Modeling the Cache Behavior of Codes with Arbitrary Data-Dependent Conditional Structures*

Diego Andrade, Basilio B. Fraguela, and Ramón Doallo

Universidade da Coruña
Dept. de Electrónica e Sistemas
Facultade de Informática
Campus de Elviña, 15071 A Coruña, Spain
{dcanosa,basilio,doallo}@udc.es

Abstract. Analytical modeling is one of the most interesting approaches to evaluate the memory hierarchy behavior. Unfortunately, models have many limitations regarding the structure of the code they can be applied to, particularly when the path of execution depends on conditions calculated at run-time that depend on the input or intermediate data. In this paper we extend in this direction a modular analytical modeling technique that provides very accurate estimations of the number of misses produced by codes with regular access patterns and structures while having a low computing cost. Namely, we have extended this model in order to be able to analyze codes with data-dependent conditionals. In a previous work we studied how to analyze codes with a single and simple conditional sentence. In this work we introduce and validate a general and completely systematic strategy that enables the analysis of codes with any number of conditionals, possibly nested in any arbitrary way, while allowing the conditionals to depend on any number of items and atomic conditions.

1 Introduction

Memory hierarchies try to cushion the increasing gap between the processor and the memory speed. Fast and accurate methods to evaluate the performance of the memory hierarchies are needed in order to guide the compiler in choosing the best transformations and parameters for them when trying to make the optimal usage of this hierarchy. Trace-driven simulation [1] was the preferred approach to study the memory behavior for many years. This technique is very accurate, but its high computational cost makes it unsuitable for many applications. This way, analytical modeling, which requires much shorter computing times than previous approaches and provides more information about the reasons for the predicted

* This work has been supported in part by the Ministry of Science and Technology of Spain under contract TIC2001-3694-C02-02, and by the Xunta de Galicia under contract PGIDIT03-TIC10502PR.

behavior, has gained importance in recent years [2–4]. Still, it has important drawbacks like the lack of modularity in some models, and the limited set of codes that they can model.

In this work we present an extension to an existing analytical model that allows to analyze codes with any kind of conditional sentences. The model was already improved in [5] to enable it to analyze codes with a reference inside a simple and single conditional sentence. We now extend it to analyze codes with any kind and number of conditional sentences, even with references controlled by several nested conditionals, and nested in any arbitrary way. Like in previous works, we require the verification of the conditions in the `IF` statements to follow an uniform distribution.

This model is built around the idea of the Probabilistic Miss Equations (PMEs) [2]. These equations estimate analytically the number of misses generated by a given code in set-associative caches with LRU replacement policy. The PME model can be applied both to perfectly nested loops and imperfectly nested loops, with one loop per nesting level. It allows several references per data structure and loops controlled by other loops. Loop nests with several loops per level can also be analyzed by this model, although certain conditions need to be fulfilled in order to obtain accurate estimations. This work is part of an ongoing research line whose aim is to build a compiler framework [6, 7], which extracts information from the analytical modeling, in order to optimize the execution of complete scientific codes.

The rest of the paper is organized as follows. The next section presents some important concepts to understand the PME model and our extension. Section 3 describes the process of formulation after adapting the previous existing model to the new structures it has to model. In Sect. 4 we describe the process of validation of our model, using codes with several conditional sentences. A brief review of the related work is presented in Sect. 5, followed by our conclusions and a discussion on the future work in Sect. 6.

2 Introduction to the PME Model

The Probabilistic Miss Equations (PME) model, described in [2], generates accurately and efficiently cache behavior predictions for codes with regular access patterns. The model classifies misses as either compulsory or interference misses. The former take place the first time that the lines are accessed, while the latter are associated to new accesses for which the corresponding cache line has been evicted since its previous access. The PME model builds an equation for each reference and nesting level that encloses the reference. This equation estimates the number of misses generated by the reference in that loop taking into account both kinds of misses. Its probabilistic nature comes from the fact that interference misses are estimated through the computation of a miss interference probability for every attempt of reuse of a line.

2.1 Area Vectors

The miss probability when attempting to reuse a line depends on the cache footprint of the regions accessed since the immediately preceding reference to the considered line. The PME model represents these footprints by means of what it calls *area vectors.* Given a data structure V and a k-way set-associative cache, $S_V = S_{V_0}, S_{V_1}, \ldots, S_{V_k}$ is the area vector associated with the access to V during a given period of the program execution. The i-th element, $i > 0$, of this vector represents the ratio of sets that have received $k - i$ lines from the structure; while S_{V_0} is the ratio of sets that have received k or more lines.

The PME model analyzes the access pattern of the references for each different data structure found in a program and derives the corresponding area vectors from the parameters that define those access patterns. The two most common access patterns found in the kind of codes we intend to model are the sequential access and the access to groups of elements separated by a constant stride. See [2] for more information about how the model estimates the area vectors from the access pattern parameters.

Due to the existence of references that take place with a given probability in codes with data-dependent conditionals, a new kind of access pattern arises in them that we had not previously analyzed. This pattern can ben described as an access to groups of consecutive elements separated by a constant stride, in which every access happens with a given fixed probability. The calculation of the area vector associated to this new access pattern is not included in this paper because of size limitations. This pattern will be denoted as $R_{rl}(M, N, P, p)$, which represents the access to M groups of N elements separated by a distance P where every access happens with a given probability p (see example in Sect. 4).

Very often, several data structures are referenced between two accesses to the same line of a data structure. As a result, a mechanism is needed to calculate the global area vector that represents as a whole the impact on the cache of the accesses to several structures. This is achieved by adding the area vectors associated to the different data structures. The mechanism to add two area vectors has also been described in [2], so although it is used in the following sections, we do not explain it here. The addition algorithm treats the different ratios in the area vectors as independent probabilities, thus disregarding the relative positions of the data structures in memory. This is in fact an advantage of the model, as in most situations these addresses are unknown at compile time (dynamically allocated data structures, physically indexed caches, etc.). This way, the PME model is still able to generate reasonable predictions in these cases, as opposed to most of those in the bibliography [3, 4], which require the base addresses of the data structures in order to generate their estimations. Still, when such positions are known, the PME model can estimate the overlapping coefficients of the footprints associated with the accesses to each one of the structures involved, so they can be used to improve the accuracy of the addition.

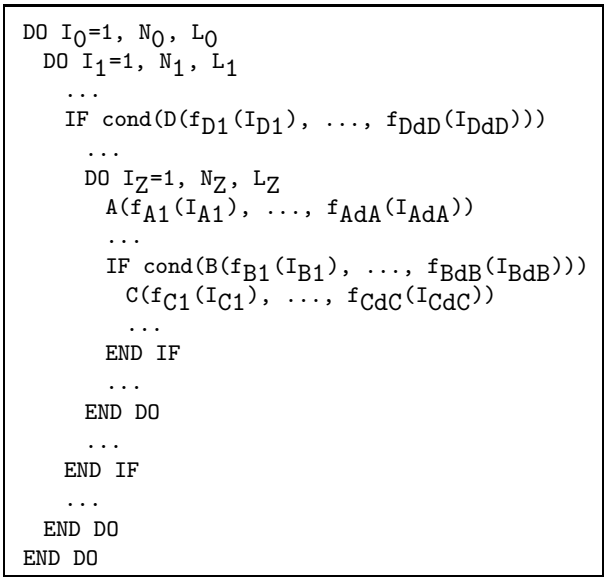

Fig. 1. Nested loops with several data-dependent conditions

2.2 Scope of Application

The original PME model in [2] did not support the modeling of codes with any kind of conditionals. Figure 1 shows the kind of codes that it can analyze after applying our extension. The figure shows several nested loops that have a constant number of iterations known at compile time. Several references, which need not be in the innermost nesting level, are found in the code. Some references are affected by one or more nested conditional sentences that depend on the data arrays. All the structures are indexed using affine functions of the loop indexes $f_{A1}(I_{A1}) = \alpha_{A1} I_{A1} + \delta_{A1}$. We assume also that the verification of the conditions in the `IF` statements follows an uniform distribution, although the different conditions may hold with different probabilities. Such probabilities are inputs to our model that are obtained either by means of profiling tools, or knowledge of the behavior of the application. We assume also the conditions are independent.

As for the hardware, the PME model is oriented to set-associative caches with LRU replacement policy. In what follows, we will refer to the total size of this cache as C_s, to the line size as L_s, and k will be the degree of associativity or number of lines per set.

3 Miss Equations

The PME model estimates the number of misses generated by a code using the concept of miss equation. Given a reference, the analysis of its behavior begins

in the innermost loop containing it, and proceeds outwards. In this analysis, a probabilistic miss equation is generated for each reference and in each nesting level that encloses it following a series of rules.

We will refer as $F_i(R, \text{RegInput}, \vec{p})$ to the miss equation for reference R in nesting level i. Its expression depends on RegInput, the region accessed since the last access to a given line of the data structure. Since we now consider the existence of conditional sentences, the original PME parameters have been extended with a new one, $\vec{p}$. 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 no conditionals are found in level j, then $p_j = 1$. When there are several nested `IF` statements in the same nesting level, p_j corresponds to the product of their respective probabilities of holding their respective conditions. This is a first improvement with respect to our previous approach [5], which used a scalar because only a single conditional was considered.

Depending on the situation, two different kinds of formulas can be applied:

- If the variable associated to the current loop i does not index any of the references found in the condition(s) of the conditional(s) sentence(s), then we apply a formula from the group of formulas called *Condition Independent Reference Formulas* (CIRF). This is the kind of PME described in [2].
- If the loop variable indexes any of such references, then we apply a formula from the group called *Condition Dependent Reference Formulas* (CDRF).

Another factor influencing the construction of a PME is the existence of other references to same data structure, as they may carry some kind of group reuse. For simplicity, in what follows we will restrict our explanation to references that carry no reuse with other references.

3.1 Condition Independent Reference Formulas

When the index variable for the current loop i is not among those used in the indexing of the variables referenced in the conditional statements that enclose the reference R, the PME for this reference and nesting level is given by

$$\begin{aligned} F_i(R, \text{RegInput}, \vec{p}) = & L_{Ri} F_{i+1}(R, \text{RegInput}, \vec{p}) \\ & + (N_i - L_{Ri}) F_{i+1}(R, \text{Reg}(A, i, 1), \vec{p}) \ , \end{aligned} \tag{1}$$

being N_i the number of iterations in the loop of the nesting level i, and L_{Ri} the number of iterations in which there is no possible reuse for the lines referenced by R. $\text{Reg}(A, i, j)$ stands for the memory region accessed during j iterations of the loop in the nesting level i that can interfere with data structure A.

The formula calculates the number of misses for a given 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 there can be no reuse in this loop. The miss probability for these iterations depends on the accesses and reference pattern in the outer loops. The second value corresponds to the iterations in which there

can be reuse of cache lines accessed in the previous iteration, and so it depends on the memory regions accesses during one iteration of the loop.

The indexes of the reference R are affine functions of the variables of the loops that enclose it. As a result, R has a constant stride S_{Ri} along the iterations of loop i. This value is calculated as $S_{Ri} = \alpha_{A_j} d_{A_j}$, where j is the dimension whose index depends on $\mathtt{I}_i$, the variable of the loop; α_{A_j} is the scalar that multiplies the loop variable in the affine function, and d_{A_j} is the size of the j-th dimension. If I_i does not index reference R, then $S_{Ri} = 0$. This way, L_{Ri} can be calculated as,

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

The formula calculates the number of accesses of R that cannot exploit either spatial or temporal locality, which is equivalent to estimating the number of different lines that are accessed during N_i iterations with stride S_{Ri}.

3.2 Condition Dependent Reference Formulas

If the index variable for the curret loop i is used in the indexes of the arrays used in the conditions that control the reference R, the behavior of R with respect to this loop is irregular. The reason is that different values of the index access different pieces of data to test in the conditions. This way, in some iterations the conditions hold and R is executed, thus affecting the cache, while in other iterations the associated conditions do not hold and no access of R takes place. As a result, the reuse distance for the accesses of R is no longer fixed: it depends on the probabilities $\vec{p}$ that the conditions that control the execution of R are verified. If the probabilities the different conditions hold are known, the number of misses associated to the different reuse distances can be weighted using the probability each reuse distante takes place.

As we have just seen, eq. (2) estimates the number L_{Ri} of iterations of the loop in level i in which reference R cannot explote reuse. Since the loop has N_i iterations, this means on average each different line can be reused in up to $G_{Ri} = N_i/L_{Ri}$ consecutive iterations. Besides, either directly reference R or the loop in level $i+1$ that contains it can be inside a conditional in level i that holds with probability p_i. Thus, $p_i L_{Ri}$ different groups of lines will be accessed on average, and each one of them can be reused up to G_{Ri} times. Taking this into account, the general form of a condition-dependent PME is

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

where $\text{WMR}_i(\text{RegInput}, j, \vec{p})$ is the weighted number of misses generated by reference R in level i considering the j-th attempt of reuse of the G_{Ri} ones potentially possible. As in Sect. 3.1, RegInput is the region accessed since the last access to a given line of the considered data structure when the execution

of the loop begins. Notice that if no condition encloses R or the loop around it in this level, simply $p_i = 1$.

The number of misses associated to reuse distance j weigthed with the probability an access with such reuse distante does take place, is calculated as

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

where $P_i(R, \vec{p})$ yields the probability that R accesses each of the lines it can potentially reference during one iteration of the loop in nesting level i. This probability is a function of those conditionals in $\vec{p}$ in or below the nesting level analyzed. The first term in (4) considers the case that the line has not been accessed during any of the previous $j-1$ iterations. In this case, the RegInput 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 $k = 1, \ldots, j-1$ previous iterations of the considered loop.

Line Access Probability The probability $P_i(R, \vec{p})$ that the reference R whose behavior is being analyzed does access 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 $\mathrm{WMR_i}(\mathrm{RegInput}, j, \vec{p})$, 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 i-th down to the one containing the reference. If fact, this probability is calculated recursively in the following way:

$$P_i(R, \vec{p}) = \begin{cases} p_i & \text{if } i \text{ is the innermost loop that contains } R \\ p_i P_{i+1}(R, \vec{p}) & \text{if the index of loop } i+1 \text{ is not used in the references in conditions that control } R \\ p_i(1 - (1 - P_{i+1}(R, \vec{p}))^{G_{R_{i+1}}}) & \text{otherwise} \end{cases} \tag{5}$$

where we must remember that p_i is the product of all the probabilities associated to the conditional sentences affecting R that are located in nesting level i.

This algorithm to estimate the probability of access per line at level i has been improved with respect to our previous work [5], as it is now able to integrate different conditions found in different nesting levels, while the previous one only considered a single condition.

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

Fig. 2. 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
          C(I,J)=C(I,J)+A(I,K)*B(K,J)
        ENDIF
      ENDDO
    ENDIF
  ENDDO
ENDDO
```

Fig. 3. Optimized product of matrices

3.3 Calculation of the Number of Misses

In the innermost level that contains the reference R, both in CIRFs and CDRFs, $F_{i+1}(R, \text{RegInput}, \vec{p})$, the number of misses caused by the reference in the immediately inner level is $AV_0(\text{RegInput})$, this is, the first element in the area vector associated to the region RegInput.

The number of misses generated by reference R in the analyzed nest is finally estimated as $F_0(R, \text{RegInput}_{\text{total}}, \vec{p})$ once the PME for the outermost loop is generated. In this expression, $\text{RegInput}_{\text{total}}$ is the total region, this is, the region that covers the whole cache. The miss probability associated with this region is one.

4 Model Validation

We have validated our model by applying it manually to the two quite simple but representative codes shown in Fig. 2 and Fig. 3. The first code implements the storage of a matrix in CRS format (Compressed Row Storage), which is widely used in the storage of sparse matrices. It has two nested loops and a conditional

sentence that affects three of the references. The second code is an optimized product of matrices; that consists of a nest of loops that contain references inside several nested conditional sentences.

Results for both codes will be shown in Sect. 4.2, but we will first focus on the second code in order to provide a detailed idea about the modeling procedure.

4.1 Optimized Product Modeling

This code is shown in Fig. 3. It implements the product between two matrix, A and B, with a uniform distribution of nonzero entries. As a first 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 a final optimization, if the element of `B` to be used in the current product is 0 then that operation is not performed. 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. 3 contains three references to memory. The model in [2] can estimate the behavior of the references `A(I,K)`, which take place in every iteration of their enclosing loops.

Thus, we will focus our explanation on the modeling of the behavior of the references `C(I,J)` and `B(K,J)` which are not covered in previous publications.

`C(I,J)` Modeling The analysis begins in the innermost loop, in level 2. In this level the loop variable indexes one of the reference of one of the conditions, so the CDRF formula must be applied.

As $S_{R2} = P$, $L_{R2} = 1 + N$, $G_{R2} \simeq 1$ and p_2 is the component in vector $\vec{p}$ associated to the probability that the condition inside the loop in nesting level 2 holds. This loop is in the innermost level. Thus, $F_3(R, \text{RegInput}, \vec{p}) = AV_0(\text{RegInput})$, then after the simplification the formulation is,

$$F_2(R, \text{RegInput}, \vec{p}) = p_2 PAV_0(\text{RegInput}) \ . \tag{6}$$

In the next upper level, level 1, the loop variable indexes one reference of one of the conditions, so the CDRF formula has to be applied. Let $S_{R1} = 0$, $L_{R1} = 1$ and $G_{R1} \simeq N$, then

$$F_1(R, \text{RegInput}, \vec{p}) = p_1 \sum_{j=1}^{N} \text{WMR}_1(R, \text{RegInput}, j, \vec{p}) \ . \tag{7}$$

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 $p_1 p_2$, where p_i is the i-th element in vector $\vec{p}$. The other one is $\text{Reg}(C, 1, i)$, the region accessed during i iterations of the loop 1 that can interfere with the accesses to `C`.

$$\begin{aligned}\mathrm{Reg}(C,1,i) =& R_{rl_{\mathrm{auto}}}(P,1,M,1-(1-p_1p_2)^i) \\ & \cup R_r(i,1,M) \cup R_{rl}(P,i,N,p_1)\ .\end{aligned} \tag{8}$$

The first term is associated to the autointerference of `C`, which is the access to P groups of one element separated by a difference M and every access takes places with a given probability. The second term represents the access to i groups of 1 element separated by a distance M. The last element represents the access to P groups of i elements separated by a distance N. Every access is going to happen with a given probability p_1.

In the outermost level, the loop variable indexes the reference of the condition. As a result, the CDRF formula is to be applied again. Being $S_{R0} = 1$, $L_{R0} = 1 + \lfloor (N-1)/L_s \rfloor$ and $G_{R0} \simeq L_s$, so the formulation is

$$F_0(R, \mathrm{RegInput}, \vec{p}) = (1 + \lfloor (N-1)/L_s \rfloor) \sum_{j=1}^{L_s} \mathrm{WMR}_0(R, \mathrm{RegInput}, j, \vec{p})\ . \tag{9}$$

As before, two functions must be evaluated to compute WMR_0. They are $P_0(R, \vec{p}) = 1 - (1-p_1p_2)^N$ and $\mathrm{Reg}(C, 0, i)$, given by

$$\begin{aligned}\mathrm{Reg}(C,0,i) =& R_{rl_{\mathrm{auto}}}(P,1,M,1-(1-p_1p_2)^N) \\ & \cup R_r(N,i,M) \cup R_l(PN,1-(1-p_1)^{L_s})\ .\end{aligned} \tag{10}$$

The first term is associated to the autointerference of `C`, which is the access to P groups of one element separated by a difference M and every access takes places 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.

`B(K,J)` Modeling The innermost loop for this reference is also the one in level 2. The variable that controls this loop, `J`, is found in the indexes of a reference found in the condition of an `IF` statements (in this case, the innermost one), one conditional, so a CDRF is to be built. As this is the innermost loop, we get $F_3(R, \mathrm{RegInput}, \vec{p}) = AV_0 RegInput$. Since $S_{R_i} = N$, $L_{R_i} = P$ and $G_{R_i} = 1$ the formulation for this nesting level is

$$F_2(R, S(\mathrm{RegInput}), \vec{p}) = PAV_0(\mathrm{RegInput})\ . \tag{11}$$

The next level is level 1. In this level the variable of the loops indexes any of the reference of any of the conditional, so we have to use the CDRF formula. Being $S_{R1} = 1$, $L_{R1} = 1 + \lfloor (N-1)/L_s \rfloor$ and $G_{R1} \simeq L_s$ the formulation is

$$F_1(R, \mathrm{RegInput}, \vec{p}) = p_1 \left(1 + \left\lfloor \frac{N-1}{L_s} \right\rfloor\right) \sum_{j=1}^{L_s} \mathrm{WMR}_1(R, \mathrm{RegInput}, j, \vec{p})\ . \tag{12}$$

Table 1. Validation data for the code in Fig. 2 for several cache configurations and different problem sizes and condition probabilities

M	N	p	C_s	L_s	K	Δ_{MR}	T_{sim}	T_{exe}	T_{mod}
6200	10150	0.4	32K	8	4	0.001	82	19	0.001
4200	17150	0.1	4K	4	2	0.401	107	18	0.001
16220	7200	0.2	16K	4	2	2.635	152	24	0.044
6200	14250	0.3	32K	8	4	0.005	146	22	0.001
9200	14250	0.1	4K	4	8	2.374	582	50	0.001
1100	15550	0.5	4K	4	8	0.027	2	1	0.001
2900	17250	0.3	32K	16	4	1.847	65	32	0.001
8900	9250	0.1	64K	8	4	3.055	118	46	0.010
4200	12150	0.1	4K	4	2	0.571	64	33	0.001
5000	15000	0.3	32K	8	4	0.183	125	54	0.001
7200	12250	0.1	4K	4	8	0.044	139	64	0.010

We need to know $P_1(R,\vec{p}) = p_1$ and the value of the accessed regions $\mathrm{Reg}(B,1,i)$ in order to compute WMR_1:

$$\mathrm{Reg}(B,1,i) = Rrl_{\mathrm{auto}}(P,i,N,p_1]) \cup R_r(i,1,M) \cup R_{rl}(P,1,M,p_1p_2)\ . \quad (13)$$

The first term is associated to the autointerference of `B`, which is the access to P groups of i elements separated by a difference N and every access takes places with a given probability. The second term represents the access to i groups of one element separated by a distance M. The last element represents the access to P groups of one element separated by a distance M, every access takes places with a given probability p_1p_2.

In the outermost level, the level 0, the variable of the loop indexes a reference in one of the conditions, so we have to apply again the CDRF formula. Being $S_{R0} = 0$, $L_{R0} = 1$, $G_{R0} \simeq M$, so the formulation is

$$F_0(R,\mathrm{RegInput},\vec{p}) = \sum_{j=1}^{M} \mathrm{WMR}_0(R,\mathrm{RegInput},j,\vec{p})\ . \quad (14)$$

In this loop, WMR_0 is a function of $P_0(R,\vec{p} = 1-(1-p_1)^{L_s}$ and the value of the accessed regions $\mathrm{Reg}(B,0,i)$:

$$\begin{aligned}\mathrm{Reg}(B,0,i) =& R_{l_{\mathrm{auto}}}(PN, 1-(1-p_1)^{L_s}) \cup R_r(N,i,M) \\ & \cup R_{rl}(P,i,M,1-(1-p_1p_2)^N)\ .\end{aligned} \quad (15)$$

The first term is associated to the autointerference of `B`, 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 places with a given probability.

Table 2. Validation data for the code in Fig. 3 for several cache configurations and different 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	0.808	35	17	0.075
750	750	1000	0.8	0.3	8K	16	16	5.081	164	62	0.053
900	850	900	0.8	0.1	16K	32	2	0.224	78	54	0.795
900	850	900	0.9	0.1	64K	8	8	0.589	136	66	0.523
900	950	1500	0.8	0.3	16K	4	2	2.411	236	159	0.110
900	950	1500	0.1	0.4	32K	8	4	5.408	51	38	0.357
1000	850	900	0.7	0.5	4K	8	2	4.394	98	97	0.054
200	250	150	0.8	0.2	16K	4	2	0.604	1	0	0.690
200	250	150	0.1	0.3	32K	8	4	2.161	0	0	0.145
200	250	150	0.3	0.1	4K	4	8	1.208	0	0	0.008
100	350	90	0.8	0.5	4K	4	8	0.070	0	1	0.042
100	350	90	0.4	0.4	8K	8	4	0.417	0	0	0.324
100	350	90	0.2	0.3	4K	8	2	0.744	0	0	0.218

4.2 Validation Results

We have done the validation by comparing the results of the predictions given by the model with the results of a trace-driven simulation. We have tried several cache configurations, problem sizes and probabilities for the conditional sentences.

Tables 1 and 2 display the validation results for the codes in Fig. 2 and 3, respectively. In Table 1 the two first columns contain the problem size and the third column stands for the probability p that the condition in the code is fulfilled. In Table 2 the first three columns contain the problem size, while the next two columns contain the probabilities p_1 and p_2 that each of of the two conditions in Fig. 3 is fulfilled. Then the cache configuration is given in both tables by C_s, the cache size, L_s, the line size, and the degree of associativity of the cache, K. The sizes are measured in the number of elements of the arrays used in the codes. The accuracy of the model is used by the metric Δ_{MR}, which is based on the miss rate (MR); it stands for the absolute value of the difference between the predicted and the measured miss rate.

For every combination of cache configuration, problem size and probabilities of the conditions, 25 different simulations have been made using different base addresses for the data structures.

The results show that the model provides a good estimation of the cache behavior in the two example codes. The last three columns in both tables reflect the corresponding simulation times, source code execution time and modeling times expressed in seconds and measured in a 2,08 Ghz AMD K7 processor-based system. We can see that the modeling times are much smaller than the trace-driven simulation and even execution times. Furthermore, 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 the tool we are currently developing. According to our experience in [2], the overhead of such tool is negligible.

5 Related Work

Over the years, several analytical models have been proposed to study the behavior of caches. Probably the most well-known model of this kind is [8], based on the Cache Miss Equations (CMEs), which are lineal systems of Diophantine equations. Its main drawbacks are its high computational cost and that it is restricted to analyzing regular access patterns that take place in isolated perfectly nested loops. In the past few years, some models that can overcome some of these limitations have arisen. This is the case of the accurate model based on Presburger formulas introduced in [3], which can analyze codes with non-perfectly nested loops and consider reuses between loops in different nesting levels. Still, it can only model small levels of associativity and it has a extremely high computational cost. More recently [4], which is based on [8], can also analyze these kinds of codes in competitive times thanks to the statistical techniques it applies in the resolution of the CMEs.

A more recent work [9], can model codes with conditional statements. Still, it does not consider conditions on the input or intermediate data computed by the programs. It is restricted to conditional sentences whose conditions refer to the variables that index the loops.

All these models and others in the bibliography have fundamental differences with ours. One of the most important ones is that all of them require a knowledge about the base address of the data structures. In practice this is not possible or useful in many situations because of a wide variety of reasons: data structures allocated at run-time, physically-indexed caches, etc. Also, thanks to the general strategy described in this paper, the PME model becomes the first one to be able to model codes with data-dependent conditionals.

6 Conclusions and Future Work

In this work we have presented an extension to the PME model described in [2]. The extension allows this model to be the first one that can analyze codes with data-dependent conditionals and considering, not only simple conditional sentences but also nested conditionals affecting a given reference. 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. Our validation shows that this model provides accurate estimations of the number of misses generated by a given code while requiring quite short computing times. In fact the model is typically two orders of maginute faster than the native execution of the code.

The properties of this model turn it into an ideal tool to guide the optimization process in a production compiler. In fact, the original PME model has

been used to guide the optimization process in a compiler framework [7]. We are now working in an automatic implementation of the extension of the model described in this paper in order to integrate it in that framework. 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. Uhlig, R., Mudge, T.: Trace-Driven Memory Simulation: A Survey. ACM Computing Surveys **29** (1997) 128–170
2. Fraguela, B.B., Doallo, R., Zapata, E.L.: Probabilistic Miss Equations: Evaluating Memory Hierarchy Performance. IEEE Transactions on Computers **52** (2003) 321–336
3. Chatterjee, S., Parker, E., Hanlon, P., Lebeck, A.: 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) 286–297
4. Vera, X., Xue, J.: Let's Study Whole-Program Behaviour Analytically. In: Proc. of the 8th Int'l Symposium on High-Performance Computer Architecture (HPCA8). (2002) 175–186
5. Andrade, D., Fraguela, B., Doallo, R.: Cache behavior modeling of codes with data-dependent conditionals. In Springer-Verlag, ed.: 7th Intl. Workshop on Software and Compilers for Embedded Systems, SCOPES 2003. Volume 2826 of Lecture Note in Computer Science. (2003) 373–387
6. Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., Tu, P.: Parallel Programming with Polaris. IEEE Computer **29** (1996) 78–82
7. Fraguela, B.B., Touriño, J., Doallo, R., Zapata, E.L.: A compiler tool to predict memory hierarchy performance of scientific codes. Parallel Computing **30** (2004) 225–248
8. Ghosh, S., Martonosi, M., Malik, S.: Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. ACM Transactions on Programming Languages and Systems **21** (1999) 702–745
9. Vera, X., Xue, J.: Efficient Compile-Time Analysis of Cache Behaviour for Programs with IF Statements. In: 5th International Conference on Algorthms and Archiectures for Parallel Processing. (2002) 396–407