MR})$	$\overline{\Delta_{MR}^{R}}$
SPMXV	9.64%	9.45%	0.92%	3.99%	10.53%
SPMXDMIKJ	48.95%	47.92%	1.41%	11.48%	3.62%
SPMXDMIJK	22.20%	21.42%	0.79%	3.56%	3.41%
SPMXDMJIK	11.68%	11.28%	0.70%	6.65%	8.46%
TRANSPOSE	18.98%	19.22%	1.60%	11.72%	11.61%

[a]Average measured ($\overline{MR_{\text{Sim}}}$) and predicted ($\overline{MR_{\text{Mod}}}$) miss rates, average value $\overline{\Delta_{MR}}$ of the absolute difference between the predicted and the measured miss rate in each experiment, maximum value of this difference $\max(\Delta_{MR})$, and average value of the relative error of the prediction $\overline{\Delta_{MR}^{R}}$, obtained for the benchmarks performing more than 10,000 tests for different cache configurations, data structures sizes, and sparse matrix densities.

5.1 Validation with Synthetic Matrices

The integration of our model [Andrade et al. 2007] in the XARK compiler [Arenaz et al. 2003] has allowed us to apply it automatically to the validation kernels. The miss rate predicted by the model was compared with the results of trace-driven simulations using synthetic matrices with an uniform distribution of their nonzero elements. Over 10,000 tests were performed for each code, changing the sizes and starting addresses of the different arrays, the cache configuration, and the density of the sparse matrix. Table II gives an idea of the accuracy of the model. Columns $\overline{MR_{\text{Sim}}}$ and $\overline{MR_{\text{Mod}}}$ contain the average values of the miss rate simulated and the miss rate predicted in the set of experiments, respectively. Column $\overline{\Delta_{MR}}$ contains the average value of the absolute value Δ_{MR} of the difference between the predicted and the measured miss rates for each experiment. We use absolute values, so that negative errors are not compensated with positive errors. Column $\max(\Delta_{MR})$ contains the largest value of Δ_{MR} observed in the set of experiments. The metric Δ_{MR}^{R} stands for the relative error of our prediction: it is the absolute value of the difference between the miss rate measured by the simulation and the miss rate predicted by the model (Δ_{MR}) divided by the miss rate measured by the simulation and expressed as a percentage, that is, $\Delta_{MR}^{R} = \Delta_{MR}/MR_{\text{Sim}} \times 100$. The last column of the table contains $\overline{\Delta_{MR}^{R}}$, the average value of the Δ_{MR}^{R} observed in each

Table III. Validation Data and Times for the Sparse Matrix–Vector Product Code for Several Cache Configurations, Matrix Sizes and Sparse Matrix Density

M	N	α	C_s	L_s	K	MR_{Sim}	MR_{Mod}	Δ_{MR}	T_{mod}
1000	1000	4.00	8K	32	1	30.11	30.00	0.11	0.015
1500	1100	12.12	32K	32	2	18.17	18.34	0.17	0.021
1600	1500	8.33	32K	64	4	8.44	8.60	0.15	0.010
1300	1400	13.74	64K	128	1	5.21	5.31	0.10	0.012
1700	1500	9.80	64K	64	2	8.67	8.82	0.15	0.032
1100	1000	22.73	128K	128	2	4.21	4.42	0.21	0.021
750	750	7.00	512K	128	8	4.23	5.13	0.90	0.014
5500	5500	0.28	1024K	64	8	8.77	8.86	0.09	0.035
3000	3000	1.19	2048K	128	4	4.26	5.64	1.38	0.033
1000	1200	16.67	128K	128	1	10.82	4.87	5.96	0.025

Table IV. Validation Data and Times for the Sparse Matrix–Dense Matrix Product IKJ Code for Several Cache Configurations, Matrix Sizes and Sparse Matrix Density

M	N	α	H	C_s	L_s	K	MR_{Sim}	MR_{Mod}	Δ_{MR}	T_{mod}
900	900	22.22	500	32K	64	1	89.27	88.27	1.00	0.019
500	500	3.20	600	64K	64	4	81.97	81.61	0.36	0.011
700	700	31.43	500	64K	64	8	29.66	23.30	6.36	0.015
1100	1100	14.55	500	128K	64	8	30.76	29.88	0.88	0.027
1000	1000	15.00	750	128K	64	4	31.18	29.59	1.58	0.038
700	700	27.14	500	256K	64	2	21.25	20.71	0.54	0.019
1000	1000	24.00	500	512K	64	2	23.18	22.45	0.73	0.023
700	700	2.86	500	1024K	32	2	32.89	32.56	0.33	0.027
1000	1000	1.58	1000	2048K	64	4	38.10	36.13	1.97	0.052
600	600	30.00	500	32K	32	8	76.68	65.20	11.48	0.032

one of the experiments. We see that both the average absolute and the relative errors of the model are, in general, very good.

Tables III, IV, and V show some random representative validation results for the sparse matrix–vector product, the sparse matrix–dense matrix product with IKJ loop ordering, and the sparse matrix transposition codes, respectively, displaying a wide range of possible validation parameters and the result obtained.

In the three tables, the first two columns, M and N, show the number of rows and columns of the sparse matrix involved in the code, respectively. Column α is the density or percentage of positions in the sparse matrix with nonzeros. In Table IV, column H shows the number of columns of the dense matrix involved in the product. The cache configuration is given in the three tables by C_s, the cache size in bytes, L_s, the line size in bytes, and K, the degree of associativity of the cache. Larger cache lines and associativities tend to be associated with larger caches, in general, in the tables, as this is the most common situation. For each combination of the input problem parameters and cache configurations, the tables display the miss rate MR_{Sim} measured by the simulations, the miss rate MR_{Mod} predicted by our model, and Δ_{MR}, the absolute value of the difference between them. These three values are expressed as percentages between 0 and 100. The last entry in every table contains the data for the experiment that generated the largest Δ_{MR}.

Table V. Validation Data and Times for the Matrix Transposition Code, for Several Cache Configurations, Matrix Sizes, and Sparse Matrix Density

M	N	α	C_s	L_s	K	MR_{Sim}	MR_{Mod}	Δ_{MR}	T_{mod}
600	600	35.00	16K	32	2	32.49	32.91	0.43	0.029
700	700	34.29	32K	32	1	28.02	26.14	1.89	0.025
3000	2000	2.50	64K	32	2	27.00	29.86	2.86	0.031
5000	2000	3.00	64K	128	1	26.53	27.47	0.93	0.027
1000	1000	15.00	128K	128	1	17.77	19.21	1.44	0.035
800	800	57.50	256K	64	4	5.35	5.17	0.18	0.034
500	500	46.80	512K	128	8	2.18	3.00	0.82	0.037
2900	2900	0.47	1024K	64	1	7.91	10.29	2.38	0.043
500	500	15.75	2048K	64	4	3.08	4.36	1.28	0.042
5000	1000	9.00	128K	64	4	11.50	23.22	11.72	0.023

Table VI. Validation Data and Times for the Sparse Matrix–Vector Product Code for Several Cache Configurations and Different Harwell–Boeing Matrices with Uniform Band Distribution

Matrix Name	Size	B	α	C_s	L_s	K	MR_{Sim}	MR_{Pred}	Δ_{MR}	T_{mod}
jpwh991	991	155	0.61	64K	64	4	9.37	8.84	0.53	0.014
jpwh991	991	155	0.61	32K	32	2	18.77	17.72	1.05	0.013
jpwh991	991	155	0.61	32K	64	1	10.29	9.84	0.45	0.012
bcsstk05	153	20	10.35	32K	64	1	9.57	9.11	0.46	0.009
bcsstk05	153	20	10.35	256K	16	4	35.04	34.13	0.91	0.009
bcsstk05	153	20	10.35	256K	32	2	17.54	17.07	0.48	0.009
bcsstm10	1086	71	1.87	32K	64	1	9.12	9.29	0.17	0.013
bcsstm10	1086	71	1.87	256K	16	4	34.66	33.91	0.74	0.017
bcsstm10	1086	71	1.87	1024K	64	4	8.67	8.48	0.19	0.015
jpwh991	991	155	0.61	8K	16	1	43.79	40.45	3.33	0.013

Finally, the last column in the three tables, T_{mod}, reflects the corresponding modeling times in seconds in a 2.08 GHz AMD K7 processor-based system, respectively. Modeling times, which were always below 1s, are several orders of magnitude shorter than trace-driven simulation for the sparse matrix–dense matrix products, and noticeably shorter, in the case of the other codes.

5.2 Validation with Real Banded Matrices

In order to validate our model for uniform banded matrices we used the sparse matrix–vector product code shown in Figure 2, the sparse matrix–dense matrix product in Figure 5, and the sparse matrix transposition in Figure 6 and we applied them to real matrices from the Harwell–Boeing collection [Duff et al. 1992] rather than to synthetic matrices. The results of some randomly chosen validation experiments are shown in Tables VI and VII for the first two codes considered, respectively. In both tables the first columns contain the name of the matrix used in every test, followed by the characteristics of the matrix, such as, the number of rows and columns size (we used square matrices), the band size B, and, in the case of sparse matrix–dense matrix product code, the number of columns H of the dense matrix. α is the percentage of positions in the sparse matrix with nonzeros. The used cache configuration C_s, L_s, and K, follows. Again, for each experiment, we show both the measured MR_{Sim} and the

Table VII. Validation Data and Times for the Sparse Matrix–Dense Matrix Product IKJ Code for Several Cache Configurations and Different Harwell–Boeing Matrices with Uniform Band Distribution

Matrix Name	Size	B	α	H	C_s	L_s	K	MR_{Sim}	MR_{Pred}	Δ_{MR}	T_{mod}
jpwh991	991	155	0.61	200	32K	64	1	93.08	93.06	0.02	0.011
jpwh991	991	155	0.61	153	16K	32	2	88.61	88.27	0.33	0.010
jpwh991	991	155	0.61	1086	32K	32	4	97.30	98.52	1.21	0.017
jpwh991	991	155	0.61	350	64K	64	4	91.26	92.10	0.83	0.011
bcsstk05	153	20	10.35	153	32K	32	4	16.49	16.84	0.35	0.009
bcsstk05	153	20	10.35	153	16K	32	2	45.34	43.30	2.04	0.009
bcsstm10	1086	71	1.87	153	16K	64	4	74.22	74.32	0.10	0.010
bcsstm10	1086	71	1.87	153	32K	128	1	63.73	62.59	1.13	0.011
bcsstm10	1086	71	1.87	153	512K	64	4	0.70	0.65	0.05	0.014
bcsstm10	1086	71	1.87	200	1024K	64	8	0.68	0.55	0.13	0.058
bcsstk05	153	20	10.35	350	32K	64	1	72.96	61.30	11.67	0.011

predicted MR_{Mod} miss rates and the absolute value of the difference between them, Δ_{MR}. Many different experiments were performed using different cache configurations; the results shown in these tables are only a small representative subset of these tests. The last entry in every table contains, again, the data for the experiment that generated the largest Δ_{MR}.

For the sparse matrix–vector product code, we performed 510 different tests changing the used matrix, the cache configuration, and the base address of the data structures involved in the code, obtaining an average value for the Δ_{MR} of 0.66% and a maximum value of 3.33%; the average value of the relative error Δ^R_{MR} was 3.96%.

For the sparse matrix–dense matrix product code, we performed 5100 different tests, changing the same parameters as for the sparse matrix–vector product code as well as the number H of columns of the dense matrix involved in the code. We obtained an average value for Δ_{MR} of 2.55% and a maximum value of 11.67%. The average value of the relative error Δ^R_{MR} was 6.60%.

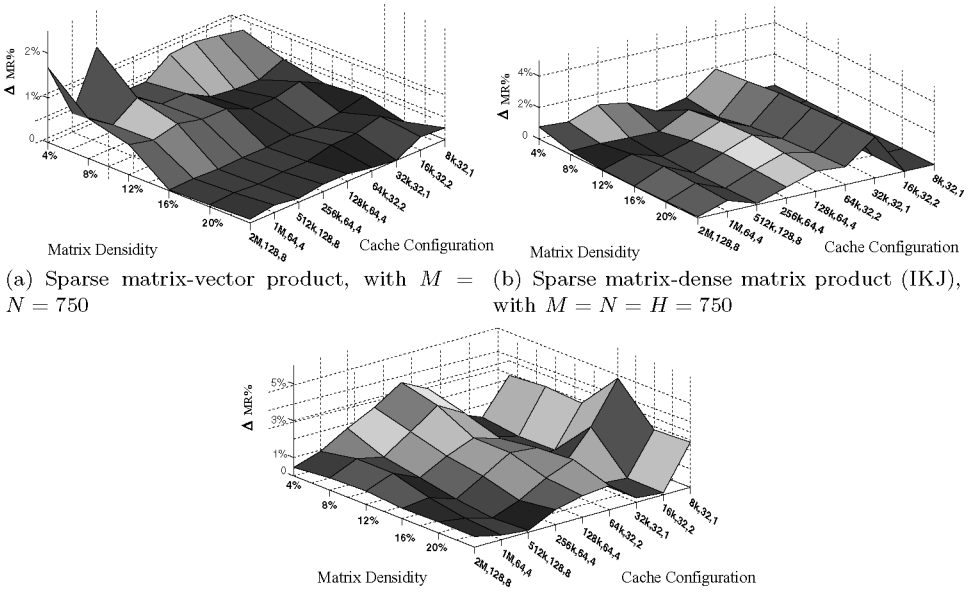
Finally, we performed the same set of 510 tests for sparse matrix transposition as for sparse matrix–vector product. In this case, the average Δ_{MR} was 1.78% and its maximum was 7.30%, being the average value of the relative error Δ^R_{MR} 11.35%.

Again, these validation results obtained using a wide range of parameter combinations, and which are very similar to the ones obtained for the model with a completely uniform distribution displayed in Table II, make us think that our model is a good estimator of the behavior of a code with irregular access patterns under the assumed conditions.

Finally, as in the previous tests, the last column in Tables VI and VII, T_{mod}, represents the time consumed by our model. The model is several orders of magnitude faster than the simulation.

5.3 Discussion

The model worked well for the sparse matrix–vector product of Figure 2 both for matrices, with an uniform distribution of the nonzeros entries, and for real



(a) Sparse matrix-vector product, with $M = N = 750$

(b) Sparse matrix-dense matrix product (IKJ), with $M = N = H = 750$

(c) Sparse matrix transposition, with $M = N = 750$

Fig. 7. Δ_{MR} as a function of the sparse matrix density and the cache configuration in different codes. Cache configurations are expressed as C_s,L_s,K, where C_s is the cache size in bytes, L_s is the line size in bytes, and K is the associativity.

banded matrices. The results were somewhat worse for the sparse matrix–dense matrix product code in Figure 5 for both kinds of matrices, although the model was still very accurate, in general. Predicting the reuse for the reference `B(C(K),J)` that generates irregular accesses in this code is possibly more complex than for the references subject to irregular access patterns in the other codes. The reason is that in this case each value of the indirection controls a whole set of tightly coupled accesses of `B(C(K),J)` to different lines with a regular stride for `J` = 1, ..., `H`, while in the other codes each individual indirection only controls the access to one line. It is good to see that in such a complex situation to predict, the predictions of the model are still good. The behavior of the model for the sparse matrix–dense matrix products in which the inner loop is `K` is similar to the one observed for the sparse matrix–vector product, as we see in Table II. Finally, the transposition of a sparse matrix in Figure 6 turned out to be the most difficult code to predict, as it is not a perfectly nested loop, like the previous examples, and it displays several levels of indirection in its fourth loop. Still, the predictions of the model were very reasonable.

The tendencies of the accuracy of the model with respect to the parameters of the caches and the density of the sparse matrix are displayed in Figure 7, in which we have used cache configurations that are similar or equal to real level 1 and 2 caches of current computers. The most important conclusion is that, in general, higher densities lead to more accurate predictions. That

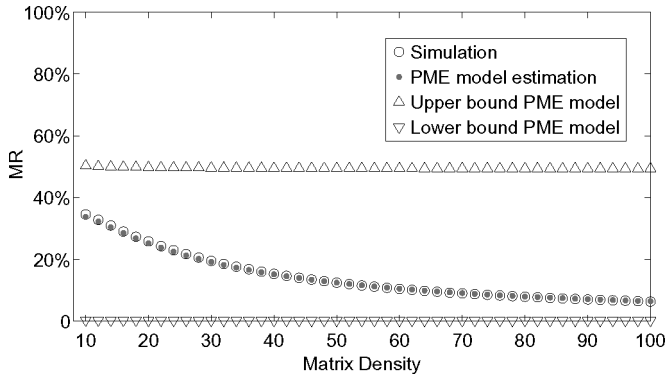


Fig. 8. Miss rate measured and predicted following different strategies as a function of the matrix density for the sparse matrix–dense matrix product (IKJ), where $M = N = H = 500$ in a cache of 64 KB with a line size of 64 bytes and associativity degree 4.

is an expected result, since the lower density, the more irregular the accesses. Also, notice that this higher irregularity leads to higher miss rates (as an example, see experiment in Figure 8), which dilute the larger values of Δ_{MR}.

As for the time required to compute its predictions, the model takes more time when the size of the problem (size of the involved data structures) is bigger, as expected, and when the cache associativity is higher. The reason for the latter behavior is that the complexity of the algorithm for calculating the area vector for some patterns depends directly on this argument. Still, modeling times are always below 1 s. In general, we can say that our model provides quite accurate estimations with a very low computing cost.

The sparse matrix–dense matrix product with IJK loop ordering is used in Figure 8 to compare the miss rate obtained by a trace-driven simulation, the miss rate predicted by the PME model, an upper bound of the prediction obtained by a simplified version of our model that considers all the irregular accesses as misses, and a lower bound obtained by ignoring the irregular accesses that appear in the code. The sizes of the data structures involved in the code and the cache configuration were kept constant while the density of the sparse matrix took values between 1 and 100%. The figure reflects that the PME model estimates the miss rate accurately, while simplified versions provide very poor estimations. This justifies the interest of our model.

Finally, we have also inquired into what happens when the model is applied to matrices with a nonuniform distribution of the entries. In order to quantify this behavior, we run experiments on 320 randomly chosen matrices from the Harwell–Boeing [Duff et al. 1992] and NEP [Bai et al. 1996] collections, using 10 different cache configurations for each one with sizes ranging from 16 KB to 2 MB, thus yielding a total of 3200 experiments per analyzed kernel. Figure 9 summarizes the results of these experiments classifying our experiments in four buckets according to the Δ_{MR} achieved: below 2.5%, between 2.5 and 5%, between 5 and 10%, and larger than 10%. We see that SPMXV, SPMXDM with JIK loop ordering and TRANSPOSE yield reasonable estimations in the vast

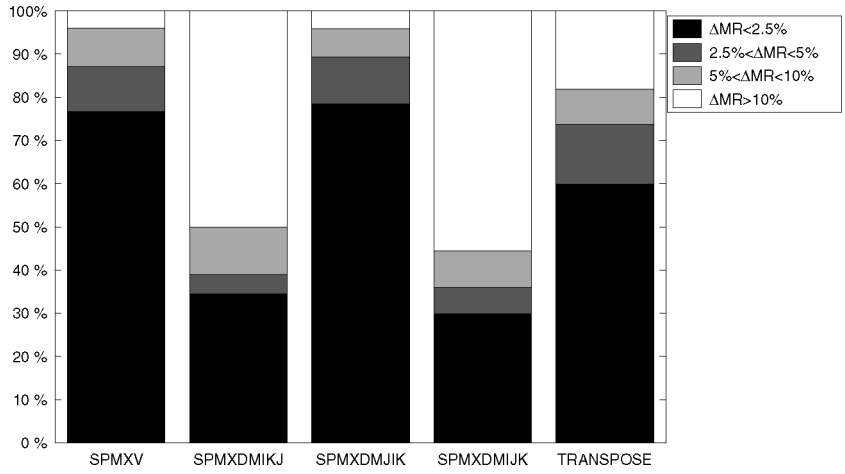


Fig. 9. Percentage of the number of experiments in which the Δ_{MR} is below 2.5%, between 2.5 and 5%, between 5 and 10%, or larger than 10% when real matrices with a nonuniform distribution of the entries are used.

majority of the cases, while SPMXDM with the IKJ and IJK orderings is less reliable. When irregular accesses are not uniformly distributed, they tend to be grouped in clusters, which increases the locality. Thus, in these cases, our model can still help understand the behavior of the cache, but the miss rate it predicts must be considered an upper bound rather than an accurate estimation, unless it is adjusted, as we did for the case of the banded matrices. In the following section, we will see that while the estimations of the model for real matrices with nonuniform distributions of the entries may be sometimes quantitatively inaccurate, its predictions are still valid to drive compiler optimizations in codes that use these real matrices.

5.4 Driving Compiler Optimizations

Analytical models can be used to provide insights about the cache memory behavior of codes and can guide optimizations in a compiler or interactive tool based on their predictions. Namely, decisions can be taken based on a cost function that considers the relative costs of the misses in each memory level as well as the CPU cycles. Memory stall time can be estimated by applying the model to the different levels of the memory hierarchy of the computer simultaneously and multiplying the number of misses estimated for each level by its miss penalty. The cycles spent in the CPU can be estimated using CPU models, such as Delphi [Cascaval 2000], which can apply heuristics to account for the properties of current high ILP superscalars. Several papers in the bibliography illustrate the success of this approach for different optimizations, such as padding [Vera et al. 2005] or tiling [Vera et al. 2003; Fraguela et al. 2005], in codes with regular access patterns.

As a simple experiment aimed to prove that our model can be used to optimize codes with irregular access patterns because of indirections, we used its

Table VIII. Memory Hierarchy Parameters in the Architectures Used (Sizes in Bytes), Miss Weights W in CPU Cycles

Architecture	L1 Parameters ($C_{s_1}, L_{s_1}, K_1, W_1$)	L2 Parameters ($C_{s_2}, L_{s_2}, K_2, W_2$)	L3 Parameters ($C_{s_3}, L_{s_3}, K_3, W_3$)
Itanium 2	(16K,64,4,8)	(256K,128,8,24)	(6MB,128,24,120)
PowerPC 7447A	(32K,32,8,9)	(512K,64,8,150)	—

predictions to decide which was the best loop ordering for the sparse matrix–dense matrix product using the parameters of two very different architectures and memory hierarchies: those of an Itanium 2 at 1.5GHz and a PowerPC 7447A at 1.5GHz. Table VIII shows the configuration of their memory hierarchies using the well-kown notation C_s, L_s, and K, using bytes to measure sizes. A new parameter W, the cost in CPU cycles of a miss in the considered memory hierarchy level, is also taken into account. Notice that the first-level cache of the Itanium 2 does not store floating-point data; so it is only used for the study of the behavior of the references to arrays of integers. Also, the PowerPC does not have a third-level cache.

Our model predicted the same behavior in both architectures for every sparse matrix: the JIK ordering would be the one that would give place to the best performance, while IKJ would be the ordering that would generate more misses in all the levels of the memory hierarchy, thus yielding the worst performance. This matches the global results displayed in Table II. The predictions were first validated executing the three versions of the sparse matrix–dense matrix product code for synthetic sparse matrices with an uniform distribution of the entries of sizes $N \times N$ that were multiplied by a $N \times N$ dense matrix, with $N = i \times 500$ for $i = 1, 2, 3, 4, 5$, and 6, and a percentage of nonzeros in the sparse matrix from 1 to 19% in steps of 2%. The codes were compiled using g77 3.4.3 with level of optimization `-O3`. The execution times systematically reflected the predictions of the model: the JIK version always outperformed the IJK version, and the IKJ code was always the slowest one. We also run a test multiplying each one of the 320 real matrices used in the preceding section by a dense matrix with 1500 columns using the three loop orderings in both machines. In the Itanium 2, the JIK ordering was the best one for 307 of the matrices, IJK for ten, and IKJ for just three of them; in the PowerPC, the JIK ordering was the fastest one in all, but one of the cases, in which IJK outperformed it. As we see, the quantitative inaccuracy that the model exhibits sometimes when predicting the cache behavior for matrices with nonuniform distributions does not preclude it from being successful when driving optimizations on codes that operate with these matrices. Finally, Table IX displays the average execution time for the three loop orderings in the two sets of experiments for both architectures in order to give an idea of the enormous impact of the optimization guided by our model.

6. RELATED WORK

While several automatable analytical models that accurately predict the cache behavior of codes with regular access patterns have been proposed [Chatterjee et al. 2001; Ghosh et al. 1999; Fraguela et al. 2003; Vera and Xue 2002], this is

Table IX. Average Execution Time (s) for the Sparse Matrix–Dense Matrix Product as a Functions of the Loop Ordering for the Experiments Using Synthetic Matrices with a Uniform Distribution of the Entries and Real Matrices with a Nonuniform Distribution of the Entries

	Synthetic Uniform Matrices			Real Nonuniform Matrices		
	Loop ordering			Loop ordering		
Architecture	IKJ	JIK	IJK	IKJ	JIK	IJK
Itanium 2	172.264	41.016	142.389	4.814	2.078	2.115
PowerPC 7447A	338.538	29.256	54.272	12.585	1.990	3.688

not the case for codes with irregular access patterns, because of the difficulty to model them. The frameworks developed hitherto that study the memory hierarchy behavior of these codes either lack a systematic strategy to enable their automatic application within a compiler or cannot provide accurate or absolute performance predictions. This way, Ladner et al. [1999] builds an ad-hoc model restricted to direct-mapped caches that does not provide a general approach to model the interaction in the cache between different interleaved access patterns. Associativity and general access pattern interaction are addressed in our probabilistic model [Fraguela et al. 1998], but it was still a nonautomatable approach. Cascaval's indirect accesses model [Cascaval et al. 2000] is integrated in a compiler framework, but it is a simple heuristic that estimates the number of cache lines accessed rather than the real number of misses. For example, it does not take into account the distribution of the irregular accesses and it does not account for conflict misses, since it assumes a fully associative cache. As a result it suffers from limited accuracy in many situations. The modal model of memory [Mitchell et al. 2001] requires not only static analysis but also runtime experimentation (potentially thousands of experiments) in order to generate performance formulas. Such formulas can guide code transformation decisions by means of relative performance predictions, but they cannot predict code performance in terms of miss rates or execution time. The validation uses two very simple codes and no information is given on how long it takes to generate the corresponding predictions. Accurate miss rates for fully associative caches can be predicted by Zhong et al. [2003] from a reuse distance characterization obtained from two runs of a given code using input data sets of different sizes. The generality of this approach allows to apply it to programs with complex access patterns and, in their experiments, the predictions approach the behavior of limited associativity caches reasonably, although this is not guaranteed. Other limitations of this approach are that the cache line size is not part of the model and it does not take into account cold misses. Finally, Andrade et al. [2006] models irregular access patterns generated by references whose execution is controlled by data-dependent conditionals, but it cannot handle indirections, which is the aim of our work in this paper. Also, that model is less automatable than the one proposed here. The reason is that in the case of the indirections, we can infer the uniform probabilities from the sizes of the data structures. This is in contrast with the data-dependent conditionals, which always require external information to know the probability they are fulfilled.

7. CONCLUSIONS

This paper presents the first automatable approach we are aware of that allows the accurate analytical modeling of the cache behavior of codes with irregular access patterns because of the use of indirections. Our strategy extends the PME model to consider indirections in which the accesses are uniformly distributed on the dereferenced array, or on a subsection of it. When this condition holds, the predictions of our model are very accurate, as our validation with codes of varying complexity shows. When irregular accesses are not uniformly distributed, the predictions of our model are still very reasonable for some codes, while in other codes they must be taken as a qualitative characterization, rather than as a quantitative estimation. We have illustrated the applicability of our work by means of a simple example of optimization guided by the model that has been successful both for synthetic matrices with an uniform distribution of the entries and for real matrices that do not hold this condition. Still, we are currently working on approaches to take into account nonuniform distributions of the entries to provide more accurate estimations. A first step in this direction is our recent paper [Andrade et al. 2007], which analyzes the access patterns derived from the processing of banded matrices.

ACKNOWLEDGMENTS

We want to acknowledge the Centro de Supercomputación de Galicia (CESGA) for the usage of its supercomputers to get the data related to the Itanium 2 architecture.

REFERENCES

Ammons, G., Ball, T., and Larus, J. R. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proc. of the ACM SIGPLAN 1997 Conf. on Programming Language Design and Implementation (PLDI'97)*. ACM Press, New York. 85–96.

Andrade, D., Fraguela, B. B., and Doallo, R. 2007. Cache behaviour modelling for codes involving banded matrices. In *Proc. 19th Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC'06)*. Lecture Notes in Computer Science, vol. 4382. Springer-Verlag, New Orleans, LO. 205–219.

Andrade, D., Fraguela, B. B., and Doallo, R. 2006. Analytical modeling of codes with arbitrary data-dependent conditional structures. *Journal of Systems Architecture 52*, 7 (July), 394–410.

Arenaz, M., Touriño, J., and Doallo, R. 2003. A GSA-based compiler infrastructure to extract parallelism from complex loops. In *Proc. 17th Intl. Conf. on Supercomputing (ICS'03)*. ACM, San Francisco, CA. 193–204.

Bai, Z., Day, D., Demmel, J., and Dongarra, J. 1996. A test matrix collection for non-Hermitian eigenvalue problems, release 1.0.

Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J. M., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Philadalphia: Society for Industrial and Applied Mathematics. Also available as postscript file on http://www.netlib.orgtemplatesTemplates.html.

Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., and Tu, P. 1996. Parallel programming with polaris. *IEEE Computer 29*, 12, 78–82.

Cascaval, G. 2000. Compile-time performance prediction of scientific programs. Ph.D. thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL.

CASCAVAL, C., DEROSE, L., PADUA, D. A., AND REED, D. A. 2000. Compile-time based performance prediction. In *Proc. 12th Int'l. Workshop on Languages and Compilers for Parallel Computing (LCPC'99)*. Lecture Notes in Computer Science, vol. 1863. Springer-Verlag, La Jolla/San Diego, CA. 365–379.

CHATTERJEE, S., PARKER, E., HANLON, P., AND LEBECK, A. 2001. Exact analysis of the cache behavior of nested loops. In *Proc. of the ACM SIGPLAN'01 Conf. on Programming Language Design and Implementation (PLDI'01)*. ACM Press, New York. 286–297.

DUFF, I. S., GRIMES, R. G., AND LEWIS, J. G. 1992. Users' guide for the Harwell-Boeing sparse matrix collection (Release I). Tech. Rep. CERFACS TR-PA-92-96. Oct.

FRAGUELA, B. B., DOALLO, R., AND ZAPATA, E. L. 1998. Modeling set associative caches behavior for irregular computations. *ACM Performance Evaluation Review (Proc. SIGMETRICS/PERFORMANCE'98) 26*, 1 (June), 192–201.

FRAGUELA, B. B., DOALLO, R., AND ZAPATA, E. L. 2003. Probabilistic miss equations: Evaluating memory hierarchy performance. *IEEE Transactions on Computers 52*, 3 (Mar.), 321–336.

FRAGUELA, B. B., CARMUEJA, M. G., AND ANDRADE, D. 2005. Optimal tile size selection guided by analytical models. In *Procs. of Parallel Computing 2005 (ParCo 2005)*. Malaga, Spain. 565–572.

GHOSH, S., MARTONOSI, M., AND MALIK, S. 1999. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Transactions on Programming Languages and Systems 21*, 4 (July), 702–745.

LADNER, R. E., FIX, J. D., AND LAMARCA, A. 1999. Cache performance analysis of traversals and random accesses. In *SODA '99: Proc. of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA. 613–622.

LAM, M. S., ROTHBERG, E. E., AND WOLF, M. E. 1991. The Cache performance and optimizations of blocked algorithms. In *Proc. of the 4th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society, Santa Clara, CA. 63–74.

MITCHELL, N., CARTER, L., AND FERRANTE, J. 2001. A modal model of memory. In *ICCS '01: Proc. of the Int'l. Conf. on Computational Sciences-Part I*. Lecture Notes in Computer Science, vol. 2073. Springer-Verlag, New York. 81–96.

TEMAM, O., FRICKER, C., AND JALBY, W. 1994. Cache interference phenomena. In *Proc. Sigmetrics Conf. on Measurement and Modeling of Computer Systems*. ACM Press, New York. 261–271.

UHLIG, R. AND MUDGE, T. 1997. Trace-Driven Memory Simulation: A Survey. *ACM Computing Surveys 29*, 2 (June), 128–170.

VERA, X. AND XUE, J. 2002. Let's study whole-program behaviour analytically. In *Proc. of the 8th Int'l Symposium on High-Performance Computer Architecture (HPCA 8)*. IEEE, Cambridge, MA. 175–186.

VERA, X., ABELLA, J., GONZALEZ, A., AND LLOSA, J. 2003. Optimizing program locality through CMEs and GAs. In *Proc. 12th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'03)*. New Orleans, LA. 68–78.

VERA, X., LLOSA, J., AND GONZALEZ, A. 2005. Near-optimal padding for removing conflict misses. In *Proc. Languages and Compilers for Parallel Computers (LCPC02)*. Lecture Note in Computer Science, vol. 2481. LNCS—Springer Verlag, College Park, MD. 329–343.

ZHONG, Y., DROPSHO, S. G., AND DING, C. 2003. Miss rate prediction across all program inputs. In *Proc. of the 12th Intl Conf. on Parallel Architectures and Compilation Techniques (PACT'03)*. 79–90.

Received March 2006; revised July 2006; accepted November 2006