Set Associative Cache Behavior Optimization *

Ramón Doallo[1] and Basilio B. Fraguela[1] and Emilio L. Zapata[2]

[1] Dept. de Electrónica e Sistemas. Univ. da Coruña
Address: Facultade de Informática, Campus de Elviña, s/n
15071 A Coruña, SPAIN
{doallo,basilio}@udc.es
[2] Dept. de Arquitectura de Computadores. Univ. de Málaga
Address: Complejo Tecnológico, Campus de Teatinos. PO Box 4114
E-29080 Málaga, SPAIN
ezapata@ac.uma.es

Abstract. One of the most important issues related to program performance is the memory hierarchy behavior. Programmers try nowadays to optimize this behavior intuitively or using costly techniques such as trace-driven simulations through a trial and error process. A systematic modeling strategy that allows an automated analysis of the memory hierarchy performance is developed in this work. This approach, besides requiring much shorter computation times, can be integrated in a compiler or an optimizing environment. The models consider caches of an arbitrary size, line size and associativity, and as we will show, they have proved a good degree of accuracy and have a wide range of applications. Loop interchange, loop fusion and optimal block size selection are the techniques whose successful application has been driven by the models in this work.

1 Introduction

The increasing gap between processor and main memory cycle times makes more critical the memory hierarchy performance, in which caches play an essential role. The traditional approaches to study and try to improve this performance are based on trial and error processes which give little information about the origins of this behavior, and do not allow to understand the way different program transformations influence it. For example, the most widely used approaches, trace-driven simulations [10] and the use of the hardware built-in counters that some microprocessor architectures implement [11], only provide the number of misses generated. Besides, both approaches require large computation times, especially simulation. On the other hand, built-in counters are limited to the study of the architectures where these devices exist.

General techniques for analyzing cache performance are required in order to propose improvements to the cache configuration or the code structure. A

* This work was supported by the Ministry of Education and Science (CICYT) of Spain under project TIC96-1125-C03

better approach is that of analytical models that extract some of their input parameters from address traces [1], [6], although they still need the generation of the trace each time the program is changed. Finally, there are a few analytical models based directly on the code [9], [4], and most of them are oriented to direct mapped caches. The last one has been implemented in an optimizing environment and later extended to set associative caches in [5]. It is based on the construction of *Cache Miss Equations* (CMEs) suitable for regular access patterns in isolated perfectly nested loops. It does not take into account the probability of hit in the reuse of data structures referenced in previous loops and seems to have heavy computing requirements.

In this work we introduce a systematic strategy to develop probabilistic analytical cache models for codes with regular access patterns, considering set associative caches with LRU replacement policy. Our approach allows the automated generation of equations that provide the number of misses generated in each memory reference and loop. An additional feature of these models, that previous works such as [5] lack, is that they take into account that portions of data structures accessed in previous loops may be in the cache when they are accessed by another set of nested loops, although the worst case hit probability is used. The automatically developed models have been validated using trace-driven simulations of typical code loops, and have proved to be very accurate in most of the cases. As an added benefit, the computing times of the model are very competitive.

The concepts on which our modeling strategy relies are introduced in the following section. The steps for modeling the accesses in regular nested loops are explained in Section 3, while Section 4 is devoted to both validation and usefulness of our approach in driving optimizations based on standard loop transformations. Conclusions and future work are discussed in the last section.

2 Basic Concepts

We classify the misses on a given data structure in a K-way associative cache in two groups: intrinsic misses, those that take place the first time a memory line is referenced, and interference misses, which take whenever a line has been already accessed but it has been replaced since its previous access. If a LRU replacement policy is applied, this means that K or more different lines mapped to the same cache set have been referenced since the last access to the line. In this way, the first time a memory line is accessed, a miss is generated. For the remaining accesses, the miss probability equals the probability that K or more lines associated to its cache set have been accessed since its previous access.

Our model uses the concept of area vector to handle this probability. Given a data structure `V`, $S_{\mathrm{V}} = S_{\mathrm{V}_0}, S_{\mathrm{V}_1}, \ldots, S_{\mathrm{V}_K}$ is the area vector corresponding to the accesses to `V` during a portion of the program execution. The i-th position of the vector contains the ratio of cache sets that have received $K - i$ lines of the structure. The first position, S_{V_0}, has a different meaning: it is the ratio of sets that have received K or more lines. Different expressions have been developed that estimate the area vectors as a function of the access pattern of the corresponding data structure [2]. The formulae and algorithms associated to the sequential access and the access to a number of regions of consecutive memory words which have a constant distance between the start of two such regions (access with a constant stride) may be found in [3]. A mechanism for adding the area vectors corresponding to the different data structures referenced between two consecutive accesses to the data structure which is being studied has been developed too.

3 Automatic Code Modeling

```
DO $I_Z$=1, $N_Z$, $L_Z$
  ...
  DO $I_1$=1, $N_1$, $L_1$
    DO $I_0$=1, $N_0$, $L_0$
      A($f_{A1}$($I_{A1}$), $f_{A2}$($I_{A2}$), ..., $f_{AdA}$($I_{AdA}$))
      ...
      B($f_{B1}$($I_{B1}$), $f_{B2}$($I_{B2}$), ..., $f_{BdB}$($I_{BdB}$))
      ...
    END DO
    ...
    C($f_{C1}$($I_{C1}$), $f_{C2}$($I_{C2}$), ..., $f_{CdC}$($I_{CdC}$))
    ...
  END DO
  ...
END DO
```

Fig. 1. General perfectly nested DO loops in a dense code.

We will now present a systematic approach to apply automatically this modeling strategy to codes with regular access patterns. Figure 1 shows the type of code to model. A FORTRAN like notation is used, as this is the language we consider. The data structures considered are vectors and matrices whose dimensions are indexed by an affine function of the enclosing loop variables. Any of these variables is not allowed to appear in more than one dimension, so that only the two types of regular accesses discussed in the previous section take place. The functions of the indices are of the general form (affine function):

$$f_{\mathrm{A}x}(\mathrm{I}_{\mathrm{A}x}) = \Delta_{\mathrm{A}x}\mathrm{I}_{\mathrm{A}x} + K_{\mathrm{A}x},\ x = 0, 1, \ldots, \mathrm{d}_{\mathrm{A}} \tag{1}$$

We obviate the $\Delta_{\mathrm{A}x}$ constants in our exposition, as their handling would be analogous to that of the steps L_i of the loops. Their consideration would just require taking into account that the distance between two consecutive points accessed in dimension x is $\Delta_{\mathrm{A}x}S_{\mathrm{A}x}$ instead of $S_{\mathrm{A}x}$, that is what we shall consider.

3.1 Modeling Perfectly Nested Loops

We will explain here in detail only the modeling of perfectly nested loops with one reference per data structure due to space limitations. When several references are found for the same data structure, the procedure is somewhat different in order to take into account the data reuses that these references may generate. Non perfectly nested loops in which only simple blocks are found between the loops of the nest can be also modeled following this strategy.

Miss equations First, the equations $F_i(R,p)$ are built. They provide the number of misses on reference R at nesting level i considering a miss probability p in the first access to a line of the referenced matrix. These equations are calculated examining the loops from the inner one containing R to the outer one, in such a way that $F_i(R,p)$ depends on $F_{i-1}(R,p)$ and the value of p for level $i-1$ is really calculated during the generation of the the formula for level i. The following rules are applied in each level:

1. When the loop variable is one of the used in the indices of the reference, but not the corresponding to the first dimension, the function that provides the number of misses during the complete execution of the loop of level i as a function of probability p is:

$$F_i(R,p) = \frac{N_i}{L_i} F_{i-1}(R,p) \tag{2}$$

   This approach is based on the hypothesis that the first dimension of any matrix is greater or equal to the cache line size. This could not hold for caches with large line sizes and matrices with small dimensions, but the conjunction of both factors gives place to very small miss rates in which the error introduced is small and little advantage can be obtained from the application of the automated model.
2. If the loop variable is not any of those used in the indices of the reference, this is a reuse loop for the reference we are studying. In this case the number of misses in this level is estimated as:

$$F_i(R,p) = F_{i-1}(R,p) + \left(\frac{N_i}{L_i} - 1\right) F_{i-1}(R, S_0(\mathtt{A}, i, 1)) \tag{3}$$

   where $S(\mathtt{Matrix}, i, n)$ is the interference area vector that stands for the lines that may cause interferences with any of the lines of matrix `Matrix` after n iterations of the loop in level i. Function (3) expresses the fact that the first iteration of the loop does not influence the number of misses on matrix `A`, being this value determined by the probability p, which is calculated externally. Nevertheless, in the following iterations the same regions of the matrix are accessed, which means that the interference area vector in the first accesses to each line in this region is composed by the whole set of elements accessed during one iteration of the reuse loop.

3. If the loop variable is the one used in the indexing of the first dimension and $L_i \geq L_\mathrm{s}$, where L_s is the line size, we proceed as in case 1, as each reference will take place on a different line. Otherwise, we have:

$$F_i(R,p) = \frac{N_i}{L_\mathrm{s}} F_{i-1}(R,p) + \left(\frac{N_i}{L_i} - \frac{N_i}{L_\mathrm{s}}\right) F_{i-1}(R, S_0(\mathtt{A}, i, 1)) \qquad (4)$$

   The miss probability in the first access to each referenced line is given by the probability p, externally calculated. The remaining accesses take place on lines referenced in the previous iteration of the loop of level i, so the interference area vector is the corresponding to one iteration of this loop.

In the first level containing the reference $F_{i-1}(R,p)$, the number of misses caused by this reference in the closer lower level, is considered to be p. Once calculated the formula for the outer loop, the number of misses is calculated as $F_z(R,1)$ (see Figure 1), which assumes that there are no portions of the data structure in the cache when the code execution begins.

Interference area vectors calculation The calculation of the interference area vectors $S(\mathtt{Matrix}, i, n)$ is performed in an automated way by analyzing the references in such a way that:

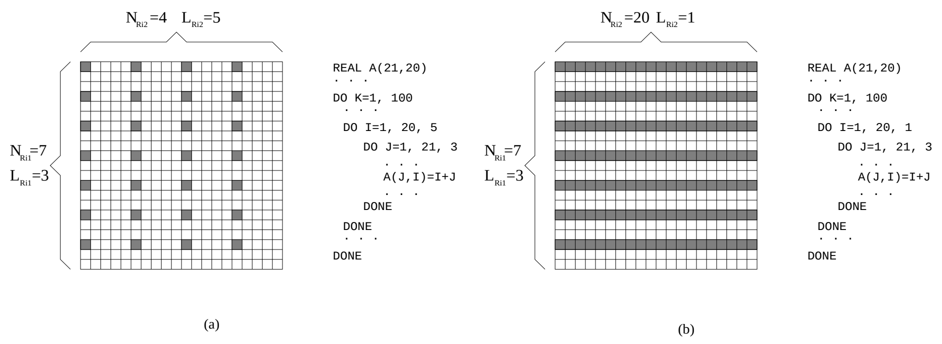
– If the variable used in the indexing of one dimension $\mathtt{I}_h$, is such that $h < i$, then we assign to this dimension a set of N_h/L_h points with a constant distance of L_h points between each two of them.
– On the other hand, if $h > i$, only one point in this dimension is assigned.
– Finally, if $h = i$, n points with a distance L_i are assigned to the dimension.

There is one exception to this rule. It takes place when the variable associated to the considered dimension $\mathtt{I}_h$ belongs to a loop that is a reuse loop for the reference whose number of misses is to be estimated, and there are no non reuse loops for that reference between levels i (not included) and h. In that case only one point of this dimension is considered.

After this analysis, the area of matrix A affected by reference R during an iteration of loop i would consist of $N_{\mathrm{R}i1}$ regions of one word in the first dimension, with a constant stride between each two regions of $L_{\mathrm{R}i1}$ words. In the second dimension, $N_{\mathrm{R}i2}$ elements would have been accessed, with a constant stride between each two consecutive ones of $L_{\mathrm{R}i2}$ groups of $\mathrm{d}_{\mathrm{A}1}$ words (size of the first dimension), and so on. This area could be represented as:

$$\mathcal{R}_{\mathrm{R}i} = ((N_{\mathrm{R}i1}, L_{\mathrm{R}i1}), (N_{\mathrm{R}i2}, L_{\mathrm{R}i2}), \ldots, (N_{\mathrm{R}i\mathrm{d}_\mathrm{A}}, L_{\mathrm{R}i\mathrm{d}_\mathrm{A}})) \qquad (5)$$

Figure 2 depicts this idea for a bidimensional matrix. This area will typically have the shape of a sequential access or an access to groups of consecutive elements separated by a constant stride, which is what happens in case (b) of the figure (the stride in case (a) is not constant). Both accesses have already been modeled for the calculation of their corresponding cross or auto-interference area vectors (see [3]).



```
REAL A(21,20)
. . .
DO K=1, 100
 . . .
  DO I=1, 20, 5
     DO J=1, 21, 3
        . . .
        A(J,I)=I+J
        . . .
     DONE
  DONE
  . . .
DONE
```

```
REAL A(21,20)
. . .
DO K=1, 100
 . . .
  DO I=1, 20, 1
     DO J=1, 21, 3
        . . .
        A(J,I)=I+J
        . . .
     DONE
  DONE
  . . .
DONE
```

Fig. 2. Areas accessed with strides 5 and 1 for the columns in a bidimensional matrix `A` during one iteration of the `K` loop, in the i-th nesting level.

3.2 Imperfectly Nested Loops and Data Reuse

Real applications consist of many sets of loops which may have in each level several other loops, this is, sets of imperfectly nested loops. The data structures are accessed in several of these sets, giving place to a probability of hit in the reuse of their data.

Our model is based on the generation of a miss equation for each reference and loop and the calculation of the miss probability in the access to a line of the considered matrix inside that loop. As we have seen in Section 3.1, when considering an isolated set of perfectly nested loops, the probability of miss in the first access to a line of a data structure is 1. The probability of hit due to accesses in previously completed loops must be considered when imperfectly nested loops are modeled. Our automatic analyzer uses a pessimistic approach to estimate this probability in order to optimize modeling times. The whole regions of the vectors and matrices accessed during the execution of the outer non-reuse loop including a reference to the studied data structure are used to compute the interference area vectors. More accurate approaches have been already developed [2].

4 Validation and Applications

A simple automatic analyzer based in our modeling strategy has been built in order to validate it. Function calls are used to describe the loops, data structures and references to the analyzer, just as the semantic stage of a code analyzer could do. Its predictions are compared with the misses measured using a locally developed simulator whose validity has been checked using dineroIII, a trace-driven cache simulator belonging to the WARTS toolset [7]. We have applied both tools to a series of codes where different program transformations may be applied, being our purpose to validate the automatically developed models and check if they choose the right approach in the different codes. Attention is also paid to the times required to make the choices.

```
DO I=0, N-1
  DO J=0, N-1
    C(I,J)=C(I,J)+A(I,J)*B(I,J)
  ENDDO
ENDDO

DO I=0, N-1
  DO J=0, N-1
    B(I,J)=B(I,J)+A(I,J)
  ENDDO
ENDDO
```

(a)

```
DO I=0, N-1
  DO J=0, N-1
    C(I,J)=C(I,J)+A(I,J)*B(I,J)
    B(I,J)=B(I,J)+A(I,J)
  ENDDO
ENDDO
```

(b)

Fig. 3. Separate (a) and fused (b) loops

4.1 Loop Fusion

Let us consider the codes in Figure 3, where (b) results from the fusion of the two loops in (a). Both the model and the trace-driven simulations recommend the fusion, as Table 1 shows, which also includes the simulation and modeling times for the separate code ((a) in Figure 3). In this table and the following ones, C_s stands for the cache size. In this case, although the model always succeeds in advising the best strategy, important deviations may be observed in the predictions for some combinations of the input parameters. This is due to two facts. On the one hand, the model follows a pessimistic approach when estimating the miss probability in the first access to a line of a vector that has been accessed in a previous loop, as it estimates it as the worst-case probability (the one associated to the full execution of the previous loop). On the other hand, most of the accesses in this code are row-wise, which makes very important the relative positions of the accesses matrices, and the model uses them only to calculate overlapping coefficients, which do not reflect properly their relation with the miss probability.

Table 1. Validation and time data for the codes in Figure 3, where loop fusion is applied

N	C_s	L_s	K	Measured misses (a)	Trace-driven simulation time (a)	Predicted misses (a)	Modeling time (a)	Measured misses (b)	Predicted misses (b)
400	8	8	1	302	0.335	473	0.005	262	324
400	32	8	1	100	0.266	205	0.006	60	138
400	32	8	2	100	0.286	116	0.007	60	76
400	128	8	2	100	0.275	101	0.017	60	61
500	512	16	2	66	0.475	76	0.059	48	48

```
DO J=0, N-1
  DO I=0, N-1
    R=0.0
    DO K=0, N-1
      R=R+A(I,K)*B(K,J)
    ENDDO
    D(I,J)=R
  ENDDO
ENDDO
```

Fig. 4. Dense matrix-dense matrix product with JIK nesting.

```
DO J2=1, N, BJ
  DO K2=1, N, BK
    DO I=1, N
      DO K=K2, K2+BK
        RA=A(I,K)
        DO J=J2, J2+BJ
          D(I,J) = D(I,J) + B(K,J) * RA
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Fig. 5. Dense matrix-dense matrix product with blocking

4.2 Loop Interchange

Let us consider the dense matrix-dense matrix product in which we use variables `I`, `J` and `K` to index the three dimensions of the problem. There are 9 versions of this algorithm depending on the order in which the loops are nested. As an example, Figure 4 shows the JIK form, where `J` is the variable for the outer loop and `K` is the one used in the inner loop. The model can be used to choose the optimal form much faster than any other technique.

In this example we have tried the IJK, JIK, JKI and KJI forms, those in which more sequential accesses take place. The results for some combinations of the input parameters are shown in Table 2, where trace-driven simulation results are compared with those of our model. In the following, misses are in thousands and times in seconds. Results have been obtained in a SGI Origin 200 with R10000 processors at 180 MHz. In this table and the following ones, N stands for the matrix size, C_{s} for the cache size in Kwords, L_{s} for the line size in words and K for the set size. The table shows both a very good degree of accuracy, which allows the model to always choose the best loop ordering, and very competitive modeling times, even comparing them to execution time (around 0.53 seconds for compilation and 0.45 for execution).

4.3 Optimal Block Sizes

As a final example, let us consider the code in Figure 5, where blocking has been applied to the dense matrix-dense matrix product with IKJ ordering. Both trace-driven simulation and modeling have been used to derive the optimal block sizes for which the dimensions of the block are multiples of 100 for some cache configurations and values of N. This has been done by trying the different block sizes, as the model estimates the number of misses, but not the optimum code parameters. Table 3 shows them in format `BJxBK`, the number of misses they

Table 2. Validation and time data for several forms of the dense matrix-dense matrix product.

Order	N	C_s	L_s	K	Measured misses	Trace-driven simulation time	Predicted misses	Modeling time
IJK	400	32	8	2	8180	36.735	8181	0.004
JIK	400	32	8	2	8040	35.914	8042	0.004
JKI	400	32	8	2	8040	38.050	8040	0.001
KJI	400	32	8	2	8180	36.899	8180	0.004
IJK	400	32	8	1	8988	35.623	9004	0.006
JIK	400	32	8	1	9502	36.771	9559	0.006
JKI	400	32	8	1	8139	37.195	8172	0.001
KJI	400	32	8	1	8279	37.446	8308	0.006
IJK	400	8	8	1	11528	36.499	11473	0.003
JIK	400	8	8	1	14091	38.290	14110	0.003
JKI	400	8	8	1	8638	37.356	8567	0.001
KJI	400	8	8	1	8779	37.566	8693	0.003

Table 3. Real optimal and predicted optimal block sizes with the number of misses they generate in the trace-driven simulation and time required to derive them.

N	C_s	L_s	K	Optimal block	Measured misses	Trace-driven simulation time	Model-based optimal block	Measured misses	Modeling time
400	8	8	1	100x100	19721	396.07	100x100	19721	0.90
400	32	8	1	100x100	5889	328.72	100x100	5889	0.93
400	128	8	1	100x100	784	306.64	100x100	784	0.73
400	128	8	2	100x100	145	304.03	100x100	145	0.52
200	32	8	2	100x100	70	17.48	100x100	70	0.17
200	128	8	2	200x200	15	16.46	100x100	16	0.24
600	512	16	2	300x600	87	1756.86	300x100	93	2.18

generate in the simulation, and the time required to derive them through simulation and modeling. Although the block recommended by the modeling is not always the optimum one, the difference in the number of misses is very small, and the time required by the simulations is dramatically greater than the one for the modeling. Even the time required to obtain the optimum block by measuring the execution time or the values of the built-in counters is greater. For example, for $N = 400$ it took 26.6 seconds of real time, adding compilations and executions, while the model requires always less than one second.

5 Conclusions and Future Work

A probabilistic analytical modeling strategy has been presented that can be integrated in a compiler or optimizing environment. It is applicable to caches of an arbitrary associativity and provides equations for the number of misses each reference generates in each loop of the program. Hit probability in the reuses of data accesses in previous loops is also considered using a simplified approach which allows non perfectly nested loops better analysis. Our automated modeling approach has been implemented in a simple analyzer in order to validate it. This tool has proved to be very fast and to provide good hints on the cache behavior that allow to choose always optimal or almost optimal program transformations, even when important deviations in the number of predicted misses may raise for certain input parameter combinations.

In this work only affine indexing schemes have been studied, but much more ones appear in real scientific problems (indirections, multiple index affine, etc.). In future we plan to extend the automatization of the modeling process to those indexing schemes. Formulae that estimate the area vectors associated to indirections have already introduced in [2].

We also intend to perform the integration of our automated modeling approach in program optimization frameworks that support it, for example by providing the analysis needed to extract the input parameters from the code (*Access Region Descriptors* [8] could be used as a systematic representation for the access patterns).

References

1. Buck, D., Singhal, M.: An analytic study of caching in computer systems. J. of Parallel and Distributed Computing. **32(2)** (1996) 205–214
2. Fraguela, B.B., Doallo, R., Zapata, E.L.: Modeling set associative caches behavior for irregular computations. ACM Performance Evaluation Review (Proc. SIGMETRICS/PERFORMANCE'98) **26(1)** (1998) 192–201
3. Fraguela, B.B.: Analytical Modeling of the Cache Memories Behavior. Ph. D. Thesis. Dept. de Electrónica e Sistemas, Univ. da Coruña (1999) (in Spanish)
4. Ghosh, S., Martonosi, M., Malik, S.: Cache miss equations: An analytical representation of cache misses. Proc. 11th ACM Int'l. Conf. on Supercomputing (ICS'97) (1997) 317–324
5. Ghosh, S., Martonosi, M., Malik, S.: Precise miss analysis for program transformations with caches of arbitrary associativity. ACM SIGPLAN Notices. **33(11)** (1998) 228–239
6. Jacob, B.L., Chen, P.M., Silverman, S.R., Mudge, T.N.: An analytical model for designing memory hierarchies. IEEE Transactions on Computers. **45(10)** (1996) 1180–1194
7. Lebeck, A.R., Wood, D.A.: Cache profiling and the SPEC benchmarks: A case study. IEEE Computer. **27(10)** (1994) 15–26
8. Paek, Y., Hoeflinger, J.P., Padua, D.: Simplification of array access patterns for compiler optimizations. Proc. ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI). (1998) 60–71
9. Temam, O., Fricker, C., Jalby, W.: Cache interference phenomena. Proc. SIGMETRICS/PERFORMANCE'94. (1994) 261–271
10. Uhlig, R.A., Mudge, T.N.: Trace-driven memory simulation: A survey. ACM Computing Surveys. **29(2)** (1997) 128–170
11. Zagha, M., Larson, B., Turner, S., Itzkowitz, M.: Performance analysis using the MIPS R10000 performance counters. Proc. Supercomputing '96 Conf. (1996) 17–22