SYSARC 639
26 October 2005 Disk Used
ARTICLE IN PRESS
No. of Pages 17, Model 3+

ELSEVIER

Available online at www.sciencedirect.com
SCIENCE DIRECT®

Journal of Systems Architecture xxx (2005) xxx–xxx

JOURNAL OF SYSTEMS ARCHITECTURE

www.elsevier.com/locate/sysarc

Analytical modeling of codes with arbitrary data-dependent conditional structures ☆

D. Andrade, B.B. Fraguela *, R. Doallo

Universidade da Coruña, Depto. de Electrónica e Sistemas, Facultade de Informática, Campus de Elviña s/n, 15071 A Coruña, Spain

Received 20 May 2004; received in revised form 28 January 2005; accepted 24 April 2005

Abstract

Several analytical models that predict the memory hierarchy behavior of codes with regular access patterns have been developed. These models help understand this behavior and they can be used successfully to guide compilers in the application of locality-related optimizations requiring small computing times. Still, these models suffer from many limitations. The most important of them is their restricted scope of applicability, since real codes exhibit many access patterns they cannot model. The most common source of such kind of accesses is the presence of irregular access patterns because of the presence of either data-dependent conditionals or indirections in the code. This paper extends the probabilistic miss equations (PME) model to be able to cope with codes that include data-dependent conditional structures too. This approach is systematic enough to enable the automatic implementation of the extended model in a compiler framework. Validations show a good degree of accuracy in the predictions despite the irregularity of the access patterns. This opens the possibility of using our model to guide compiler optimizations for this kind of codes.

Keywords: Memory hierarchy; Cache behavior; Performance prediction; Irregular access patterns

1. Introduction

There has been a growing interest in the study and understanding of the behavior of the memory hierarchies in the past years. The reason is the essential role they play in the performance of modern computers, mainly because of the increasing difference between main memory and processor speeds. One of the most effective ways to reduce the impact of this difference is the usage of memory hierarchies with one or, more typically, several level of caches.

The first approach to study the behavior of these systems was the usage of trace-driven simulations [1]. This approach, while very accurate, has many drawbacks: difficulty to store the traces, large computing times, and lack of an explanation for the behavior observed in many cases. The first two problems can be overcome by the usage of hardware counters [2], but they still offer no explanations about the behavior observed and they are restricted

☆ This work was supported in part by the Ministry of Education and Science of Spain under contracts TIC2001-3694-C02-02 and TIN2004-07797-C02-02, and by the Xunta de Galicia under contract PGIDIT03TIC10502PR.

* Corresponding author. Tel.: +34 981 167000x1219; fax: +34 981 167160.

E-mail addresses: dcanosa@udc.es (D. Andrade), basilio@udc.es (B.B. Fraguela), doallo@udc.es (R. Doallo).

1383-7621/$ - see front matter
doi:10.1016/j.sysarc.2005.04.004

to the platforms in which they are available. Besides, none of those approaches is suitable to guide the optimization process of a compiler. This way, a number of analytical models have appeared that try to address these issues [3–7].

Analytical models suffer typically from two kinds of problems: a certain lack of accuracy and a limited scope of applicability, either because of the limited number of code structures that can model or because of a restriction to model a given kind of hardware. Some of the most recent models have achieved very good degrees of accuracy in their predictions, and they are general enough to consider both the direct-mapped and set-associative caches with LRU replacement that are found nowadays in almost every computer. Still, they continue to restrict their applicability to codes that must exhibit regular access patterns. Unfortunately, most real codes comprise either indirections or portions of code whose execution depends on conditions computed at run-time. These structures break the regularity of the accesses and, as a result, they are beyond the scope of these models.

In this paper we extend one of these models, the probabilistic miss equations (PME) model [7], to enable it to analyze automatically codes that include data-dependent conditional structures. We will consider codes with any kind and number of conditional sentences, even with references controlled by several nested conditionals, and nested in any arbitrary way. Only two restrictions are set on the conditions. The first one is that their verification must follow an uniform distribution, although each condition may have a different probability of being fulfilled. The second one is that the conditions must be independent, this is, the probability a given condition is fulfilled is not influenced by the fact any other condition(s) is/are fulfilled or not. These restrictions ease the mathematical treatment of the problem in this first attempt to model automatically codes with irregular access patterns, while allowing to represent the most important modeling problems derived from such irregularities. Still, we acknowledge these conditions do not hold in most real codes. This way, we are currently working in the modeling of conditions that are fulfilled with non-uniform distributions.

The PME model, which we describe in detail in Section 2, builds a separate expression for each reference and each loop that encloses it, that estimates the number of misses generated by the reference during the execution of that loop. Its equations are probabilistic because the number of misses is estimated as the product of the estimated number of accesses by the estimated probability each one of those accesses generates a miss. Such probability is derived from the footprint on the cache of the different regions accessed between two consecutive accesses to the same line by the reference that is being analyzed. This way, the original PME model in [7] only used probabilities to describe the probability an access resulted in a miss, while the number of accesses and the shape of the footprints was fixed. Our extension also uses probabilities to estimate the number of accesses, and to estimate the footprint of the regions that can preclude a reuse in an access. The reason is that references affected by data-dependent conditionals only take place with a given probability. As a result, a new strategy to generate probabilistic miss equations has been developed to deal with these codes.

Notice that the PME model provides more information than other analytical models of the memory because it generates an individual equation for each reference and nesting level, and the miss probabilities are computed adding the contributions of the accesses of the different references found within the reuse distance. This way, a very detailed individual analysis for every reference and how it influences the behavior of other references is provided.

This paper is structured as follows: The following section provides an introduction to the PME model extensively described in [7]. Then, Section 3 describes the scope of application of the new extension and its formulation. Section 4 is devoted to the validation of the extended model. A brief review of the related work is presented in Section 5, followed by our conclusions and a discussion on the future work in Section 6.

2. Probabilistic miss equations (PME) model

As mentioned in the previous section, the PME model is originally oriented to the modeling of codes with regular access patterns. The model considers caches of an arbitrary size, line size and associativity whose replacement policy is LRU. It supports both perfectly and imperfectly nested loops with a fixed number of iterations. The model allows several references per data structure and loop, and it requires the indexing functions for the different dimensions of the references to be affine functions of the enclosing loops index variables, which is the most common situation. The model can also take into account the probability of hit due to the reuse of cache lines in different loop nests, which enables it to model complete codes. Still, the inter-nest reuse modeling accuracy is subject to the fulfillment of certain conditions.

The estimation of the number of misses generated by the execution of a given code in a certain cache is made separately for each reference in this model. In fact, the model generates a separate equation for each loop and for each reference that estimates the number of misses it generates in that loop. This is modular and it allows the user to know which are the hot spots and references in the code. The model classifies misses in two categories. Compulsory misses are those that take place the very first time a line is referenced in the code. Interference misses are attempts to reuse a line that fail because the line was evicted from the cache since its previous access. The distinction is reflected in the way the PMEs are built, as each kind of misses is estimated separately. The references that can give place to a reuse are also classified in their turn according to their reuse distance, this is, the portion of code executed since the latest access to the line they try to reuse. The reason is that different reuse distances have associated a different probability of resulting in a miss. The number and type of the different accesses is estimated from the indexing functions of the references and the sizes of the loops.

The probabilistic nature of the PME model comes into play when the interference misses are estimated. They are calculated separately for each potential reuse distance, as the product of the number of accesses that could enjoy a potential reuse of a line in the cache with that distance, by the probability each access really results in a miss. The probability is estimated from the cache footprint of those regions that have been accessed since the latest reference to the line, this is, during the considered reuse distance.

We will now describe the strategy to represent these footprints and estimate the corresponding miss probabilities and how PMEs are built for references that are not subject to conditional accesses, this is, those considered in [7].

2.1. Miss probability calculation

The PME model measures reuse distances in terms of loop iterations. Fig. 1 shows the steps the PME model follows to derive the miss probability associated to a given reuse distance. We will now comment them in turn.

2.1.1. Access pattern identification

In the first step, the access pattern followed by the references involved in a reuse distance is extracted from their indexing functions and the shape of the loops that enclose them. This task is eased due to the usage of affine indexing functions in the references considered by the model. The access patterns can be described by means of the memory regions they reference, using for example notations like the *Access Region Descriptors* [8]. Nevertheless, the PME model represents access patterns as functions whose output is the footprint of the access on the cache. The model associates a different function to each typical class of access pattern found in the codes analyzed (sequential access, access to regions separated by a constant stride, etc.). The function arguments complete the description of the access pattern. For example, the only argument required to characterize a sequential access is the number of words accessed.

2.1.2. Cache impact quantification

The second step evaluates the access pattern functions to obtain their associated cache footprints. These footprints are represented in the PME model by what we call *area vectors*. An area vector V consists of $K+1$ probabilities $V_0 V_1 \ldots V_K$, where K is the degree of associativity of the cache whose behavior is analyzed. This representation is designed to be very convenient for the calculation of the impact of the corresponding accesses on the miss probability when trying to reuse lines from

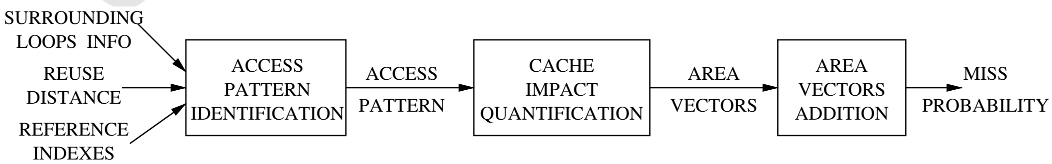

Fig. 1. Procedure for estimating miss probabilities from the code.

other access patterns or even from the access that is being considered. In fact, two kinds of area vectors are distinguished:

- *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, the first component, V_0, is the probability that a set in the cache has received K or more lines accessed by the pattern. One can think of this probability also as the ratio of cache sets that have received K or more lines during the access. Then, V_1 is the probability a cache set has received exactly $K-1$ lines; V_2 is the probability a cache set has received exactly $K-2$ lines; and so on. In general, except for $i=0$, V_i is the probability a given set has received exactly $K-i$ lines due to the access.
- *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. Let us consider a simple cache footprint in a 2-way associative cache with eight sets such that 7 of the 8 sets have received two lines, and the other set has only received one line. The cross-interference area vector V_{cross} for this footprint is $(7/8, 1/8, 0)$, since 7 out of the 8 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 V_{self} for this footprint is $(0, 14/15, 1/15)$. Its first component indicates that none of the lines involved in the access has to compete for its cache set with other two or more other lines from the access pattern. The second component is the ratio of lines of the footprint that share their cache set with exactly one line (14 out of 15). Finally, as the third component points out, only one of the fifteen 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.

Area vectors are derived for each access pattern either analytically or by simulation or following a hybrid approach. The method to estimate the area vectors associated to the most commonly found access patterns has been described in [7]. Section 3.1 describes the estimation of the area vector for two new access patterns not previously considered.

2.1.3. Area vectors addition

Interference probabilities are directly obtained from area vectors because in a K-way associative cache, the probability of missing in the cache when trying to reuse a line corresponds to the probability that K (or more) different lines, mapped to the cache set associated with that line, have been referenced since its previous access. This is exactly the first component of any area vector. The other components are also required because several data structures may be accessed during a given reuse distance. The PME model estimates the area vector for the accesses to each structure separately and then adds them to calculate the global area vector in the third step of the process depicted in Fig. 1, the *Area Vectors Addition*. This way, components not in the first position of their corresponding area vectors may be combined to increase the probability that in the global footprint there are K or more lines mapped to a cache set. The addition of area vectors, whose operand is $\cup$, is described in detail in [7].

2.2. Condition independent PMEs

The PME model numbers the loops in a nest from the outermost one, zero, to the innermost one, Z; and it analyzes the behavior of the references beginning in the innermost loop that contains them and proceeding outwards. This way, the model generates an estimator $F_i(R, \text{RegIn})$ of the number of misses generated by each reference R during the execution of each enclosing loop at nesting level i. This PME depends on RegIn, the footprints generated by regions accessed in outer loops that may interfere with the reuse of the footprint of R in loop i.

Every estimator is a summatory. The first term corresponds to the accesses that cannot enjoy reuse in the considered loop, so it is associated to the misses that are compulsory from the point of view of the loop. The miss probability for these accesses depends on RegIn, the footprint due to accesses in outer loops. The remaining terms correspond to the accesses that can enjoy reuse, there being one term for each different potential reuse distance. Every term is a product of the estimated number of accesses that reuse cache lines with a given reuse distance multiplied by the miss probability associated to that distance.

A description on how to derive PMEs both for references that can and cannot reuse lines accessed by other references is found in [7]. In order to make this paper more self-contained and help understand our extension in Section 3, we will explain here the construction of PMEs for references that carry no reuse with other references in its loop nest. These PMEs are built as

$$F_i(R, \mathrm{RegIn}) = L_{Ri}F_{i+1}(R, \mathrm{RegIn}) + (N_i - L_{Ri})F_{i+1}(R, \mathrm{Reg}_i(R, 1)), \quad (1)$$

where N_i is the number of iterations of the loop at nesting level i, L_{Ri} is the number of iterations in which R cannot reuse lines in this loop, $F_{i+1}(R,$ RegIn) is the PME for the same reference R in the immediately inner loop and $\mathrm{Reg}_i(R, n)$ are the regions accessed during n iterations of the loop i that may interfere with the accesses of R. The formula reflects that the miss probability for the L_{Ri} loop iterations in which there can be no reuse in this loop, depends on the accesses in the outer loops (given by RegIn), while the miss probability for the accesses in the remaining iterations is a function of the regions accessed during the portion of the program executed between those reuses, which is one iteration of this loop. Notice how the calculation for the PME in level i provides the RegIn argument for F_{i+1}, which estimates the behavior of R during the execution of the immediate inner loop.

Two special cases must be considered when evaluating the PMEs:

- In the innermost loop $F_{i+1}(R, \mathrm{RegIn}) = AV_0(\mathrm{RegIn})$, this is, the first element of the area vector associated to the region RegIn. The reason is that the estimator is associated here to a single access in a single iteration of this innermost loop.
- When the outermost loop is reached, the input region for $F_0(R, \mathrm{RegIn})$, which estimates the total number of misses generated by R in the nest, is $\mathrm{RegIn}_{\mathrm{total}}$, an imaginary region that covers the whole cache and that generates a miss probability one. The reason is that the PMEs propagate this region as RegIn for those accesses that carry no reuse at all in the nest and which, as a result, are compulsory misses for the nest.

Since the indices of the references are affine functions of the enclosing loop variables, the accesses of every reference R have a constant stride S_{Ri} associated to the loop i. Consequently, the number of different lines that are accessed in N_i iterations with stride S_{Ri}, can be calculated as

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

where L_s is the number of array elements a cache line holds. This L_{Ri} value corresponds also to the number of iterations in which the accesses of R cannot reuse lines brought to the cache by previous accesses in this loop. The remaining $N_i - L_{Ri}$ iterations can exploit either spatial or temporal locality, with a reuse distance of a single iteration of the considered loop.

3. Modeling of condition dependent references

The modeling strategy described in the preceding section is valid for codes without conditional sentences, which is the scope of application of all the previous works in the bibliography, as we will see in Section 5. Only Vera and Xue [9] has considered codes with conditional sentences, but it is restricted to conditions on the loop indices, which are completely predictable and analyzable off-line and which tend to follow quite regular patterns. In practice, many codes include data-dependent conditionals whose outcome depends on computations made at run-time, and where the pattern of the condition is highly irregular. As a result, the references affected by those conditions exhibit very irregular access patterns that no model has managed to analyze following a modular and systematic approach. This is the main contribution of our work.

The scope of application of our model is shown in Fig. 2. We now consider any number of arbitrarily nested conditional statements, with an arbitrary number of atomic conditions that involve any number of data elements. The figure only shows one data element per condition for simplicity. The `IF` structures condition the execution of isolated references or complete loops or nests. The restrictions in the PME model of constant number of loop iterations and affine indexing continue to hold. Also, our current systematic strategy to model irregular access patterns requires the conditions in the code to follow an uniform distribution and to be independent. This latter restriction means that the

```
DO I0=1, N0, L0
  DO I1=1, N1, L1
  ...
  IF cond(D(fD1(ID1), ..., fDdD(IDdD)))
  ...
    DO IZ=1, NZ, LZ
      A(fA1(IA1), ..., fAdA(IAdA))
      ...
      IF cond(B(fB1(IB1), ..., fBdB(IBdB)))
        C(fC1(IC1), ..., fCdC(ICdC))
      ...
    END DO
  ...
  END DO
END DO
```

Fig. 2. Loop nest with data-dependent conditions.

probability that a given condition is fulfilled or not does not depend on the verification of other conditions in the code. We expect to relax these restrictions in future works. The different conditions may be fulfilled with different probabilities each.

Two kinds of extensions are required to consider irregular accesses. One is the identification of new access patterns that give place to footprints not considered by the original PME model, and for which methods must be developed in order to estimate their corresponding area vectors. The other one is the consideration of a new kind of PMEs in which reuses take place only with a given probability, and whose reuse distance varies depending on the behavior of the conditional sentences found in the nest. We will now consider in turn these two issues.

3.1. Irregular access patterns

The two access patterns usually found in codes with regular access patterns are the sequential access and the access to groups of consecutive elements of the same size that are separated by a constant stride. Their irregular counterparts, when uniform probabilities of access are considered, are described in a similar way, with the important difference that now each one of the elements involved in the pattern is accessed with a given probability p that is the same one for every element. The modeling of these new access patterns, which we detail below, depends on the cache parameters. A cache is defined by its total size C_s, its line size L_s, and its associativity K. For simplicity, both C_s and L_s are measured in elements or words of the access we are considering. Two derived parameters that help simplify some expressions are the number of sets in the cache, $N_K = C_s/(KL_s)$, and $C_{sk} = C_s/K$, the cache size devoted to each level of associativity.

3.1.1. Sequential access with uniform probability

We denote as $S_{sp}(n,p)$ the cross-interference area vector associated to an access to n consecutive elements in which each one of them has a probability p of being referenced. The $K+1$ elements of this vector are calculated as

$$S_{sp_i}(n,p) = P(X = K - i) \quad m < i \leqslant K,$$
$$S_{sp_m}(n,p) = P(X \geqslant K - m),$$
$$S_{sp_i}(n,p) = 0 \quad 0 \leqslant i < m,$$

where $X \in B(n/C_{sk}, 1-(1-p)^{L_s})$, being $B(n,p)$ the binomial distribution[1] and $m = \max\{0, K - \lceil n/C_{sk} \rceil\}$. 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 $\lceil n/C_{sk} \rceil$, so the area vector elements $S_{sp_i}(n,p)$ for $0 \leqslant i < m$ 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 $B(n/C_{sk}, 1-(1-p)^{L_s})$. The probability of access per line of this binomial is easy to calculate, as since each individual element in a cache line has a probability p of begin 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 $1-(1-p)^{L_s}$. Since position i, $i > 0$, in the area vector represents the ratio of sets that receive $K - i$ lines in the access, its value will be the probability the variable associated to this binomial takes the value $K - i$. The lowest element in the area vector with non-zero probability, m, is the probability the number of lines accessed is $K - m$ or more.

3.1.2. Access to groups of elements separated by a constant stride with uniform probability

We denote as $S_{rp}(N_r, T_r, L_r, p)$ the cross-interference area vector associated to 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

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

referenced. This area vector 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 happen with a given probability p, a second phase is needed where the different combinations of accesses are weighted with the probability that they happen.

3.1.2.1. Calculation of the code footprint. We first define the helper function $pos(i) = i \bmod C_{sk}$, which calculates which position in the cache corresponds to an arbitrary memory position i.

In a first step, the first position C_i of every region i that compounds the pattern mapped on a cache of size C_{sk}, is calculated as

$$C_1 = 0,$$
$$C_i = pos(C_{i-1} + L_r), \quad 1 < i \leqslant N_r.$$

In the following, $CV(i)$ will stand for the number of regions that begin in the position i of the cache. Now we calculate for every cache set, $1 \leqslant j \leqslant N_K$, the number of different lines mapped to the considered cache set j in which exactly i of their elements may be referenced by this access pattern. This is the set of values $N(j,i)$, where $1 \leqslant i \leqslant L_s$.

The value of $N(j,i)$ for $i < \min(T_r, L_s)$ is calculated as

$$N(j,i) = CV(pos(jL_s - T_r + i)) + CV(pos(jL_s + L_s - i))$$

since only the regions that begin exactly $T_r - i$ positions before the beginning of the considered set or in the ith position of the set can contribute with a line where only i of its elements may be referenced by the access pattern.

The calculation of the remaining $N(j,i)$ depends on whether $T_r < L_s$. If this is the case, then

$$N(j, T_r) = \sum_{t=0}^{L_s - T_r} CV(pos(jL_s + t)),$$
$$N(j,i) = 0, \quad T_r < i \leqslant L_s$$

since the regions beginning in the first $L_s - T_r + 1$ positions of the set will have one line in which T_r of its elements may be accessed, and given that $T_r < L_s$, it is impossible that there are regions with lines where more than T_r elements may be accessed.

Finally, if $T_r \geqslant L_s$, all the $N(j,i)$ but $N(j,L_s)$ have been calculated. The value for the latter is calculated as

$$N(j, L_s) = \sum_{t=L_s}^{T_r} CV(pos(jL_s - T_r + t))$$

because any region that begins either in the first position of the set or in the $T_r - L_s - 1$ immediately preceding positions will have one line mapped to the considered set j in which all of its elements may be affected by the access pattern.

3.1.2.2. Weighting the accesses probabilities. In the previous phase we have estimated the footprint of this access pattern without taking into account the probability that each element in the footprint is really referenced. Let us remember that the footprint is represented by the values $N(j,i)$, which are the number of lines mapped to set j that contain i words affected by the access pattern. Since the access to each element happens only with probability p, this is an upper bound of the real number of lines that are accessed. This way, the purpose of this phase is to estimate how many lines are really accessed taking into account that the probability of access to each element in the region is p.

Our strategy to estimate the total area vector for this access pattern is to calculate the area vector for each set j independently and to average them. The area vector for each single set j, S_j, represents the distribution of probability that the access generated references to l different lines mapped to this set for $0 \leqslant l < K$ in the positions $S_{j(K-l)}$ of the vector, or to K or more different lines, in the position S_{j0}. This distribution of probability is calculated from L_s binomial variables, X_{ji}, $1 \leqslant i \leqslant L_s$, where X_{ji} is the number of lines that are really accessed out of the $N(j,i)$ ones that are mapped to set j and which contain exactly i positions that can be referenced by the access pattern analyzed. This way, $X_{ji} \in B(N(j,i), 1-(1-p)^i)$, where $B(n,p)$ stands for the binomial distribution. The probability of the binomial is given by the fact that if in a given line only i positions may be subject to access, and the access to each position only happens with probability p, then the probability the line has really been accessed is $1-(1-p)^i$. As a result, if we define $X_j = \sum_{i=1}^{L_s} X_{ji}$, then the area vector for the set j can be estimated as $S_{j(K-l)} = P(X_j = l)$, $0 \leqslant l < K$ and $S_{j0} = P(X_j \geqslant K)$.

3.2. Condition dependent PMEs

In order to consider the probabilities that the different conditional statements that may affect a given reference R in its nest hold, we extend the PME that estimates the behavior of a reference R in a loop i with a new argument $\vec{p}$. This vector contains in position j the probability p_j that the (possible) conditionals that guard the execution of the reference R in nesting level j are verified. If a given loop contains no conditional structures, then $p_j = 1$, which means the execution in this level is unconditional. When there are several nested `IF` statements in the same nesting level, p_j is the product of the probabilities of holding their respective conditions.

We have found that $F_i(R, \mathrm{RegIn}, \vec{p})$ may take two different forms when considering codes with data-dependent conditional statements. If the reference is not affected by any conditional sentence or if the variable that indexes loop i does not index any of the references found in the condition(s) of the conditional(s) sentence(s) that affect the execution of R, then the PME takes the form described in Section 2.2. This kind of PME disregards its input $\vec{p}$, which is not used in the computations. But if this is not the case, this is, if the variable of the loop is used in the indexing of a data array involved in a conditional that controls the execution of the reference R that is being studied, then a new kind of PME must be used. From now on we will distinguish both kinds of PMEs by calling the former ones *Condition Independent PMEs* and these new ones *Condition Dependent PMEs*.

Just as we did in Section 2.2, we will now describe the construction of Condition Dependent PMEs for references that carry no reuse with other references. We will do it in two steps. First, we will develop the general form of a Condition Dependent PME. This PME is based on the probability that the reference that is being analyzed actually accesses each one of the lines of the set that the reference can potentially access during one iteration of the loop i we are considering. In a second step, an algorithm to derive this probability will be presented.

3.2.1. General form of a condition dependent PME

A PME must be built for each loop i enclosing a reference R. The PME is basically a summatory where each term is the product of the number of accesses that have a given reuse distance, multiplied by the PME for the lower level when the input footprint corresponds to that reuse distance. When reference R is affected by data-dependent conditionals, this is, when one or more `IF` structures that depend on data control the reference, the reuse distances are not fixed. Depending on the pattern of verification of the conditions that control the execution of the reference, its accesses may try to reuse lines with very different distances. These reuse distances will have different probabilities of happening, depending on the distribution of probability of the verification of the conditionals that control the execution of the reference. This way, the PMEs for this kind of references will use probabilities not only to represent the miss probability for a given reuse distance, as those in Section 2.2 did, but also to estimate how many accesses take place with each possible reuse distance. Notice that PMEs measure the reuse distance in terms of iterations of the loop they are associated to, and the unit of reuse in a cache is the line. As a result, the base probability to weight the different reuse distances must be the probability that the reference that is being analyzed accesses one of the lines it may potentially access during each iteration of the loop i that is being considered. In general, when the conditionals do not follow an uniform distribution, a set of different probabilities for different iterations and/or lines must be used. As the scope of this analysis is restricted to conditionals that follow an uniform distribution, in this work this probability is a single parameter, $P_i(R, \vec{p})$, that has the same value for every iteration of the loop i and for every line that R may access. This way, the condition dependent PME for loop i and reference R has the form

$$F_i(R, \mathrm{RegIn}, \vec{p}) = p_i L_{Ri} \sum_{j=1}^{G_{Ri}} \mathrm{WMR}_i(R, \mathrm{RegIn}, j, \vec{p}), \tag{3}$$

where L_{Ri} is the number of iterations in which new different lines would be accessed by reference R due to the stride in loop i if it were not subject to conditional execution, and p_i is the probability the conditional sentences that control the execution of R in this loop level are true. The product of these two terms gives the average number of iterations in which R accesses different lines due to its stride for this loop. This number of iterations must be multiplied by the PME for the immediately lower level evaluated with the appropriate reuse distance area vector, which is what the term WMR_i stands for, a weighted number of misses for a reference in level i. As stated before, because of the control by data-dependent conditionals, a range of different reuse distances with different probabilities may take place. This range has an average upper bound G_{Ri}, the number of iterations that can potentially reuse the lines accessed in the L_{Ri} iterations that give place to accesses to new lines. The product of both terms must be equal to the number of iterations of the loop, thus $G_{Ri} = N_i/L_{Ri}$.

Let us now develop the value of $\text{WMR}_i(R, \text{RegIn}, j, \vec{p})$, the weighted number of misses generated by reference R in loop i when RegIn is the region accessed since the last access to any of the lines affected by the reference of R before loop i begins its execution, and the line is used in the jth possible iteration in which the line could be accessed. This function is computed as

$$\begin{aligned}\text{WMR}_i(\text{RegIn}, j, \vec{p}) &= \overline{P_i(R,\vec{p})}^{\,j-1} F_{i+1}(R, \text{RegIn} \cup \text{Reg}_i(R, j-1), \vec{p}) \\ &\quad + \sum_{k=1}^{j-1} P_i(R,\vec{p})\overline{P_i(R,\vec{p})}^{\,k-1} F_{i+1}(R, \text{Reg}_i(R,k), \vec{p}),\end{aligned} \tag{4}$$

where $P_i(R,\vec{p})$, the probability that R accesses during one iteration of loop i one of the lines that belong to its potential access pattern, is used to weight the probabilities that the different reuse distances take place. In this equation $\overline{p}$ stands for $1-p$, this is, the converse probability of p. Let us remember that $\text{Reg}_i(R,n)$ stands for the regions accessed during n iterations of the loop i that may interfere with the accesses of R. The first term in Eq. (4) considers the case that the line has not been accessed during any of the previous $j-1$ iterations. 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 regions accessed during these $j-1$ previous iterations of the loop in order to estimate the complete interference region. The second term weights the probability that the last access took place in each of the $j-1$ previous iterations of the considered loop.

3.2.2. Line access probability

The probability $P_i(R,\vec{p})$ that reference R accesses one of the lines that belong to the region that it can potentially access during one iteration of loop i is a basic parameter to derive $F_i(R, \text{RegIn}, \vec{p})$, as we have just seen. This probability depends not only on the access pattern of the reference in this nesting level, but also in the inner ones, so its calculation takes into account all the loops from the ith down to the one containing the reference. If fact, this probability is calculated recursively in the following way:

- If i is the innermost loop containing R, then

$$P_i(R,\vec{p}) = \begin{cases} 1 & \text{if the accesses of } R \text{ are consecutive with respect to loop } i, \\ p_i & \text{otherwise,} \end{cases}$$

where a consecutive access with respect to a given loop implies that the accesses that take place in consecutive iterations of the loop do reference consecutive memory positions. The condition for this to happen even when the accesses of R depend on an `IF` statement is that the index for the first dimension of R only makes (sequential) progress within the same `IF` statement that controls R. As an example, this is what happens with references `B(posB)` and `jB(posB)` in the innermost loop of the CRS code (Fig. 4) that we use in Section 4 to validate our model: their index `posB` only advances when these references take place; thus consecutive accesses affect consecutive memory positions, even if the references are controlled by a condition.

- If i is not the innermost loop containing R, then

$$P_i(R,\vec{p}) = \begin{cases} p_i P_{i+1}(R,\vec{p}) & \text{if the index of loop } i+1 \text{ is not used in the references found in conditions that control } R, \\ p_i \overline{\overline{P_{i+1}(R,\vec{p})}^{\,G_{R_{i+1}}}} & \text{otherwise,} \end{cases}$$

where we must remember that $\overline{p} = 1-p$ and that p_i is the product of all the probabilities associated to the conditional sentences affecting R in nesting level i.

4. Validation

Our validation of the model is based on the comparison of its cache miss predictions with the result of trace-driven simulations. We have used three simple kernels shown in Figs. 3–5. The first code is a synthetic kernel with a conditional sentence that control the access to a data structure `C`. Then, Fig. 4 implements the storage of a matrix in CRS format (Compressed Row Storage), which is widely used to store sparse matrices in a compressed form. The code has two nested loops and

```
DO I = 1,M
  X = A(I)
  DO J = 1,N
    Y = B(J)
    IF (B(J) .GT. K) THEN
      C(J) = X + Y
    ENDIF
  ENDDO
ENDDO
```

Fig. 3. Synthetic kernel code.

```
posB = 1
DO I = 1, N
  offB(I) = posB
  DO J = 1, M
    IF (A(I,J) .NEQ. 0) THEN
      B(posB) = A(I,J)
      jB(posB) = J
      posB = posB + 1
    ENDIF
  ENDDO
ENDDO
```

Fig. 4. CRS storage algorithm.

```
DO I = 1, M
  DO K = 1, N
    IF (A(I,K) .NEQ. 0)
      DO J = 1, P
        IF (B(K,J) .NEQ. 0) THEN
          C(I,J) = C(I,J) + A(I,K) * B(K,J)
        ENDIF
      ENDDO
    ENDIF
  ENDDO
ENDDO
```

Fig. 5. Optimized product of matrices.

a conditional sentence that affects three of the references. Finally, Fig. 5 is an optimized product of matrices that contains references inside several nested conditional sentences. These conditionals try to avoid unuseful computations when one of their inputs is a zero.

In order to illustrate in detail our modeling strategy, we will explain step by step the modeling of the matrix product code, which is the most complex one. Then, the formulas for the references that experience non-regular access patterns in the other two codes will be provided for the sake of complexness. Finally, we will discuss the validation results.

4.1. Optimized product modeling

The code in Fig. 5 implements the product of two matrices, `A` and `B`, which may have many zero entries. As an optimization, when the element of `A` to be used in the current product is 0, then all its products with the corresponding elements of `B` are not performed. As an additional optimization, if the element of `B` to be used in the current product is 0 then that operation is not performed either. This avoids two floating point operations and the load and storage of `C(I,J)`.

Without loss of generality, we assume a compiler that maps scalar variables to registers and which tries to reuse the memory values recently read in processor registers. Under these conditions, the code in Fig. 5 contains three references to memory. The model in [7] can estimate the behavior of the reference `A(I,K)`, which takes place in every iteration of its enclosing loops. This, way we will focus our explanation on the modeling of the behavior of the references `C(I,J)` and `B(K,J)`, since the access to `A(I,K)` is not conditional, and thus it is already covered in previous publications.

4.1.1. Modeling of `C(I,J)`

The analysis of the behavior of this reference, which we will call R along this explanation for simplicity, begins in the innermost loop, in level two. In this level the loop variable indexes one of the references of one of the conditions that control the accesses of `C(I,J)`, so the PME for this loop will be Eq. (3). As for its parameters, since $S_{R2} = P$, then $L_{R2} = 1 + N$ and $G_{R2} \simeq 1$; and p_2 is the component in vector $\vec{p}$ associated to the probability that the condition inside the loop in nesting level 3 holds. Also, when expanding Eq. (4) we must take into account that this loop is in the innermost level, thus $F_3(R, \text{RegIn}, \vec{p}) = AV_0(\text{RegIn})$. After the simplification the formulation is

$$F_2(R, \text{RegIn}, \vec{p}) = p_2 PAV_0(\text{RegIn}).$$

In the next upper level, level one, the loop variable indexes also one reference of one of the conditions, so the same equations are to be applied. In this loop, $S_{R1} = 0$, $L_{R1} = 1$ and $G_{R1} = N$, so

$$F_1(R, \text{RegIn}, \vec{p}) = p_1 \sum_{j=1}^{N} \text{WMR}_1(R, \text{RegIn}, j, \vec{p}).$$

In order to compute WMR_1 we need to calculate the value for two functions. One is $P_1(R, \vec{p})$, which for our reference takes the value p_1p_2, where p_i is the ith element in vector $\vec{p}$. The other one is $\text{Reg}_1(R,i)$, the region accessed during i iterations of loop 1 that can interfere with the accesses of our reference:

$$\text{Reg}_1(R,i) = R_{\text{rp}_{\text{self}}}(P,1,M,1-(1-p_1p_2)^i) \cup R_\text{r}(i,1,M) \cup R_\text{rp}(P,i,N,1-(1-p_1p_2)^i).$$

The first term is associated to the self-interference of the reference we are studying. It is associated to the access to P groups of one element with stride M and every access takes place with a given probability. This access pattern was analyzed in Section 3.1.2, where the calculation of its cross-interference area vector was explained in detail. The self-interference area vector, which would be the one to apply in this equation, follows similar steps. The second term, $R_\text{r}(i,1,M)$, represents the access to i groups of 1 element separated by a distance M. The last term represents the access to P groups of i elements separated by a constant stride N, each individual access taking place with a given probability $1-(1-p_1p_2)^i$. Here the cross-interference area vector is used, so the explanation in Section 3.1.2 applies.

In the outermost level, the loop variable indexes a reference used in one of the conditions. As a result, Eq. (3) is to be applied again. In this case, $S_{R0}=1$, $L_{R0}=1+\lfloor (M-1)/L_\text{s}\rfloor$ and $G_{R0}\simeq L_\text{s}$, so the formulation is

$$F_0(R,\text{RegIn},\vec{p}) = (1+\lfloor (M-1)/L_\text{s}\rfloor) \times \sum_{j=1}^{L_\text{s}} \text{WMR}_0(R,\text{RegIn},0,j,\vec{p}).$$

As before, two functions must be evaluated to compute WMR_0. They are $P_0(R,\vec{p}) = 1-(1-p_1p_2)^M$ and $\text{Reg}_0(R,i)$, given by

$$\text{Reg}_0(R,i) = R_{\text{rp}_{\text{self}}}(P,1,M,1-(1-p_1p_2)^N) \cup R_\text{r}(N,i,M) \cup R_l(PN,1-(1-p_1)^{L_\text{s}}).$$

The first term is associated to the self-interference of our reference, which is the access to P groups of one element separated by a difference M and every access takes place with a given probability. The second term represents the access to N groups of i elements separated by a distance M. The last element represents the access to PN consecutive elements with a given probability.

4.1.2. Modeling of `B(K,J)`

The innermost loop for this reference, which we will now call R along this section, is also the one in level 2. The variable that controls this loop, `J`, is not used in the indexing of referenced found in conditions that control the execution of this reference, thus Eq. (1) is to be applied. As this is the innermost loop, in the evaluation of this equation, $F_3(R,\text{RegIn},\vec{p}) = AV_0 RegIn$. Since $S_{R2}=N$ and $L_{R2}=P$, the formulation for this nesting level is

$$F_2(R,S(\text{RegIn}),\vec{p}) = PAV_0(RegIn).$$

The next level is level one. In this level the variable of the loops indexes references in the two conditional statements than affect our reference, so Eq. (3) applies again. In this case, $S_{R1}=1$, $L_{R1}=1+\lfloor (N-1)/L_\text{s}\rfloor$and $G_{R1}\simeq L_\text{s}$, so the formulation is

$$F_1(R,\text{RegIn},\vec{p}) = p_1(1+\lfloor (N-1)/L_\text{s}\rfloor) \times \sum_{j=1}^{L_\text{s}} \text{WMR}_1(R,\text{RegIn},j,\vec{p}).$$

We need to know $P_1(R,\vec{p}) = p_1$ and the value of the accessed regions $\text{Reg}_1(R,i)$ to compute WMR_1:

$$\text{Reg}_1(R,i) = R_{\text{r}_{\text{self}}}(P,1,N) \cup R_\text{r}(i,1,M) \cup R_{\text{r}p}(P,1,M,p_2).$$

The first term is associated to the self-interference of `B`, which is the access to P groups of 1 elements separated with stride N. The second term represents the access to `A`: i groups of one element separated by a distance M. The last element describes the access to `C`: P groups of one element separated by a distance M, every access takes place with a given probability p_2.

In the outermost level, the variable of the loop indexes a reference in one of the conditions, so we have to apply again Eq. (3). For this loop and reference, $S_{R0}=0$, $L_{R0}=1$ and $G_{R0}=M$, so the formulation is

$$F_0(R,\text{RegIn},\vec{p}) = \sum_{j=1}^{M} \text{WMR}_0(R,\text{RegIn},j,\vec{p}).$$

In this loop, WMR_0 is a function of $P_0(R,\vec{p}) = 1-(1-p_1)^{L_\text{s}}$ and the value of the accessed regions $\text{Reg}_0(R,i)$:

$$\text{Reg}_0(R,i) = R_{l_{\text{self}}}(PN,1-(1-p_1)^{L_\text{s}}) \cup R_\text{r}(N,i,M) \cup R_\text{rp}(P,i,M,1-(1-p_1p_2)^N).$$

The first term is associated to the self-interference of our reference, which is the access to PN elements with a given probability. The second term represents the access to N groups of i elements separated by a distance M. The last element represents the access to P groups of i elements separated by a distance M, every access takes place with a given probability.

4.2. PMEs for the irregular accesses in the synthetic benchmark

In the synthetic benchmark in Fig. 3 the only reference that generates an irregular access pattern is `C(J)`, and it is due to the enclosing `IF` structure that depends on a condition on B(J). The PME that reflects the behavior of `C(J)` in the innermost loop is

$$F_1(R, \mathrm{RegIn}, \vec{p}) = p_1 N/L_s \sum_{j=1}^{L_s} \mathrm{WMR}_1(R, \mathrm{RegIn}, j, \vec{p}),$$

substituting $L_{Ri} = N/L_s$ and $G_{Ri} = L_s$ for $i = 1$ in Eq. (3). In the calculation of $\mathrm{WMR}_1(R, \mathrm{RegIn}, j, \vec{p})$ in Eq. (4) we would use $P_1(R, \vec{p}) = p_1$ and $\mathrm{Reg}_1(R, i) = R_s(i) \cup R_{l_{self}}(i, 1 - (1 - p_1)^{\min(i, ls)})$.

The PME associated to the behavior of `C(J)` in the outermost loop, which provides the prediction for the whole nest for this reference, is

$$F_0(R, \mathrm{RegIn}) = F_1(R, \mathrm{RegIn}) + (M - 1)F_1(R, \mathrm{Reg}_0(R, 1)),$$

substituting $N_i = M$ and $L_{Ri} = 1$ for $i = 0$ in Eq. (1). In this PME, $\mathrm{Reg}_0(R, 1) = R_s(1) \cup R_s(N) \cup R_{l_{self}}(N, 1 - (1 - p_1)^{L_s})$.

4.3. PMEs for the irregular accesses in the CRS benchmark

The references that generate irregular accesses in the CRS storage algorithm depicted in Fig. 4 are `B(posB)` and `jB(posB)`, which are controlled by a condition on `A(I,J)`. Both references follow exactly the same irregular access pattern, so we only provide here the formulas for the modeling of `B(posB)`, as those of `jB(posB)` are analogous. Starting the analysis in the innermost loop, we get

$$F_1(R, \mathrm{RegIn}, \vec{p}) = pM/L_s \sum_{j=1}^{L_s} \mathrm{WMR}_1(R, \mathrm{RegIn}, j, \vec{p}),$$

substituting $p_i = p$, $L_{Ri} = M/L_s$ and $G_{Ri} = L_s$ for $i = 1$ in Eq. (3). In the calculation of $\mathrm{WMR}_1(R, \mathrm{RegIn}, j, \vec{p})$ in Eq. (4) we would use $P_1(R, \vec{p}) = 1$ and $\mathrm{Reg}_1(R, i) = R_s(i) \cup R_s(ip)$.

Finally, in the outermost loop, the number of misses can be predicted as

$$F_0(R, \mathrm{RegIn}) = NF_1(R, \mathrm{RegIn}),$$

substituting $N_i = N$ and $L_{Ri} = N$ for $i = 0$ in Eq. (1).

4.4. Validation results

In order to validate our model its predictions were compared with the results of trace drive simulations using different cache configurations, problem sizes and probabilities for the fulfillment of the conditionals for the three example codes. The combinations used to validate the model for each code are shown in Table 1. Rows M, N and P correspond to the problem size, this is, the number of iterations of each loop, expressed as the value of its upper limit. Then come the probabilities p_i that the conditional sentences found in the codes are true. The synthetic and the CRS codes have a single conditional and no P loop, thus rows P and p_2 are empty for them. Then, the cache configurations used in the validation are shown in the format (C_s–L_s–K), this is (cache size–line size–associativity). The cache and line sizes are expressed in words or elements of the matrices accessed, not in bytes. Then, Table 1 shows the total number of parameter combinations tried for each code taking into account the previous rows. For each one of these combinations a total of twenty five different simulations were made using different base addresses for the data structures. This improves the validation of the model by taking into account many different relative positions for the mapping on the cache of the different data structures. The last two rows in the table show the average and the maximum value for each code of the metric Δ_{MR} that we use to measure the accuracy of the model. This metric is the average of the absolute value of the difference between the predicted and the measured miss rate (MR) in each one of the 25 simulations performed for each parameter combination. As expected, the average and maximum errors grow with the complexity of the code. Still, we consider that a maximum absolute error of only about 11% is very satisfactory. Also, the large difference between the average and the maximum Δ_{MR} shows that (relatively) large errors are very infrequent and, in general, the predictions estimate well the cache behavior.

Table 1
Parameter combinations used for the validation and average and maximum miss rate prediction error

Parameter	Kernel		
	Synthetic	CRS	Matrix product
M	950, 1750, 2000, 4500, 6000	1000, 1200, 1400, 1600, 1800	350, 550, 400, 600
	1200, 2500, 3000, 4000, 9500	1250, 1350, 2450, 2650, 3000	250, 350, 450, 650
P	–	–	600, 700, 750, 800
p_1	0.1, 0.2, 0.3, 0.4, 0.5	0.1, 0.2, 0.3, 0.4, 0.5	0.1, 0.2, 0.3, 0.4
p_2	–	–	0.1, 0.2, 0.3, 0.4
	4K–4–1	4K–4–1	4K–4–1
Cache	4K–4–2	4K–4–2	4K–4–2
Configurations	8K–4–1	8K–4–1	–
(C_s–L_s–K)	8K–4–2	8K–4–2	8K–4–2
Sizes in words	16K–8–2	16K–8–2	16K–8–2
Combinations	625	625	4096
Avg Δ_{MR}	0.22%	1.43%	2.23%
Max Δ_{MR}	3.81%	8.05%	11.32%

Tables 2–4 show the validation results for some randomly chosen combinations of the problem size, the conditional probabilities and the cache configurations for the three codes proposed in Figs. 3–5, respectively. The columns in the three tables have the same meaning as the respective rows in Table 1. Many of the combinations chosen in these tables do not belong to the set of experiments described by

Table 2
Validation data for the synthetic kernel in Fig. 3 for several cache configurations, problem sizes and condition probabilities

M	N	p	C_s	L_s	K	Δ_{MR}	T_{sim}	T_{exe}	T_{mod}
50,000	47,500	0.4	16K	8	2	0.015	182.211	68.022	0.005
50,000	47,500	0.2	8K	32	4	0.004	138.187	50.003	0.005
22,000	14,500	0.4	32K	16	4	0.001	28.244	7.033	0.003
22,000	14,500	0.4	8K	8	1	0.067	65.002	7.129	0.004
18,000	22,000	0.2	32K	16	2	0.574	23.021	7.586	0.004
18,000	22,000	0.1	16K	8	2	0.076	22.112	6.012	0.004
18,000	22,000	0.3	4K	32	4	0.141	95.223	8.010	0.004
14,500	19,500	0.7	64K	8	8	0.000	32.224	7.697	0.005
14,500	19,500	0.2	16K	4	2	0.252	20.269	5.331	0.005
14,500	19,500	0.3	8K	4	1	0.124	20.901	6.465	0.004
1750	1750	0.4	8K	4	8	0.000	1.123	1.000	0.003
1750	1750	0.7	8K	8	4	0.000	0.988	0.322	0.003

Table 3
Validation data for the CRS code in Fig. 4 for several cache configurations, problem sizes and condition probabilities

M	N	p	C_s	L_s	K	Δ_{MR}	T_{sim}	T_{exe}	T_{mod}
6200	10,150	0.4	32K	8	4	0.01	16.308	4.022	1.225
4200	17,150	0.1	4K	4	2	0.04	14.797	6.401	0.246
16,220	7200	0.2	16K	4	2	0.03	27.477	5.011	3.646
6200	14,250	0.3	32K	8	4	0.00	21.089	5.891	1.221
9200	14,250	0.1	4K	4	8	0.04	37.768	11.001	1.196
1100	15,550	0.5	4K	4	8	0.02	2.724	1.668	0.021
2900	17,250	0.3	32K	16	4	0.17	10.363	4.573	0.572
8900	9250	0.1	64K	8	4	0.64	17.119	11.228	2.516
4200	12,150	0.1	4K	4	2	0.04	9.364	3.880	0.246
5000	15,000	0.3	32K	8	4	0.11	17.852	10.330	0.804
7200	12,250	0.1	4K	4	8	0.04	18.224	9.646	0.721

Table 4
Validation data for the optimized matrix product code in Fig. 5 for several cache configurations, problem sizes and condition probabilities

M	N	P	p_1	p_2	C_s	L_s	K	Δ_{MR}	T_{sim}	T_{exe}	T_{mod}
750	750	1000	0.2	0.1	16K	8	8	0.79	24.444	11.233	0.203
750	750	1000	0.8	0.3	16K	16	16	1.31	86.845	72.069	0.987
900	850	900	0.9	0.1	64K	8	8	0.59	85.358	65.266	0.990
900	950	1500	0.1	0.4	32K	8	4	6.62	31.768	16.201	0.511
900	950	1500	0.8	0.3	16K	4	2	2.04	171.755	85.023	0.149
1000	850	900	0.7	0.5	4K	8	2	3.13	110.328	108.211	0.139
200	250	150	0.8	0.2	16K	4	2	0.48	0.764	0.550	1.034
200	250	150	0.1	0.3	32K	8	4	5.91	0.134	0.112	0.301
200	250	150	0.3	0.1	4K	4	8	1.45	0.406	0.323	0.030
100	350	90	0.8	0.5	4K	4	8	0.14	0.500	0.201	0.031
100	350	90	0.4	0.4	8K	8	4	0.40	0.218	0.122	0.586
100	350	90	0.2	0.3	4K	8	2	0.05	0.104	0.101	0.309

Table 1, so that the behavior of the model can be analyzed for a wider scope of parameters. The last three columns in each table correspond, respectively, to the simulation time, execution time and modeling times expressed in seconds and measured in a Athlon 2400 processor-based system (2086 GHz). As we see, modeling times are much shorter than trace-driven simulation times despite the fact hat we use a very fast and simple simulator. In fact, many times they are even faster than the native execution times. Furthermore, sometimes modeling times are several orders of magnitude shorter than trace-driven simulation and even execution times. The modeling time does not include the time required to build the formulas for the example codes. This will be made automatically by a tool we are currently developing. According to our experience in [10], the overhead of such tool is negligible.

Figs. 6 and 7 show the evolution of both the number of misses and the miss rate measured and predicted for different cache configurations and probabilities of the conditionals for the CRS and the matrix product codes, respectively. The figures show, as the previous tables, that the model is successful in predicting the behavior of the cache. A new interesting conclusion we can draw from these figures is that our extended model is indeed required to predict correctly the behavior of the memory hierarchy when irregular access patterns are involved. We can see that a simplified model that did not support irregular access patterns and which

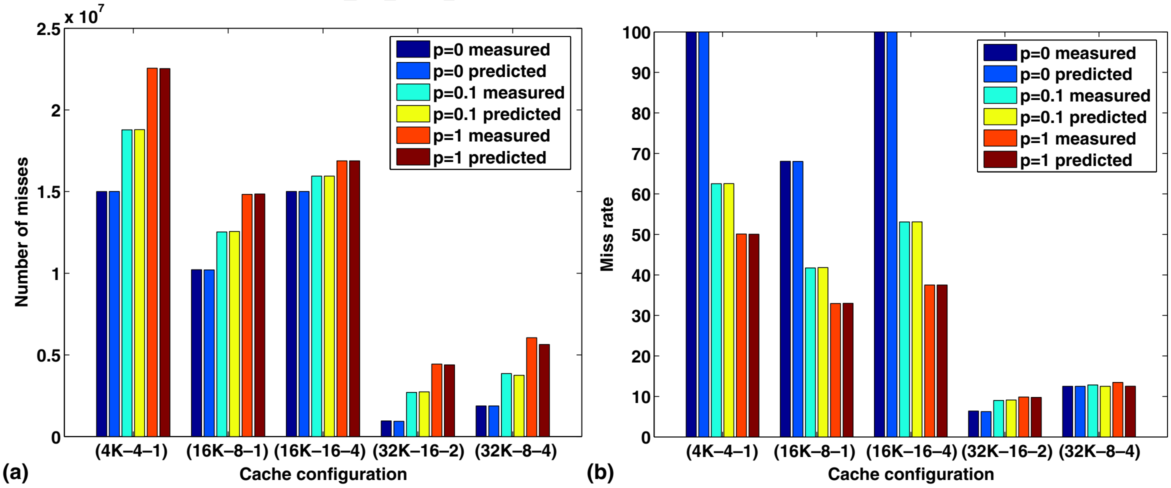

Fig. 6. Measured versus predicted (a) misses and (b) miss rates for several cache configurations and different probabilities of verification of the conditionals for the CRS code with $M = 1500$ and $N = 10{,}000$. The cache configurations are expressed as (C_s–L_s–K), with sizes in matrix elements.

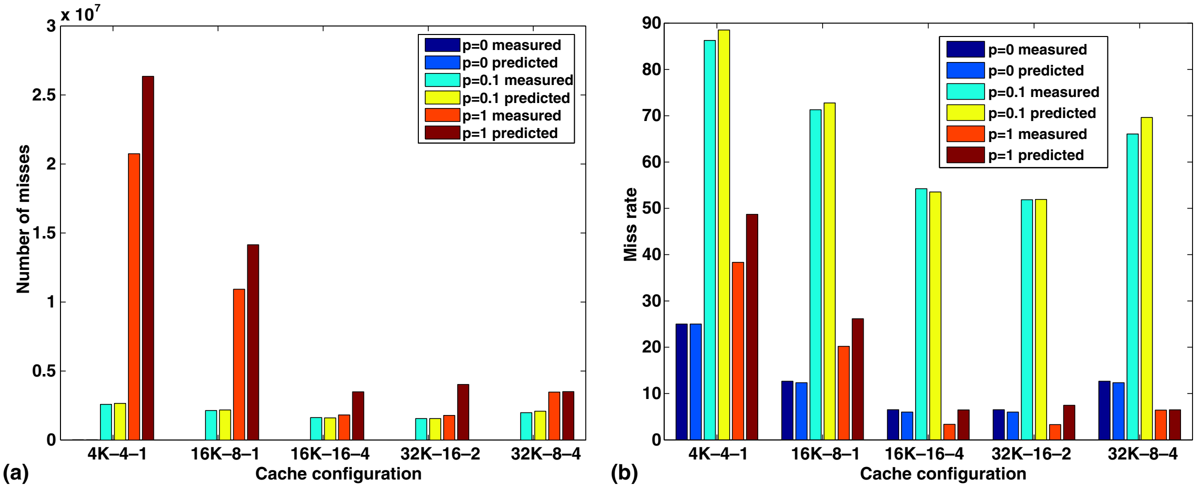

Fig. 7. Measured versus predicted (a) misses and (b) miss rates for several cache configurations and different probabilities of verification of the conditionals for the optimized matrix product code with $M = 300$, $N = 300$ and $P = 300$. The cache configurations are expressed as (C_s–L_s–K), with sizes in matrix elements.

chose to make all probabilities either 0 or 1 (the two extremes cases) would yield predictions very different from the real values obtained for intermediate probabilities like 0.1, shown in the figures. This justifies the interest of our research.

5. Related work

There are a number of previous works that also try to study and improve the behavior of the memory hierarchy by means of analytical models based on the structure of the code. Among those works we find [11], which is restricted to the modeling of direct-mapped caches and that lacks an automatic implementation. Later [12,4], overcame some of these limitation. This way [12], is based on the construction of the cache miss equations (CMEs), which are lineal system of Diophantine equation, where each solution corresponds to a potential cache miss. One of its main limitations is its high computing cost. The computing times required by [4] are much shorter, and similar to those of our model, however, its errors are larger than those of our model. Both works share the limitation that their modeling is only applicable to regular access patterns found in perfectly nested loops, and they do not take into account the possible reuses in structures that have been accessed in previous loops. This is a very important subject, as most misses in numerical codes are inter-nest misses [13], which implies that optimizations should consider several nests.

More recently [5,6], allow the analysis of non-perfectly nested loops and consider the reuse between loops in different nests. The former is based on Presburger formulas and provides very accurate estimations for small kernels but it can only handle modest levels of associativity (for example its validation only considers degrees of associativity one and two), and it is very time-consuming, which reduces its applicability. In fact, running a simulation is much faster than solving the equations this model generates. As for the latter, it is based on the extension of [14] in order to quantify the reuse, and it applies the CMEs of [12] in order to estimate the number of misses. The time it requires to solve the CMEs is reduced considerably by applying statistical techniques that allow to provide a prediction within a confidence interval. This model can analyze complete programs, imposing the conditions that the accesses follow regular patterns and that the codes do not contain data-dependent constructions, neither in the loop conditions nor in the conditional sentences. The model precision is similar to that of ours in most of the cases, however its computing times are longer. In a later work [9], this model was extended to consider continual sentences that could be analyzed statically at compile-time and were based on the indexes of the loops, not on the data read or computed in the program. These conditionals follow predictable and mostly regular access patterns, so there is little relation to our work.

Unlike our model, all these approaches require knowing the base addresses of the data structures. This restricts their scope of application, as these addresses are not available in many situations (physically-addressed caches, dynamically allocated data structures,...). Besides, none of them can model codes with data-dependent conditions. Indeed, it is the probabilistic nature of our model what allows us to consider this broad scope of codes.

6. Conclusions

In this work we have presented an extension to the PME model described in [7]. The extension allows this model to be the first one that can analyze codes with data-dependent conditionals. The extended model can handle conditionals nested in any arbitrary way that can affect isolated references or whole loop nests. We are currently limited by the fact that the conditions must follow an uniform distribution, but we think our research is an important step in the direction of broadening the scope of applicability of analytical models. This raises the possibility of driving compiler optimizations for codes with irregular access patterns based on compile-time estimations of the model, and helps understand better the complex behavior of these codes. Our experiments show that the model provides accurate estimations of the number of misses generated by a given code while requiring quite short computing times. Typical prediction errors and within 2% of the miss rate, and maximum errors, which are quite infrequent, range between 3.8% and 11.3% depending on the complexity of the code.

We are now working in an extension of our model to consider non-uniform distributions of probability for the accesses. We are also developing an automatic implementation of the extension of the model described in this paper in order to integrate it in a compiler framework, in a similar way to what was done with the original model [10]. We plan to use the Polaris [15] compiler framework as platform for this purpose, although the model can be coupled with any other front-end and used to model any programming language. As for the scope of the program structures that we wish to be amenable to analysis using the PME model, our next step will be to consider codes with irregular accesses due to the use of indirections or pointers.

References

[1] R. Uhlig, T. Mudge, Trace driven memory simulation: a survey, ACM Computing Surveys 29 (2) (1997) 128–170.
[2] M. Zagha, B. Larson, S. Turner, M. Itzkowitz, Performance analysis using the MIPS R10000 performance counters, in: ACM (Ed.), Proc. Supercomputing '96 Conference, ACM Press and IEEE Computer Society Press, 1996, pp. 17–22.
[3] S. Ghosh, M. Martonosi, S. Malik, Cache miss equations: an analytical representation of cache misses, in: Proc. 11th ACM Int'l. Conf. on Supercomputing (ICS'97), ACM Press, 1997, pp. 317–324.
[4] J.S. Harper, D.J. Kerbyson, G.R. Nudd, Analytical modeling of set-associative cache behavior, IEEE Transactions on Computers 48 (10) (1999) 1009–1024.
[5] S. Chatterjee, E. Parker, P. Hanlon, A. Lebeck, Exact analysis of the cache behavior of nested loops, in: Proc. of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation (PLDI'01), 2001, pp. 286–297.
[6] X. Vera, J. Xue, Let's study whole-program behaviour analytically, in: Proc. of the 8th Int'l Symposium on High-Performance Computer Architecture (HPCA8), 2002, pp. 175–186.
[7] B.B. Fraguela, R. Doallo, E.L. Zapata, Probabilistic miss equations: evaluating memory hierarchy performance, IEEE Transactions on Computers 52 (3) (2003) 321–336.
[8] Y. Paek, J. Hoeflinger, D. Padua, Simplification of array access patterns for compiler optimizations, ACM SIGPLAN Notices 33 (5) (1998) 60–71.
[9] X. Vera, J. Xue, Efficient compile-time analysis of cache behaviour for programs with IF statements, in: International Conference on Algorithms and Architectures for Parallel Processing, 2002, pp. 396–407.
[10] B.B. Fraguela, J. Tourino, R. Doallo, E.L. Zapata, A compiler tool to predict memory hierarchy performance of scientific codes, Parallel Computing 30 (2) (2004) 225–248.
[11] O. Temam, C. Fricker, W. Jalby, Cache interference phenomena, in: Proc. Sigmetrics Conference on Measurement and Modeling of Computer Systems, ACM Press, 1994, pp. 261–271.
[12] S. Ghosh, M. Martonosi, S. Malik, Cache miss equations: a compiler framework for analyzing and tuning memory behavior, ACM Transactions on Programming Languages and Systems 21 (4) (1999) 702–745.
[13] K.S. McKinley, O. Temam, Quantifying loop nest locality using SPEC'95 and the perfect benchmarks, ACM Transactions on Computer Systems 17 (4) (1999) 288–336.
[14] M.E. Wolf, M.S. Lam, A data locality optimizing algorithm, in: Proc. of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, 1991, pp. 30–44.
[15] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, P. Tu, Parallel programming with Polaris, IEEE Computer 29 (12) (1996) 78–82.

Diego Andrade received the MS degree in computer science from the University of A Coruna, Spain, in 2002. He is currently a PhD student in the Department of Electronics and Systems in the University of A Coruna. His research interests are about evaluation and prediction, analytical modeling, high performance simulation and compiler transformations.

Basilio B. Fraguela received both the MS degree and the PhD degree in computer science from the University of A Coruna, Spain, in 1994 and 1999, respectively. He is currently an associate professor in the Department of Electronics and Systems of the University of A Coruna. His primary research interests are in the fields of performance evaluation and prediction, analytical modeling, high performance simulation and compiler transformations.

Ramon Doallo received his BS and MS degrees in Physics from the University of Santiago de Compostela in 1987, and his PhD in Physics from the University of Santiago de Compostela in 1992. In 1990 he joined as associate professor the Department of Electronics and Systems of the University of A Coruna, Spain, where he became full professor in 1999. He has extensively published in the areas of computer architecture, and parallel and distributed computing. He is member of the IEEE Computer Society.