Cache Behavior Modeling of Codes with Data-Dependent Conditionals

Diego Andrade, Basilio B. Fraguela, and Ramón Doallo

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

Abstract. The increasing gap between the speed of the processor and the memory makes the role played by the memory hierarchy essential in the system performance. There are several methods for studying this behavior. Trace-driven simulation has been the most widely used by now. Nevertheless, analytical modeling requires shorter computing times and provides more information. In the last years a series of fast and reliable strategies for the modeling of set-associative caches with LRU replacement policy has been presented. However, none of them has considered the modeling of codes with data-dependent conditionals. In this article we present the extension of one of them in this sense.

1 Introduction

The memory hierarchy plays an essential role in bridging the increasing gap between the processor and the memory speed. The optimal usage of the memory hierarchy is specially important in real-time systems and systems that require low power and energy consumption. This way, although the research in this area has traditionally focused on the optimization of codes executed in computers, we consider that its relevance is even greater in the field of the embedded systems. Programmers use many methods in order to improve the performance of the memory hierarchy during the execution of their codes. Unfortunately, the only tool available for a long time to study this behavior has been trace-driven simulation [1]. The main drawback of this method is the long computing time it requires. Some architectures implement built-in hardware counters [2], but their availability is limited to certain architectures. In addition, in both cases either the code or a simulation needs to be executed in order to obtain data on the memory hierarchy performance, and neither of them explains the observed behavior. Analytical models are faster than the previous methods and give us much

[⋆] This work has been supported in part by the Spanish Ministry of Science and Technology under contract TIC2001-3694-C02-02 more information. Many models of this kind have been proposed in the bibliography [3–5]. The main drawbacks of these models are the lack of modularity and the fact they can only model a limited set of program structures.

The model we propose in this paper is an extension of the probabilistic model introduced in [3]. That work proposes a very modular model, what makes it easily extensible. We have extended the set of code constructions it supports with data-dependent conditionals, a program structure that no previous work in this area has modeled. As a first step, we only consider conditions that follow an uniform distribution, but we regard this extension very interesting as a first step towards the study of whole real programs.

The model proposed in [3] builds automatically equations, referred as Probabilistic Miss Equations (PMEs), that estimate the number of misses that a given code generates. This method models the behavior of set-associate caches with LRU replacement policy. It is applicable to perfectly nested loops and non-perfectly 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 analysed by this model, although certain conditions need to be fulfilled in order to obtain accurate estimations.

This paper describes the extension of this model in order to consider codes with data-dependent conditionals that follow an uniform distribution. Sect. 2 presents the main concepts in which our model is based. Then, Sect. 3 introduces the area vector concept, which is used by our model to represent the impact of a series of accesses to a data structure on the cache. The strategy to build formulas that estimate the number of cache misses in codes containing data-dependent conditionals is explained in Sect. 4, which is followed by a validation using a simple code and trace-driven simulations in Sect. 5. Sect. 6 is a brief review of the related works. Finally, Sect. 7 is devoted to the conclusions and future work.

2 Modeling Concepts

We consider a cache with a size of C_{s} words, a line size of L_{s} words, an a associativity degree k, where we refer as word to the size of the elements of our data structures. There are two situations that can generate a miss in the access to a line. The first one is the first access to this line, which is known as an intrinsic miss. Each one of the remaining accesses will result in a miss if k or more different lines accessed since the last reference to that line are mapped to the same cache set. These misses are known as interference misses. This way, the probability an access results in an interference miss is equal to the probability that k or more lines have been mapped to cache set of the accessed line since the previous access to the line took place.

The misses generated by a reference can be estimated by means of a formula that includes the number of different lines it accesses (intrinsic misses), the number of line reuses it generates, and the interference probability for such accesses (interference misses). The calculation of this probability involves estimating the memory region accessed between each two consecutive accesses to the same line, and the mapping of this region on the cache. The miss probability will be equal to the ratio of sets that receive k or more different lines.

```
DO $I_0$=1, $N_0$
  DO $I_1$=1, $N_1$
  ...
    DO $I_Z$=1, $N_Z$
      A($f_{A1}$($I_{A1}$), ..., $f_{AdA}$($I_{AdA}$))
      ...
      IF B($f_{B1}$($I_{B1}$), ..., $f_{BdB}$($I_{BdB}$))
        C($f_{C1}$($I_{C1}$), ..., $f_{CdC}$($I_{CdC}$))
      ...
    END DO
  ...
  END DO
END DO
```

Fig. 1. Nested loops with data-dependent conditions

Figure 1 shows a nest of normalized loops that contains references inside data-dependent conditionals. This is the type of structures we consider in our extension. Our model considers references whose indexes are affine functions of the type $f_{A1}(I_{A1}) = \alpha_{A1} I_{A1} + \delta_{A1}$. The references can be found in any nesting level, not just in the innermost one. The number of iterations of every loop must be known at compile time and must be the same in every execution of the loop. The reuse among different references to the same data structure can be analyzed using our model only if those references are uniformly generated [6], that is, they only differ in one or more of the added δ constants. This is by far the most common situation in scientific codes. Uniformly generated references are typically found in the same scope in a given nest, as they use the same variables for their indexing. Thus, as a simplification, when there are references to the same data structure in different scopes of the same nest, their potential reuse is not considered. Still, if the references are found in different nests (which may share outer level loops), reuse is estimated following a conservative approach.

As for the conditional structures, in this work we consider conditions whose verification follows an uniform distribution, as stated in the introduction. This means that in every evaluation of the condition there is a constant probability p that it is fulfilled.

3 Area Vectors

Miss probabilities are calculated using area vectors. These vectors represent the impact on the cache of the accesses to one or several data structures. Given a data structure V, $S_V = S_{V_0}, S_{V_1}, \ldots, S_{V_k}$ 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. As for S_{V_0}, it is the ratio of sets that have received k or more lines.

The two most common access patterns found in the kind of codes we intend to model are the sequential access and the access described as "access to n groups of t elements separated by a constant stride d". The representation and calculation of the impact on the cache of these and other access patterns by means of area vectors has been solved in [3].

3.1 Area Vectors Addition

It is very common that references to more than one data structure take place between two accesses to the same line of a data structure. This implies that a mechanism is needed to add the area vectors associated with these structures in order to calculate the global area vector.

Given two area vectors $S_U = (S_{U_0}, S_{U_1}, \ldots, S_{U_k})$ and $S_V = (S_{V_0}, S_{V_1}, \ldots, S_{V_k})$, the addition of them, $S_U \bigcup S_V$, is defined as

$$\begin{aligned} (S_U \cup S_V)_0 &= \textstyle\sum_{j=0}^{K} \left(S_{U_j} \sum_{i=0}^{K-j} S_{V_i} \right) \\ (S_U \cup S_V)_i &= \textstyle\sum_{j=i}^{K} S_{U_j} S_{V_{(K+i-j)}} \qquad 0 < i \leq K \ . \end{aligned} \tag{1}$$

This method is based in the addition of independent probabilities, which means that it does not take into account the relative positions of the data structures in memory. If such positions are known, the overlapping coefficients of the footprints associated with the accesses to these structures on the cache can be estimated. The accuracy of the addition may be improved by using them to scale or weight the area vectors [3]. This way, area vectors corresponding to data structures that overlap more in the cache with the data structure affected by the reference that is being analyzed, get higher weights than those ones that overlap less or even do not overlap. The latter case would in fact turn the corresponding area vector S into an empty one ($S_i = 0, 0 \leq i < k$ and $S_k = 1$).

4 Probabilistic Miss Equations

Our method generates a *Probabilistic Miss Equation* (PME) for each reference in each nesting level. Let $F_i(R, S(\text{RegInput}), p)$ be the PME that estimates the number of misses generated by reference R in nesting level i. It is a function of $S(\text{RegInput})$, the area vector associated to the region that has been accessed since the last access to a given line of the data structure that R references. If the reference is inside a conditional sentence whose condition follows an uniform distribution, p is the probability that the condition is true. The probability of the conditionals can be obtained either by several means : profiling, input data analysis, or previous knowledge of the application field.

The loops are examined from the innermost one to the outermost one in order to calculate the number of misses generated by each reference. In each level a formula is generated depending on whether the variable associated to the current loop indexes or not any of the references found in the condition(s) of the conditional sentence. If the loop variable is not used in the indexes of any of these variables, then a *Condition Independent Reference Formula* (CIRF) is applied. Otherwise, a *Condition Dependent Reference Formula* (CDRF) is built.

4.1 Condition Independent Reference Formulas

This kind of formulas has already been described in [3]. It assumes that, if the analyzed reference reuses a given line in the current loop, the last access to that line took place in the previous iteration of the considered loop. The reuse in the loop may take place either because of temporal reuse (the loop variable does not index the reference) or spatial reuse (the loop variable indexes the reference and its stride is smaller than the line size). Let N_i be the number of iterations in the loop of the nesting level i, and L_{Ri} be the number of iterations in which there is no possible reuse for the lines referenced by R, then we can define $F_i(R, S(\text{RegInput}), p)$ as

$$\begin{aligned} F_i(R, S(\text{RegInput}), p) =& L_{Ri} F_{i+1}(R, S(\text{RegInput}), p) + \\ & (N_i - L_{Ri}) F_{i+1}(R, S(\text{Reg}(A, i, 1)), p) \ , \end{aligned} \tag{2}$$

where $\text{Reg}(A, i, j)$ stands for the memory region accessed during j iterations of the loop in the nesting level i that can interfere with data structure A. $S(\text{Reg}(A, i, j))$ represents the area vector associated to that region.

The formula reflects the fact that for the L_{Ri} iterations in which there can be no reuse in this loop, the miss probability depends on the accesses and reference patterns in the outer loops. In the remaining iterations, this probability is calculated as a function of the accessed regions during the portion of program executed between those reuses, this is, during one iteration of loop i.

The indexes of the reference R are affine functions of the variables of the loops that enclose it. As a result, R follows 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,

$$L_{Ri} = 1 + \left\lfloor \frac{N_i - 1}{max\{L_s/S_{Ri}, 1\}} \right\rfloor \ . \tag{3}$$

The formula calculates the number of accesses of R that can not 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}.

4.2 Condition Dependent Reference Formulas

The second kind of formulas is applied when $\mathtt{I}_i$, the variable associated to the current loop, is used in the indexes of the references found in the condition of a conditional sentence that controls the execution of the reference R whose behavior we are analyzing. In this case, the last access of R to a given line may have happened an indeterminate number of iterations ago, depending on the probability p that the condition is fulfilled and thus R is executed.

Weighted Reuse When a reference is located inside a data-dependent conditional sentence whose outcome changes for the different iterations of a given loop, it is not possible to estimate accurately the number of iterations of the loop between two accesses to the same line by the reference. The reason is that accesses only take place with a given probability. Thus, a probabilistic approach must be followed to estimate this value, which is the reuse distance in the loop. This way, the probability that the last access has happened 1,2... iterations ago must be weighted. We define the weighted reuse for the j-th consecutive access to a given line during the execution of the loop in nesting level i, $WR(p_i, \text{RegInput}, i, j, p)$ with this purpose. In this expression, p_i stands for the probability the line is accessed by the considered reference during one iteration of the loop, and RegInput, stands for the region accessed since the last reference to the line when the loop execution begins, just as in the previous formulas. The weighted reuse is calculated as

$$\begin{aligned} WR(p_i, S(\text{RegInput}), i, j, p) =& (1-p_i)^{j-1} F_{i+1}(R, S(\text{RegInput}) \cup S(\text{Reg}(A, i, j-1)), p) + \\ & \sum_{k=1}^{j-1} p_i (1-p_i)^{k-1} F_{i+1}(R, S(\text{Reg}(A, i, k-1)), p) \ . \end{aligned} \tag{4}$$

The first term 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 $j-1$ previous iterations of the considered loop.

Given a loop with n iterations, we define the total weighted reuse in its n iterations, $TWR(p_i, S(\text{RegInput}), i, n, p)$, as[1] the addition of the weighted reuse for every one of them:

$$TWR(p_i, S(\text{RegInput}), i, n, p) = \sum_{j=1}^{n} WR(p_i, S(\text{RegInput}), i, j, p) \ . \tag{5}$$

[1] If n is not an integer value, it is estimated as $TWR(p_i, S(\text{RegInput}), i, n, p) = (n - \lfloor n \rfloor) TWR(p_i, S(\text{RegInput}), i, \lceil n \rceil, p) + (1 - (n - \lfloor n \rfloor)) TWR(p_i, S(\text{RegInput}), i, \lfloor n \rfloor, p)$

Line Access Probability The fact that every access takes place only with probability p complicates the calculation of the probability that a given line is accessed during each iteration of the considered loop. This probability depends not only on the access pattern to the line in this nesting level, but also in the inner ones. This way, access probabilities are calculated starting in the innermost loop and analyzing the nest outwards, just as the PMEs.

In the CIRF formula we had defined L_{Ri} as the number of loop iterations where there is no possible reuse. Now we define G_{Ri} as the number of iterations that can potentially reuse the lines accessed in those L_{Ri} iterations. The product of both terms must be equal to the number of iterations of the loop, thus $G_{Ri} = N_i/L_{Ri}$. We represent the probability that a line is accessed during one iteration of the loop in nesting level i as p_i. If the loop variable for the level $i+1$ is not used in the indexes of the references found in the condition, then $p_i = p_{i+1}$. Otherwise, $p_i = 1 - (1 - p_{i+1})^{G_{Ri+1}}$. In the innermost loop $p_i = p$.

Formulation Once the previous concepts have been established, the final formula that estimates the number of misses of a conditional dependent reference R (CDRF) in nesting level i is,

$$F_i(R, S(\text{RegInput}), p) = L_{Ri} TWR(p_i, S(\text{RegInput}), i, G_{Ri}, p) \ . \tag{6}$$

4.3 Calculation of the Number of Misses

In the innermost level that contains the reference R, $F_{i+1}(R, S(\text{RegInput}), p)$, the number of misses caused by the reference in the immediately inner level is $S_0(\text{RegInput})$, this is, the first element in the area vector associated to the region RegInput. If the reference is inside a conditional sentence, this value is multiplied by p, as the reference only happens with probability p.

Once the formulas for the outermost level are calculated, the number of misses is estimated as $F_0(R, S(\text{RegInput}_{\text{total}}), p)$, where $\text{RegInput}_{\text{total}}$ is the total region, this is, the region that covers the whole cache. The miss probability associated with this region is one.

5 Model Validation

We have validated our model by applying it manually to the simple codes shown in Figs. 2 and 3. They are, respectively, a synthetic kernel and an optimized matrix product. These codes consist of a nest of loops that contain references inside a conditional sentence. We are using FORTRAN in the examples we model, but there is no problem in modeling codes with other languages. The analytical model only depends on the access patterns, not on the language that generates them.

A tool to apply automatically our modeling strategy is currently under construction.

```
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. 2. Synthetic kernel code

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

Fig. 3. Optimized matrix product)

5.1 Synthetic Kernel Modeling

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. 2 contains three references to memory. If the compiler followed a different policy to generate the code, we would just model the access pattern generated by the references it produces. The model in [3] can estimate the behavior of the references `A(I)` and `B(J)`, which take place in every iteration of their enclosing loops. Notice that the second access to `B(J)` would reuse the value which was previously loaded in order to check the condition found in the code. This way, `C(J)` is the only access to memory that takes place under the control of a conditional, which has an uniform probability p of being fulfilled, and thus we will focus our explanation on the modeling of its behavior.

The modeling begins in the innermost loop, in level 1. This loop variable indexes the reference involved in the condition, so the CDRF is to be used. Let $S_{R1} = 1$, $L_{R1} = 1 + \lfloor (N-1)/L_s \rfloor$, $G_{R1} \simeq L_s$, $p_1 = p$,then we obtain the following formula,

$$F_1(R, S(\text{RegInput}), p) = (1 + \lfloor (N-1)/L_s \rfloor)\, TWR(p, S(\text{RegInput}), 1, L_s, p)\ . \tag{7}$$

As this loop is in the innermost level, $F_2(R, \mathrm{RegInput}, p) = pS_0(\mathrm{RegInput})$. The calculation of TWR (5) from WR (4) requires to estimate the memory regions accessed during i iterations of this loop that may generate interference with `C`, the data structure affected by the reference we are analyzing:

$$S(\mathrm{Reg}(C, 1, i)) = S_s(i) \cup S_{\mathrm{sauto}}(i)\ . \tag{8}$$

The first term corresponds to the sequential access to i consecutive elements of `B`, and the second term stands for the autointerference produced by the access to i consecutive elements of `C`. The autointerference is the interference that the accesses to a given data structure may generate on other accesses to that same structure. It is calculated in a slightly different way to that of cross interferences, which are the interferences due to the accesses to other data structures. The reason is that accesses to a given line do not generate interferences on that very same line, but they can of course generate interference with other lines of the same data structure.

In the next outer level, level 0, the loop index does not index the reference used in the conditional, thus the CIRF is applied. Its formulation for $L_{R0} = 1$ is,

$$\begin{aligned} F_0(R, S(\mathrm{RegInput}), p) = F_1(R, S(\mathrm{RegInput}), p) +& \\ (M-1)\, F_1(R, S(\mathrm{Reg}(C, 0, 1)), p)&\ . \end{aligned} \tag{9}$$

In this case $\mathrm{Reg}(C, 0, 1)$, the region accessed during one iteration of the in loop level 0 that may affect data structure `C` in the cache is

$$S(\mathrm{Reg}(C, 0, 1)) = S_s(1) \cup S_s(N) \cup S_{s_{\mathrm{auto}}}(N)\ . \tag{10}$$

The first term is associated to one element in `A` and the second one stands for the access to N consecutive element of `B`. Finally, the third term corresponds to the autointerference produced by the access to N consecutive elements of `C`.

As we have reached the outermost level, the number of misses generated by the reference may be estimated as $F_0(R, S(\mathrm{RegInput}_{\mathrm{total}}), p)$, where $\mathrm{RegInput}_{\mathrm{total}}$ is the region that covers all the cache and so $S_0(\mathrm{RegInput}_{\mathrm{total}}) = 1$.

5.2 Optimized Product Modeling

The second code used in the validation is shown in Fig. 3. This kernel multiplies a matrix with a uniform distribution of zero entries by another matrix B. As an optimization, when the element of `A` to be used in the current product is 0, the operation is not performed. This way two arithmetic operations and one data load are avoided.

This code comprises three different references. Considering the assumptions described in the previous example, the behavior of references `C(I,J)` and `A(I,K)` could be modeled following [3]. Thus we will devote our explanation to the analysis of `B(K,J)`.

In the innermost level, level 2, the loop variable indexes the reference of the condition, so the CDRF formula must be applied. As $S_{R2} = 1$, $L_{R2} = 1 + \lfloor (N-1)/L_s \rfloor$, $G_{R2} \simeq L_s$ and $p_2 = p$, then the formulation is

$$F_2(R, S(\text{RegInput}), p) = (1 + \lfloor (N-1)/L_s \rfloor)\, TWR(p, S(\text{RegInput}), 2, L_s, p) \ . \tag{11}$$

This loop is in the innermost level. Thus, $F_3(R, \text{RegInput}, p) = pS_0(\text{RegInput})$. In this case the calculation of WR (4) requires

$$S(\text{Reg}(B,2,i)) = S_{l_{\text{auto}}}(i, p_{\text{line}}) \cup S_r(i,1,M) \ . \tag{12}$$

The first term represents the autointerference of `B`, which is due to the access to i consecutive elements with a uniform probability of access per cache line of `B` of p_{line}. The second term corresponds to the access to i elements of `A` that belong to different columns, each column having a size of M elements. In general, $S_r(g,s,d)$ calculates the area vector associated to the access to g groups of size s separated by d elements.

In the next level, level 1, the loop variable indexes the reference in the condition, so the CIRF formula is to be applied. Let $L_{R1} = P$, the formulation is

$$F_1(R, S(\text{RegInput}), p) = PF_2(R, S(\text{RegInput}), p) \ . \tag{13}$$

Also $p_1 = 1 - (1-p)^{L_s}$. 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 $S_{R0} = 0$, $L_{R0} = 1$, $G_{R0} = M$ and $p_0 = p_1$, the formulation is

$$F_0(R, S(\text{RegInput}), p) = TWR(p_0, S(\text{RegInput}), 0, M, p) \ . \tag{14}$$

We need to know the value of the accessed regions $\text{Reg}(B,0,i)$ to compute WR:

$$S(\text{Reg}(B,0,i)) = Sl_{\text{auto}}(P*N, p_{\text{line}}) \cup S_r(N,i,M) \cup S_r(P,i,M) \ . \tag{15}$$

The first term is associated to the autointerference of `B`, which is the access to $P*N$ consecutive elements with an uniform probability of access to each line of p_{line}. The second term represents the access to i consecutive elements from each one of the N columns of matrix `A`, which have a size of M elements each. The third term represents the access to i consecutive elements from each one of the P columns in `C`, which also have size M.

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}	Δ_{NM}	σ	$T_{simulation}$	$T_{execution}$	$T_{modeling}$
50000	47500	0.2	65536	16	2	0.372	5.067	5.515	141	60	0.005
50000	47500	0.6	65536	16	8	0.001	0.021	0	262	74	0.004
50000	47500	0.2	8192	32	4	0.004	0.094	0	138	50	0.005
50000	47500	0.4	16384	8	2	0.015	0.086	0	182	68	0.005
50000	47500	0.8	16384	8	2	0.001	0.012	0	255	67	0.004
22000	14500	0.4	32768	16	4	0.001	7.010	7.375	28	7	0.003
22000	14500	0.2	16384	8	4	0.239	1.260	0.144	21	6	0.005
22000	14500	0.9	16384	8	16	0.005	0.041	0	50	7	0.003
22000	14500	0.4	8192	8	1	0.067	0.381	0	65	7	0.004
22000	14500	0.4	8192	32	2	0.007	0.165	0	22	8	0.004
22000	14500	0.7	8192	32	8	0.007	0.206	0	31	7	0.004
18000	22000	0.2	32768	16	2	0.574	8.051	8.326	23	7	0.004
18000	22000	0.6	32768	16	4	0.341	4.489	3.963	40	10	0.005
18000	22000	0.1	16384	8	2	0.076	0.431	0.383	22	6	0.004
18000	22000	0.8	16384	8	8	0	0	0	52	8	0.004
18000	22000	0.3	4096	32	4	0.141	0.417	0	95	8	0.004
14500	19500	0.7	65536	8	8	0	0.032	0	32	7	0.005
14500	19500	0.2	16384	4	2	0.252	0.790	0.766	20	5	0.005
14500	19500	0.3	8192	4	1	0.124	0.366	0	20	6	0.004
14500	19500	0.8	8192	4	4	0.009	0.032	0	43	6	0.004
1750	1750	0.4	8192	4	8	0	0.108	0	1	1	0.003
1750	1750	0.7	8192	8	4	0	0.230	0	0	0	0.003
950	1150	0.4	1024	4	8	0.349	1.046	0	0	0	0.001
950	1150	0.2	4096	8	16	0	0.389	0	0	0	0.001
850	1200	0.6	1024	16	4	0	0	0	0	0	0

5.3 Validation Results

Our validation is based on the comparison of the predictions of the model with the results of trace-driven simulations. Different cache configurations, problem sizes and probabilities for the conditionals were used in our experiments.

Table 1 and Table 2 show the validations results for the codes in Figs. 2 and 3, respectively. The first three columns contain the problem size as well as the probability p that the condition is fulfilled. Then the cache configuration is given 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 elements of the arrays used in the codes.

Two metrics have been used in order to study the accuracy of the model. One of them, Δ_{MR}, is based on the miss rate (MR). It stands for the absolute value of the difference between the predicted and the measured miss rate. We also use Δ_{NM}, which expresses the error in the prediction of the number of misses as a percentage of the number of misses measured by the trace-driven simulation. The tables also include σ, the typical deviation of the number of misses measured

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	C_s	L_s	K	Δ_{MR}	Δ_{NM}	σ	$T_{simulation}$	$T_{execution}$	$T_{modeling}$
1700	1600	1250	0.2	32768	16	2	0.010	0.039	0.037	331	162	0.035
1700	1600	1250	0.4	16384	32	2	0	0	0	372	197	0.041
1700	1600	1250	0.6	16384	32	16	0	0	0	1047	210	0.214
1700	1600	1250	0.2	8192	8	4	0.017	0.018	0	342	162	0.023
1700	1600	1250	0.8	8192	8	4	0	0	0	715	199	0.023
1000	850	900	0.3	8192	8	4	0.007	0.038	0.019	83	40	0.016
1000	850	900	0.8	4096	4	8	0.033	0.047	0	206	45	0.017
1000	850	900	0.2	4096	4	1	0.068	0.085	0.015	64	36	0.013
1000	850	900	0.3	4096	8	1	0.054	0.074	0.004	70	40	0.013
1000	850	900	0.7	4096	8	2	0.017	0.026	0	122	46	0.014
900	850	900	0.1	65536	8	1	0.065	0.525	0.006	48	29	0.020
900	850	900	0.9	65536	8	8	0.015	0.233	0	107	38	0.024
900	850	900	0.2	16384	32	2	0.055	0.064	0	63	32	0.026
900	850	900	0.8	16384	32	2	0.036	0.064	0	104	39	0.024
750	750	1000	0.4	32768	4	2	0.040	0.260	0	56	31	0.019
750	750	1000	0.2	16384	8	4	0.114	0.633	0.568	57	26	0.016
750	750	1000	0.4	8192	16	1	0.147	0.210	0.005	56	32	0.020
750	750	1000	0.8	8192	16	16	0.064	0.109	0	180	32	0.054
200	250	150	0.8	16384	4	2	0.114	0.810	0.020	1	0	0
200	250	150	0.3	4096	4	8	0.113	0.759	0	1	0	0
200	250	150	0.8	2048	4	2	0.348	1.257	0.468	1	0	0
200	250	150	0.1	1024	8	4	0.139	0.143	0	1	0	0.01
100	350	90	0.8	4096	4	8	0.077	0.547	0	1	0	0
100	350	90	0.8	1024	4	8	0.077	0.111	0	1	0	0.01
100	350	90	0.4	2048	8	4	0.141	0.176	0	0	0	0.01

expressed as a percentage of the average number of misses measured. For every combination of a cache configuration and a data input, 25 different simulations have been made, using different base addresses for the data structures in each of those simulations. The usage of the overlapping coefficients helps adapt the model prediction to the variability of the cache behavior that is due to the different relative positions of the data structures.

The model provides a good estimation of the caches behavior, as the tables show. The prediction error is smaller or almost equal than the typical deviation introduced by the own variability of the number of misses of the code. We can observe that when the cache works well, this is, when it is large enough to hold the program data structures, the typical deviation is much greater and so the error in the prediction is greater too, although it is smaller or similar than the typical deviation.

Although the model only considers one cache at a time, it is straightforward to predict the behavior of a whole memory hierachy using it. The model can be applied separately to each level of the hierarchy and then combine these partial results to obtain a good prediction of the behavior of the whole memory system. Some of our experiments in [3] prove this.

The three last columns in Tables 1 and 2 show the simulation, source code execution, and modeling times (in seconds) measured in a 800 MHz Pentium III system for our two example codes, respectively. As we see, modeling times are several orders of magnitude shorter than trace-driven simulation and even execution times. The modelling 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 [3], the overhead of such tool is negligible.

6 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 [7], which is restricted to the modeling of direct-mapped caches and that lacks an automatic implementation. Later, [8] and [9] overcame some of these limitation. Cache Miss Equations (CMEs) are constructed in [8], which are lineal systems of Diophantine equations, where each solution corresponds to a potential miss cache. One of its main limitations is its high computional cost. The computing times required by [9] are much shorter, and similar to those ones of our model, however, the error is larger than that of our model. Both models in [8] and [9] share the limitation that they are only suitable for 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 [10], which implies that optimizations should consider several nests.

More recently, [4] and [5] allow the analysis of not 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, in fact, running a simulation is much faster than solving the equations this model generates, which reduces its applicability. As for the latter, it is based on the extension of [11] in order to quantify the reuse, and it applies the CMEs of [8] 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 input 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.

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 (physicallly-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.

7 Conclusions and Future Work

An extension to the model in [3] has been presented which allows the analysis of codes with data dependent conditional sentences whose conditions follow an uniform distribution. No other model in the bibliography can estimate the cache behavior of this kind of codes. A validation using simple codes has proved the model estimation to be very accurate despite the very short time required to compute it.

Our model is very suitable to guide the optimization process of a compiler and to help programmers understand the behavior of their codes. In the field of embedded systems the study of the behavior of the memory hierarchy is not only relevant in order to reduce the execution times and the energy and power consumption, but also to calculate the WCET (*Worst Case Execution Time*). Studying the application of our model to this latter usage is part of our future work.

We are currently working in an automatic implementation that applies our model automatically and transparently to the programmer on a great variety of codes. We plan to use the Polaris [12] compiler framework as platform for this purpose, although the model can be coupled with any other front-end and used to model codes written in any programming language. We believe that the probabilistic model proposed here is suitable for the modeling of other kinds of codes, like those that contain irregular access patterns due to the use of indirections and pointers. These codes have been largely ignored in all the previous bibliography despite being common in scientific and engineering applications. Some problems related to the modeling of this kind of codes that we anticipate include getting the distribution of the accesses, mapping the real distribution to one that can be modeled. The more irregular the distribution is, the bigger the mathematical complexity of the associated model so we should try to minimize the corresponding modeling time. All of these problems are to be solved first manually. Then a way to characterize each step and develop a method to apply it automatically is to be found.

References

1. Uhlig, R., Mudge, T.: Trace-Driven Memory Simulation: A Survey. ACM Computing Surveys **29** (1997) 128–170
2. Ammons, G., Ball, T., Larus, J.R.: Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In: SIGPLAN Conference on Programming Language Design and Implementation. (1997) 85–96

3. Fraguela, B.B., Doallo, R., Zapata, E.L.: Probabilistic Miss Equations: Evaluating Memory Hierarchy Performance. IEEE Transactions on Computers **52** (2003) 321–336
4. 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
5. 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
6. Gannon, D., Jalby, W., Gallivan, K.: Strategies for Cache and Local Memory Management by Global Program Transformation. Journal of Parallel and Distributed Computing **5** (1988) 587–616
7. Temam, O., Fricker, C., Jalby, W.: Cache Interference Phenomena. In: Proc. Sigmetrics Conference on Measurement and Modeling of Computer Systems, ACM Press (1994) 261–271
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. Harper, J.S., Kerbyson, D.J., Nudd, G.R.: Analytical Modeling of Set-Associative Cache Behavior. IEEE Transactions on Computers **48** (1999) 1009–1024
10. McKinley, K.S., Temam, O.: Quantifying Loop Nest Locality Using SPEC'95 and the Perfect Benchmarks. ACM Transactions on Computer Systems **17** (1999) 288–336
11. Wolf, M.E., Lam, M.S.: A Data Locality Optimizing Algorithm. In: Proc. of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation. (1991) 30–44
12. 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