ELSEVIER

Available online at www.sciencedirect.com
SCIENCE @ DIRECT®

Parallel Computing 30 (2004) 225–248

PARALLEL
COMPUTING

www.elsevier.com/locate/parco

A compiler tool to predict memory hierarchy performance of scientific codes ☆

B.B. Fraguela [a], R. Doallo [a,*], J. Touriño [a], E.L. Zapata [b]

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

[b] *Depto. de Arquitectura de Computadores, Complejo Tecnológico Campus de Teatinos, Universidad de Málaga, 29080 Málaga, Spain*

Received 15 March 2002; received in revised form 15 February 2003; accepted 15 September 2003

Abstract

The study and understanding of memory hierarchy behavior is essential, as it is critical to current systems performance. The design of optimising environments and compilers, which allow the guidance of program transformation applications in order to improve cache performance with as little user intervention as possible, is particularly interesting. In this paper we introduce a fast analytical modelling technique that is suitable for arbitrary set-associative caches with LRU replacement policy, which overcomes weak points of other approaches found in the literature. The model was integrated in the Polaris parallelizing compiler, to allow automated analysis of loop-oriented scientific codes and to drive code optimizations. Results from detailed validations using well-known benchmarks show that the model predictions are usually very accurate and that the code optimizations proposed by the model are always, or nearly always, optimal.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Memory hierarchy; Cache behavior; Performance prediction; Compiler optimizations

☆ This work was supported in part by the Ministry of Science and Technology of Spain under contract TIC2001-3694-C02-02.

* Corresponding author.

E-mail addresses: basilio@udc.es (B.B. Fraguela), doallo@udc.es (R. Doallo), juan@udc.es (J. Touriño), ezapata@ac.uma.es (E.L. Zapata).

0167-8191/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2003.09.004

1. Introduction

The widening disparity between processor and main-memory speeds makes memory hierarchy performance one of the most critical factors that influences global system performance. Programmers are aware of this and try to hand-tune it to their codes through a time-consuming trial and error process [1], either intuitively, or using costly traditional techniques such as trace-driven simulations [2], or profiling (e.g. using built-in hardware counters [3]). These methods are not the most suitable ones, because they provide very little information on the reasons for a given memory hierarchy behavior and their time requirements are high. As a result, a lot of effort is made to automatically and efficiently guide this kind of optimization. Present compiler technology usually relies on limited scope heuristics and simple analyses that lack the necessary generality and precision. New general techniques for analysing cache performance are required to give accurate predictions of memory hierarchy behavior, improve its understanding, and to provide the information needed to propose improvements to the cache configuration or the code structure.

Analytical models based on the direct analysis of the source code seem to be a very reasonable approach to meet these challenges, as they overcome the limitations of the other strategies. Still, traditionally, they have had two important disadvantages with respect to the aforementioned techniques. One has been the lack of accuracy, as many models give rough approximations of the number of misses, rather than precise quantitative estimations. This is not surprising, since cache behavior is known to be highly unstable and sensitive to small changes in a large number of parameters [4,5]: code structure, problem sizes, base addresses, cache layout, etc. Another drawback of the analytical models has been their limited scope, as, in some way, they all restrict either the code structure or the memory hierarchy they can analyse. Finally, only the construction of some models [6–10] has been systematized to the point of allowing their integration in frameworks that can apply them automatically. The resulting tools analyse cache performance and/or propose suitable program transformations in order to optimize cache behavior for the given input codes.

Our group has developed a fast analytical model [11] that makes accurate predictions of the performance of set-associative caches with LRU replacement policy for general loop-oriented codes. It is based on the generation of a set of what we denote as probabilistic miss equations (PMEs). These equations provide estimators for the number of misses generated by each reference in each loop. Unlike most of the other models found in the bibliography, which consider separately each loop nest, ours takes into consideration, the probability of hits due to reuse of data structures accessed in loops in different nests. This is very important, since most misses may occur between loop executions [12]. Although a conservative approach is applied in the current implementation, the analysis of real scientific codes without limiting the study to isolated loop-nests becomes feasible. It should also be mentioned that our model is the only one of this kind that is based on a probabilistic approach. This property enables it to predict memory system performance without knowing the base addresses of the data structures, something that no other model can do. If such

addresses are available, the model uses them to improve the prediction. A final property of our model is its velocity, which enables it to be used for extensive searches in the optimization space of large codes.

In this paper we have unified the different cases for generating the miss equations exposed in [11] in a single expression. Furthermore, the model has been integrated in the Polaris parallelizing compiler framework [13], to automatically make performance estimations and propose code optimizations. In this paper the model is validated, using both small kernels and codes from the SPECfp95 suite, and the feasibility of driving compiler optimizations that lead to optimal or almost optimal solutions using the analytical model is shown. In fact, we will see that the optimizations proposed by our tool are usually better than those of a current production compiler.

This paper is structured as follows: The following section provides a detailed review of related work. Section 3 introduces the underlying modelling ideas on which PMEs are based. It explains how to derive both formulae that give the number of misses for each reference in a set of nested loops, and the miss probabilities generated by the accesses performed during the given reuse distances that such formulae require. Details about the structure and functionality of our tool, integrated in Polaris to analyse real codes and suggest code optimizations, can be found in Section 4. The next section compares the model predictions with the measurements of trace-driven simulations to evaluate its accuracy and speed. Section 6 focuses on automatic code optimization. Section 7 follows with our conclusions.

2. Related work

A number of analytical models based on the code structure have been developed, but they differ in distinct levels of coverage with respect to scope of applicability, type of results delivered, accuracy, execution times required for obtaining model results, or implementation in a compiler framework. For example, Ferrante et al. [14] consider an arbitrary degree of associativity, but they estimate the number of different lines that the references in a given loop-nest access rather than the real number of misses. This is often misleading, as conflict misses play a very important role that is not considered in that work. A model based on a code that considers all the kinds of misses and which can be applied to general numerical codes is presented in [5], although it is restricted to direct-mapped and isolated perfect loop-nests. The miss formulae of this model are based on the calculation of the footprints on the cache of the different references of the loop. More recently, Ghosh et al. [7] and Harper et al. [8] have overcome some of these restrictions. The first paper introduces *Cache Miss Equations* (CMEs), a system of linear Diophantine equations where each solution corresponds to a potential cache miss. Although CMEs can be generated in a few seconds for standard benchmark programs and are very accurate, using them to try to predict or optimize the cache behavior seems to have heavy computing requirements. On the other hand, Harper et al [8] focus on the same class of nests as [5] and also base their approach on footprints. Their model is devoted to set-associative

caches and has modeling times very similar to those of our tool. Tests show that the error of our model is usually between two and three times smaller than theirs, using the same example codes they propose.

All the works referenced above share a common limitation that we have overcome: their modelling is suitable for regular access patterns in isolated, perfectly nested loops, and they do not take into account the probability of hits in the reuse of data structures referenced in previous loops. This is a very important issue, as pointed out by McKinley and Temam [12]. In any case, our model is not the only one capable of handling non-perfect loop-nests and analysing a whole code rather than isolated loop-nests. Recently Vera and Xue [9] have reduced the computing requirements of the CMEs by applying statistical techniques and have extended the model to handle complete codes with regular computations. Table 1 compares the accuracy and speed of that model and ours using the cache configurations that they use in most of their evaluations. MMT is a matrix product with blocking in two dimensions that Vera and Xue use to compare their model with ours, while Tomcatv and Swim are two well-known SPECfp95 codes. We can see that accuracy is similar, while our model is much faster. Notice that the models must be applied hundreds or thousands of times to search in the optimization space for optimal block sizes, paddings, etc. So typical compiler applications will require computing times which are several orders of magnitude longer than those required by a single evaluation of the models. An approach based on Presburger formulae [10] also addresses the reuse in non-perfect loop-nests and gives exact predictions for small kernels that can include certain conditional statements and non-linear data layouts, while allowing a certain amount of symbolic analysis. Drawbacks also exist, as it can only handle modest levels of associativity and is very time-consuming, which currently reduces its applicability. Blieberger et al. [15] propose a model completely focused on symbolic evaluation and analysis which addresses set-associative caches. Their strategy

Table 1
Comparison between Vera and Xue's *EstimateMisses* algorithm (subindex E) using a 95% confidence and an interval of 0.05 [9], and our probabilistic model (subindex P) for a 32 Kbyte cache with lines of 32 bytes and different degrees of associativity (K)

Benchmark	K	Err_E	Err_P	Time_E	Time_P
MMT	1	0.23	0.02	100.00	0.45
$N = BJ = 100$,	2	0.37	0.20	100.00	0.36
$BK = 50$	4	0.37	0.06	110.00	0.32
Tomcatv (ref. input set)	1	0.04	0.31	300.00	0.87
	2	0.03	0.32	370.00	0.88
	4	0.06	0.11	580.00	0.92
Swim (ref. input set)	1	0.25	0.05	2470.00	2.30
	2	0.25	0.12	2630.00	2.10
	4	0.27	0.03	3230.00	2.00

The error (Err) is expressed as the absolute difference between the predicted and the measured miss ratios. Modeling times (Time, in ms) for both models were measured on a 933 MHz Pentium III.

involves instrumenting source codes that are then symbolically evaluated to generate symbolic trace-files which store all the references of the program as symbolic expressions and recurrences. The symbolic evaluation of this trace-file, based on the cache parameters, yields the final analytical cache hit function. Unfortunately it has only been evaluated with simple kernels, for which it yields very accurate results, and there is no information about its computing requirements. A very important advantage, exclusive of our model, is that it is able to predict the behavior of a code without knowing the base addresses of the data structures, which is a quite common situation. For this reason it is also the only one applicable to physically indexed caches or dynamically allocated data structures.

Some other recently designed tools, such as Delphi [6] and SPLAT [16] analyse codes through both run-time profiling and analytic models. Delphi bases its model on stack distances, by generating stack histograms for each individual loop-nest. This restricts the accurate estimation to fully associative caches with LRU replacement policy (although it includes a probabilistic approach to handle general set-associative caches), and causes the loss of reuse information between different loop-nests. Delphi also includes a model for indirect accesses and a CPU model that we have used as a complement to our model in Section 6, to obtain real computing time estimations. The locality analysis in SPLAT uses a series of phases. In the volume phase the cache behavior is considered to be similar to that of an LRU fully associative cache, while in the interference phase the model only focuses on direct-mapped caches. However, it relies on the CMEs to estimate the behavior of set-associative caches. Both tools, Delphi and SPLAT, require computing times to perform their analysis which are typically several orders of magnitude larger than those of our tool. In both cases prediction errors are usually similar or larger than ours, but only consider codes in which the vast majority of the accesses are sequential, like *tomcatv* and *swim*. These kinds of codes are better suited to their strategy of modelling set-associative or direct-mapped caches as if they were fully associative. Experiments using the matrix product with blocking, a code which presents a larger variety of access patterns, show that large deviations appeared in the predictions of Delphi (we had no access to SPLAT), while our model generates very good predictions (see Section 5).

3. Model concepts

We classify misses as either cold or interference misses. When a line is accessed for the very first time, it gives place to a cold miss. All the remaining misses on that line take place when the processor tries to reuse the line, but the line has been replaced due to accesses to other lines that interfere with it in its cache set. For example, in a K-way associative cache with LRU replacement, the condition for a cache line to be replaced is that K, or more different lines mapped to its same cache set, be accessed since the last access to this line. Analysing the access patterns of the different references during a given portion of the program execution, provides knowledge of how many different lines have been accessed, as well as how many of them will be reused

and which other lines will be accessed between those reuses. This way, miss equations that estimate the number of misses that such references generate can be constructed. Our model differs from all the related works in the bibliography in its probabilistic approach. It builds PMEs based on the estimation of miss probabilities for the line reuses. These probabilities depend on the cache area affected by the accesses performed between the consecutive accesses to each line.

Let us now consider an initial modelling scope to develop these ideas. This scope is a set of normalized nested loops, like the one in Fig. 1, where several arrays are accessed through references indexed by affine functions of the enclosing loop variables. The references can be found in any nesting level, so imperfectly nested loops can be analysed. The number of iterations of each loop will be known at compile time and is the same in each execution of the loop. The code is executed in a machine with an arbitrary cache size, line size and degree of associativity. The only limitation regarding the cache is that the replacement policy must be LRU. Under these conditions, which depict the most common indexing scheme in scientific and engineering codes, a PME for each reference in each nesting level is generated following some simple ideas:

- Each loop enclosing a given reference gives place to a stride in the accesses of the reference (which may be zero) and to a fixed number of accesses following this stride. This way, a function to estimate the number of misses generated by the reference during execution of that loop can be built (see Section 3.1).
- The existence of several references to a given array gives place to reuses of those lines that are accessed by two or more of these references (group reuse). As a result, miss equations for loops in which these reuses occur will not reflect the total size of such loops, but rather the number of iterations between consecutive reuses. In order to simplify the analysis, our model only computes the reuse among refe-

```
DO I_0=1, N_0
  ...
  DO I_1=1, N_1
  ...
    DO I_Z=1, N_Z
      A(α_A1 I_A1 + δ_A11, ..., α_AdA I_AdA + δ_AdA1)
      ...
      A(α_A1 I_A1 + δ_A12, ..., α_AdA I_AdA + δ_AdA2)
      ...
      A(α_A1 I_A1 + δ_A1ℜA, ..., α_AdA I_AdA + δ_AdAℜA)
    END DO
    ...
  END DO
  ...
END DO
```

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

rences which are in translation, that is, their indices only differ in one or more of the added δ constants. Most reuse in scientific codes comes from this kind of reference groups.

- The cache area affected by the accesses of a given reference during a number of executions of any of its enclosing loops can be computed either analytically or through simulation, or following a mixed approach. The probability that these accesses generate interferences that affect this specific reference, or others, can be derived from this cache area (see Section 3.2).
- Once these ideas are developed, it is clear that there is no reason to restrict our study to a single simple nest. Computing reuse distances and interference probabilities between references located in different nests has also been done, although a number of restrictions must be met (see [11] for more information).

The modelling requires data such as the size of the loops, the indices of the references and so on. Many times this data is not available at compile time. In these cases, code instrumentation could capture such data at run-time, which for example is the approach of Delphi [6]. As for data locations, the model can use them if they are available, but it can also predict the memory behavior using a completely probabilistic approach that does not use them. Our model cannot be applied to codes that include conditional structures.

In the following we discuss the PMEs and the miss probabilities that PMEs use in Sections 3.1 and 3.2, respectively. It is interesting to note that although this basic modelling scope allows the analysis of many programs, our modelling ideas are general enough to allow the expansion of this scope in a number of ways. For example, our model already supports some kinds of loop dependences (blocking) and loops with a different number of iterations in different executions (triangular loops).

3.1. Probabilistic miss equations

Our model associates a probabilistic miss estimator $F_{Ri}(p)$ to each reference R and enclosing loop at nesting level i, which estimates the number of misses the reference generates during the execution of that loop. The estimator is a function of the miss probability p in the first access to a line of the data structure that R references during the execution of that loop. This probability is an input parameter for the estimator in this nesting level, because it depends on the access patterns and accesses in outer and/or previous loops. If R is enclosed by loop $i+1$, $F_{Ri}(p)$ is calculated in a PME in terms of $F_{R(i+1)}(p)$, as each iteration of the loop i involves the execution of the immediate inner loop $i+1$, and thus $F_{R(i+1)}(p)$ includes the access pattern of the reference in that loop and the misses generated in it. On the other hand, the analysis of loop i provides p for any immediate inner loop $i+1$. Thus, the analysis of each reference starts in the innermost loop that contains it, and finishes in the outermost loop. In the innermost loop containing a reference, there is no inner loop to analyze that carries information on the access pattern of the reference. As a result, in the PME that generates the estimator for this loop, the $F_{R(i+1)}(p)$ estimator really corresponds to one execution of the loop body, rather than to the execution of any inner loop. This

way, in the PME that generates the estimator for the innermost loop, $F_{R(i+1)}(p)$ is replaced by p, the probability of a miss in any single access. The input miss probability for the outermost loop of the code is one, as there are no portions of the data structure in the cache when the execution begins. Notice that the generation of a miss estimation per reference and loop facilitates the detection of those references and loops where data accesses become a serious bottleneck (hot spots).

Let N_i be the number of iterations of the loop at nesting level i, and L_{Ri} be the number of iterations in which R cannot reuse lines in this loop, then the estimator $F_{Ri}(p)$ for this reference R is obtained by the PME

$$F_{Ri}(p) = L_{Ri}F_{R(i+1)}(p) + (N_i - L_{Ri})F_{R(i+1)}(P_R(\mathrm{It}(i,1))) \quad (1)$$

if R carries no reuse with other references. $\mathrm{It}(i,n)$ represents the memory regions accessed during n iterations of the loop at nesting level i, and $P_R(\mathrm{RegionsSet})$ is the interference probability that the access to region(s) RegionsSet generates on the accesses of R. In Section 3.2 we elaborate on the mapping of accessed regions to the interference probabilities they generate. This formula reflects that the miss probability for the L_{Ri} loop iterations in which there can be no reuse in this loop, depends on the accesses and patterns in the outer loops (given by p), while the miss probability for the remaining iterations and their accesses is a function of the regions accessed during the portion of the program executed between those reuses, which is one iteration of this loop.

The indices of R are affine functions of the enclosing loop variables, so there is a constant stride S_{Ri} for this reference associated to the loop i. As a result, L_{Ri} can be calculated as

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

where L_s is the number of array elements a cache line holds. The formula estimates the number of accesses of R that cannot exploit either temporal or spatial locality, and it is equivalent to estimating the number of different lines that are accessed in N_i iterations with stride S_{Ri}. On the one hand, if the index of loop i does not index reference R, then $S_{Ri} = 0$ and $L_{Ri} = 1$. This is correct, as in that case there is a single iteration for cold accesses with respect to this loop (the first one) and all the remaining iterations are potential reuses due to temporal locality. On the other hand, if R is indexed by the loop variable, then $S_{Ri} > 0$ and it accesses different data items in different iterations. In this case, the reuse of lines related to the execution of this loop is only possible by exploiting spatial locality. As we access L_{Ri} different lines during the N_i iterations of the loop with stride S_{Ri}, then this locality can be exploited by the remaining $N_i - L_{Ri}$ iterations. This way Eq. (2) captures both temporal and spatial locality, and in either case the reuse distance is always one iteration of the loop which is being considered, which justifies Eq. (1).

The above formulae are applied when the reference carries no reuse with other references. When the reference belongs to a group of references in translation, they all exhibit the same strides, but may differ in the δ constants of the indices. This gives rise to possible (spatial or temporal) systematic group reuse between them, which

is taken into account when building the corresponding PMEs. Our model sorts the references in descending order of their base address in order to compare them and estimate the reuse distance, measured in loop iterations, of the lines accessed by these groups of references. The PMEs corresponding to these references are summationies, in which each term expresses the number of accesses that have a given reuse distance multiplied by the miss probability associated with that distance. This algorithm is explained in [11].

Example. In order to illustrate our model, we will explain the modelling of the following simple loop, where the references have been numbered:

```
REAL*8 A(250), B(250,250)
DO I=1, 200
  T=T+A1(I)+A2(I+2)
  B3(1,I)=2*T
END DO
```

Let us consider for example a computer with 8-byte words and a 2-way 4 Kw cache with 4-word lines. Then, $L_s = 4$, and the PMEs for references R_2, `A(I+2)`, and R_3, `B(1,I)`, for loop `I` at nesting level 0 with $N_0 = 200$ iterations, would be:

$$F_{R_2 0}(p) = 50 \cdot p + 150 \cdot (P_{R_2}(\mathrm{It}(0,1)))$$
$$F_{R_3 0}(p) = 200 \cdot p$$

as R_2 has stride $S_{R_2 0} = 1$ in the loop, which implies in Eq. (2) that $L_{R_2 0} = 50$; and the stride for R_3 is $S_{R_3 0} = 250$, resulting in $L_{R_2 0} = 200$. Notice that L_{Ri} is always the number of different lines the reference R accesses in the loop. Finally, this loop contains no other loops, so $F_{R(i+1)}(p)$ is replaced in the PMEs by p, as we have explained.

The PME for reference R_1, `A(I)`, cannot be derived from Eq. (1) because it exhibits group reuse with respect to the reference R_2. Although this kind of modeling is not explained in this paper, we illustrate it for this example with this simple reasoning. Its PME is $F_{R_1 0}(p) = 100 \cdot (P_{R_1}(\mathrm{It}(0,1)))$, because in 50 out of the 200 iterations of the loop it is accessing the line that R_2 accessed in the previous iteration, and in another 50 iterations it is accessing the same line it accessed in the previous iteration. This yields a total of 100 accesses whose reuse distance is $\mathrm{It}(0,1)$, this is, one execution of the loop under consideration. In the remaining 100 iterations, R_1 is accessing the same line as R_2 and there are no intermediate accesses, so these accesses cannot result in misses. As a result, these iterations are not represented in the PME.

3.2. Miss probabilities

In Section 3.1 interference probability has been represented through the reuse distance between two consecutive accesses to the line accessed by the reference whose behavior is being studied. The process to calculate the miss probability associated to a given reuse distance has three steps: *Access Pattern Description*, *Cache Impact Quantification*, and *Area Vectors Addition*. The first one involves the description of

the access pattern followed by the references inside the reuse distance. This is achieved by examining the indices of each dimension as well as the number of iterations during the reuse distance for each loop that controls these indices. Considering these two factors together, we will obtain the shape and size of the accessed region.

In the second step, *Cache Impact Quantification*, each region is analyzed in order to describe the way its access affects the cache. This involves calculating the cache footprint of the region and its contribution to the miss probability we are estimating. This is represented in our model by the interference area vectors. Given a data structure A and a K-way associative cache, we call $S_A = S_{A_0}, S_{A_1}, \ldots, S_{A_K}$ is the area vector associated with the accesses to A during a given period of the program execution. The element in the ith position of the vector stands for the ratio of sets that have received $K - i$ lines of this structure. The exception is S_{A_0}, as it is the ratio of sets that have received K or more lines. Thus, S_{A_0} is the probability that the accesses to array A produce an interference, as we are considering caches with LRU replacement policy. Notice that estimating the area vector of a given region just means counting how many different lines of that region fall in each cache set. A region has two kinds of interference area vectors associated to it. Self-interference area vectors contain the information required to estimate the miss probability on the reference(s) that access the region they depict. On the other hand, cross interference area vectors are generated by the accesses of the references to other regions. The calculation of both kinds of area vectors for a given access pattern is different only because a given line does not generate interferences with itself, thus the calculations are very similar.

There are a number of typical access patterns which account for the vast majority of the accesses in scientific codes. A previous work [17], focused on non-automatic modeling of irregular codes, describes formulae and algorithms to estimate the area vectors associated to many of these patterns. The most common regular patterns found in loop-oriented scientific and engineering applications are sequential access and access to regions of the same size separated by a constant stride. Simulation must be used when we lack the analytical tools to calculate this impact.

In the following, we show the sequential access to n words as an example, in order to illustrate our area vector estimation approach. Its cross interference area vector, $S_s(n)$, can be estimated as:

$$\begin{aligned} S_{s_{(K-\lfloor l \rfloor)}}(n) &= 1 - (l - \lfloor l \rfloor) \\ S_{s_{(K-\lfloor l \rfloor - 1)}}(n) &= l - \lfloor l \rfloor \\ S_{s_i}(n) &= 0 \quad 0 \leqslant i < K - \lfloor l \rfloor - 1, K - \lfloor l \rfloor < i \leqslant K \end{aligned} \tag{3}$$

where $l = \max\{K, (n + L_s - 1)/(C_s/K)\}$ is the average number of lines of this access that are placed in each set when it finishes. In this expression L_s stands for line size and C_s for cache size. The term $L_s - 1$ added to n stands for the average extra words brought to the cache in the first and last accessed lines. The formula expresses that $(l - \lfloor l \rfloor) \times 100\%$ of the cache sets keep $K - \lfloor l \rfloor - 1$ lines of this access and the remaining cache sets keep $K - \lfloor l \rfloor$ lines.

The self-interference area vector associated to this access, $S_{sa}(n)$, expresses the interference that the lines associated to this access generate on themselves. It can be estimated as $S_{sa}(n) = S_s(C(n)C_s/K)$, because the self-interference for each individual line is equivalent to the cross interference generated by the sequential access to $C(n)C_s/K$ words, where $C(n)$ is the average number of lines of this access that compete with another line in its set. This value is calculated as $C(n) = \lfloor v \rfloor / v(2v - \lfloor v \rfloor - 1)$ where $v = (n + L_s - 1)/(C_s/K)$.

Once the effect on the cache of the accesses to the different data structures has been calculated, a third step, *Area Vectors Addition*, is required to add them together, in order to obtain the global effect on the cache. If the relative position of the data structures is known, each area vector is scaled before its addition: the amount of overlapping between its region and the data structure associated to the reference whose PME is being calculated is expressed using a coefficient. Once the scaling has been done, the area vectors are added considering their area ratios as independent probabilities. Given two area vectors S_U and S_V, their addition, represented by the operator $\cup$, is calculated as

$$\begin{aligned}
(S_U \cup S_V)_0 &= \sum_{j=0}^{K} \left(S_{U_j} \sum_{i=0}^{K-j} S_{V_i} \right) \\
(S_U \cup S_V)_i &= \sum_{j=i}^{K} S_{U_j} S_{V_{(K+i-j)}} \qquad 0 < i \leqslant K
\end{aligned} \tag{4}$$

It should be mentioned that when the analysed references are in translation, the access pattern is sequential and the array base addresses are known, then an optimized algorithm which considers the pathological conflicts that may arise in such cases is applied.

Example. The PMEs calculated in the example in Section 3.1 must be completed with the corresponding miss probabilities. Both $F_{R_1 0}(p)$ and $F_{R_2 0}(p)$ depend on the miss probability on the line they access due to the regions accessed during one loop iteration. In each of the iterations in our example loop, two elements (separated by one) from `A` and one element from `B` are accessed. This access pattern to `A` is equivalent, from the point of view of the cache footprint, to an access to three consecutive elements. This way, both regions can be modeled as a sequential access. As the degree of associativity is $K = 2$, the area vectors that depict these regions have three elements. The cross interference area vectors associated to these regions are $S_{\text{cross A It}(0,1)} = S_s(3) = (0, 1.5/512, 510.5/512)$ for array `A`, and $S_{\text{cross B It}(0,1)} = S_s(1) = (0, 1/512, 511/512)$ for matrix `B`, both calculated according to Eq. (3). The self-interference area vectors for both data structures would be $S_{\text{self A It}(0,1)} = S_{sa}(3) = S_{\text{self B It}(0,1)} = S_{sa}(1) = (0, 0, 1)$, as neither of these regions interferes with itself in the cache.

As explained, the last step in order to calculate the miss probabilities is to add the area vectors corresponding to the different data structures accessed during the considered period of the program execution. In our example, both P_{R_2} (It(0, 1)) and P_{R_1}

$(\mathrm{It}(0,1))$ are the first element of the area vector $S_{\text{cross B It}(0,1)} \cup S_{\text{self A It}(0,1)}$, which yields (0, 1/512, 511/512) according to Eq. (4). As a result, the final expression for the PMEs of the three references in our example code is $F_{R_1 0}(p) = 0$, $F_{R_2 0}(p) = 50 \cdot p$ and $F_{R_3 0}(p) = 200 \cdot p$.

4. A compiler framework for analytical modelling

The final purpose of the modelling task described in the previous section is the integration of the model in a compiler environment. This allows the analysis of real scientific codes, both effectively and quickly, in such a way that code optimizations can be proposed. Our modelling technique has been embedded in the Polaris parallelizing compiler [13], using its development infrastructure. The whole environment also includes the Delphi CPU model [6]. Thus, the computation cycles predicted by Delphi can be added to the stall cycles caused by the misses predicted by our model, in order to estimate real execution times. Memory hierarchies of arbitrary size, block size and associativity per memory level, and LRU replacement policy (which nowadays is by far the most common) can be considered to depict the behavior of memory systems. Basically, our framework takes as inputs the memory hierarchy parameters and a FORTRAN 77 program, and returns system performance results using our analytical models. It also provides timing results for each stage of the tool, as well as the desired level of monitoring capabilities (from a user-level up to a developer-level).

The tool structure is depicted in Fig. 2, and it consists of the following blocks:

- In the *Cache Parameters* configuration file, the user specifies for each level of the memory hierarchy: its size, line size (both expressed in number of words), degree of associativity and miss weight. This file also contains, among others, switches to enable/disable the code generation of the simulator and also the analyser shown in Fig. 2. All parameters can also be specified on the command line.
- The *Modelling Library* is a set of C routines that collect, in a modular way, our analytical model described in the previous section.
- The *Delphi CPU Modelling Library* is used in those cases in which predictions involving the processor execution time are required.
- The *Optimization Library* analyses the input code and uses the predictions of the memory hierarchy behavior (and also the CPU, if required), and suggests code restructuring in order to reduce the total execution time. An example of optimization currently implemented is the selection of an optimal block size for the points in the program where a blocking pattern is detected.
- The *Parser/Integrator* is the integrative part of this framework. It analyses the source code, detects array reference patterns inside a loop, and uses the corresponding model from the *Modelling Library*. Note that the functionality of our performance model could be embedded in other compiler environments by changing this module.

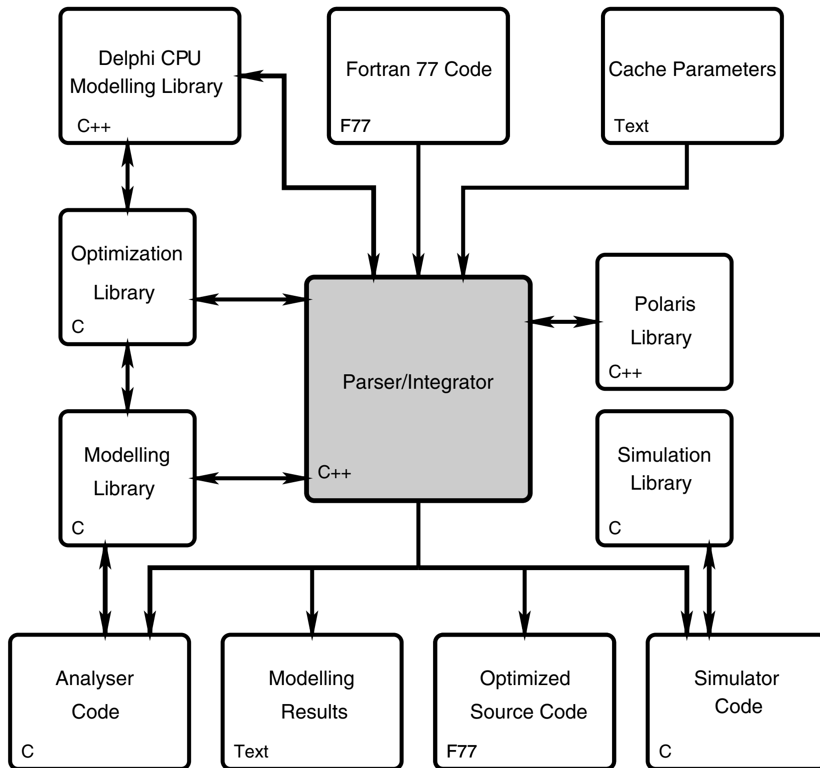

Fig. 2. Tool diagram.

Our tool provides both general and detailed information about cache behavior in the *Modelling Results*: global number of misses, misses per data structure, misses per loop (or set of loops). Thus, cache hot spots can be detected and localized to drive code optimizations. Optionally, the following files can be generated:

- The *Analyser Code* is a C program (with the appropriate calls to the *Modelling Library*) that obtains the modelling results of the input code (although the cache parameters and optionally, the data structures base addresses, can be altered). The aim of this code is to be used afterwards for testing purposes on any machine, independently of the Polaris environment.
- The *Simulator Code* is the C code of a trace-driven simulator that validates the analytical results obtained for the specific input code. As in the case of the analyser, it allows the modification of the memory hierarchy parameters and the data structures base addresses. Simulation is performed by an optimized library of functions (*Simulation Library* in Fig. 2) that has been extensively validated matching its results with those of the dineroIII simulator (a component of the WARTS toolset [1]).

- The *Optimized Source Code* is the input code after a source-to-source transformation. In case of a blocking transformation, currently implemented, for each blocking, the original block size is replaced by the optimal block size (computed by means of our analytical models).

5. Experimental results

The main validation of our model has been performed by comparing the number of misses it predicts with the values measured using trace-driven simulations. The code to perform these simulations can also be generated automatically by our tool (*Simulator Code* and *Simulation Library* in Fig. 2). Thus, we have applied both approaches to a series of codes using a wide variety of combinations of cache parameters and data structure dimensions. Also, for each combination, several simulations (approximately twenty) were performed, changing the value of the data structure base addresses using a random generator.

Two kinds of metrics have been calculated for each cache parameter combination: Δ_{MR}, which is based on the miss ratio (MR), and Δ_{NM}, based on the number of misses. The first metric, Δ_{MR}, is the average of the differences between the predicted and measured miss ratios obtained for each combination of array base addresses tried for each cache; Δ_{NM} is the average error in the prediction of the number of misses in these trials, expressed as a percentage of the number of measured misses. Both the miss ratio differences and the percentage error in the prediction of the number of misses were taken in absolute value to calculate the averages, so that negative and positive values did not compensate each other. The standard deviation, σ, of the number of measured misses in the simulations (expressed as a percentage of the average number of measured misses) is also taken into account to help understand the memory behavior of the algorithms.

5.1. Validation through synthetic codes

The validation was first performed using three kernels covering different access patterns and loop-nests:

(1) The synthetic code in Fig. 3 represents a type of code with several non-perfect nestings. A number of 216 parameter combinations (and more than 4000 simulations) were tried.
(2) A matrix–matrix product with blocking (Fig. 4) was validated checking 2400 parameter combinations resulting in 16,320 simulations.
(3) Finally, a system solver by forward substitution (shown in Fig. 5) that involves a triangular loop, has also been included; 480 parameter combinations were tried, requiring a total of 9600 simulations.

Table 2 includes validation results for these codes. The Δ_{MR} values are highly satisfactory, with small errors. The codes also perform very well on average in the pre-

```
DO L=1, M
  DO J=1, N
    DO I=1, N
      R=0.0
      DO K=1, N-1
        R=R+A(I,K)*B(K,J)
      END DO
      D(I,J)=R
    END DO

    R=0.0
    DO I=1, N
      R=R+D(I,J)*V(I)
    END DO
    E(J,L)=R
  END DO

  DO J=1, N-1
    DO I=1, N
      A(I,J)=D(I,J)+D(I,J+1)
    END DO
  END DO
END DO
```

Fig. 3. Example code with several non-perfect nestings.

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

Fig. 4. Matrix–matrix product with blocking in two dimensions.

diction of the absolute number of misses, although the deviations are somewhat higher. This difference in the validation metrics means that the important deviations take place in the parameter combinations for which a small number of misses is obtained. That is, those in which the memory behavior analysis produces little optimization. A relatively small error in the prediction of the number of misses for these combinations generates very large values of Δ_{NM}, which unbalance the average.

```
DO I=1, N
  R=B(I)
  DO J=1, I-1
    R=R-A(I,J)*X(J)
  END DO
  X(I)=R/A(I,I)
END DO
```

Fig. 5. Forward substitution code with triangular loop.

Table 2
Average validation values (in percentage)

Algorithm	MR	Δ_{MR}	Δ_{NM}	σ
Non-perfect nestings	12.53	0.13	7.57	39.10
Matrix product	21.69	0.30	6.14	12.68
Forward substitution	17.33	0.39	5.77	1.22

For example, only 31.3% of the combinations for the code with non-perfect nestings have a Δ_{NM} over 5%, and their average miss ratio Δ_{MR} is 0.22%; while the remaining 68.7% of the combinations have an average miss ratio of 17.94%. Even more meaningful is the fact that if the parameter combinations with miss ratios smaller than 0.5% are not considered for the calculation of the average Δ_{NM} (just those in which the memory hierarchy is already behaving very well), it drops to only 1.45%.

Tables 3–5 show the validation results of some parameter combinations for the codes shown in Figs. 3–5, respectively. In these tables and in what follows, C_s stands for the cache size in Kwords, L_s is the line size in words and K is the degree of associativity, while M, N, BJ and BK stand for the problem size (see the related codes). These tables also include average times in seconds, both for the trace-driven simulation used for the validation (T. sim.) and modelling (T. mod.), measured in an SGI Origin 200 server with R10000 processors at 180 MHz. We must recall that many efforts have been made to optimize our simulator, and these simulation times are

Table 3
Validation and time results corresponding to the non-perfect nestings code

M	N	C_s	L_s	K	MR	Δ_{MR}	Δ_{NM}	σ	T. sim.	T. mod.
100	100	2	2	1	29.33	0.15	0.50	0.69	34.845	0.003
100	100	64	8	2	0.01	0.01	17.95	172.154	42.534	0.006
200	200	4	4	4	12.68	0.00	0.03	0.01	949.837	0.003
200	200	16	16	1	4.49	0.14	3.03	3.28	555.659	0.004
200	400	64	16	2	3.15	0.01	0.35	0.65	5364.425	0.006
400	100	128	16	4	0.00	0.00	0.16	0.09	234.490	0.006
400	200	64	8	4	0.11	0.01	8.38	2.14	1909.567	0.004
400	400	1024	128	2	0.00	0.00	41.17	109.54	10587.293	0.069

Table 4
Validation and time results corresponding to the matrix product code

N	BJ	BK	C_s	L_s	K	MR	Δ_{MR}	Δ_{NM}	σ	T. sim.	T. mod.
200	100	200	2	4	1	30.11	0.06	0.20	0.19	3.70	0.005
200	100	100	16	4	2	6.73	0.05	0.69	0.54	4.31	0.005
200	50	100	32	4	4	0.41	0.01	3.03	0.97	5.65	0.006
400	50	50	4	8	1	12.71	0.24	1.86	0.56	29.55	0.005
400	200	200	16	8	2	6.53	0.02	0.25	0.25	34.25	0.006
400	100	50	64	16	4	0.08	0.01	10.09	7.51	44.64	0.008
400	200	100	128	16	2	0.10	0.01	11.37	18.91	35.36	0.017
400	50	400	256	32	4	0.01	0.00	9.72	7.95	44.72	0.015

Table 5
Validation and time results corresponding to the forward substitution code

N	C_s	L_s	K	MR	Δ_{MR}	Δ_{NM}	σ	T. sim.	T. mod.
200	8	32	1	40.53	0.25	0.61	0.69	0.009	0.009
500	4	4	2	13.09	0.49	3.72	0.77	0.066	0.003
500	32	16	1	4.38	0.39	8.72	3.61	0.057	0.016
1000	16	8	1	10.16	0.86	8.43	1.15	0.225	0.013
1000	32	4	4	12.58	0.04	0.30	0.00	0.353	0.011
1000	128	16	2	3.21	0.07	2.24	0.34	0.269	0.035
2000	64	16	2	3.18	0.04	1.34	0.19	1.046	0.044
2000	256	8	4	6.28	0.02	0.35	0.00	1.434	0.054

much shorter than those we would obtain using more general and standard simulators. Even so, modelling is typically between three and six orders of magnitude faster. The exception is the forward substitution code, where the triangular loop halves the number of accesses and thus, the simulation time, while its irregularity forces our tool to perform a sampled simulation to estimate the area vector. Still, modelling is much faster than simulation.

Large values of Δ_{NM} arise in the non-perfect nestings code for some combinations with a very small miss ratio (almost 0%), where a prediction error of a small number of misses in absolute terms becomes a large relative error, as we have already explained. There is another general and important reason for this behavior of the model for these combinations. Due to the probabilistic nature of the model, the convergence of this kind of approach is not favoured by small problems where few misses are generated. An additional reason for some large deviations in codes with non-perfect nestings is the use of a conservative approach in the estimation of the number of misses for data structures that have been accessed in previous loops. This simple and quick strategy affects the accuracy of the prediction for codes with several non-perfect nestings, giving place to an overestimation.

Another interesting fact in the tables is that the parameter combinations with the greatest prediction errors are those that have a wide variation (high σ) in their number of misses depending on the relative positions of the data structures. Moreover, such prediction deviations are almost always similar to or noticeably smaller than

the standard deviation of the real number of misses, making them a very reasonable prediction even when the relative error is large. Again, this effort can be explained by the probabilistic nature of the model: it tends to predict a number of misses close to the average number of misses generated through the different simulations, but the particular values generated in the different simulations may be far from this average value.

5.2. Validation through the SPECfp95 suite

In order to demonstrate the wide range of validity of our tool, we applied it to programs belonging to the SPECfp95 suite, as shown in Table 6. In the case of the first two SPECfp95 programs we have analysed the whole code, while in the remaining codes we have chosen the most CPU time consuming routine. In the cases where such routine calls other routines, they have been inlined in the calling routine. Four benchmarks were not considered in our study, for different reasons: *turb3d* and *apsi* most demanding loops are very small and so they generate very few misses; *fppp* main routines contain many non-regular loops and conditional sentences; finally, *wave5* has many indirect accesses, not suitable for our analysis techniques.

Table 6 shows the resulting number of loops and references analyzed (Loops and Refs columns, respectively), as well as the percentage of the program execution time associated to them (% Ex. T.) and the validation results for each code. As in the previous experiments in Section 5.1, miss ratio errors (Δ_{MR}) are very good, with a maximum average value of only 0.12%. The average errors relative to the number of misses are somewhat larger, but still small and within the range of the corresponding σ. Notice also that the largest relative deviations usually take place in the codes with the smallest miss ratios: memory hierarchy is working better and the small number of misses turns an error of a few misses into a large relative error. In any case, Δ_{NM} exhibits excellent values. In fact, Fig. 6 proves the high precision of the model for some caches with different sizes and degrees of associativity. Benchmark *su2cor-matmat* has not been included in the figure because of its reduced number of misses, and the number of misses generated by *tomcatv* and *swim* have been scaled to fit in the plot. Tables 7 and 8 shows the miss ratio deviations Δ_{MR} and modelling times for the same cache configurations, respectively. These times refer only to the model execution itself, so they do not include the time required by Polaris to read the file, parse

Table 6
Main code parameters and average validation values (in percentage)

Benchmark	Loops	Refs	% Ex. T.	MR	Δ_{MR}	Δ_{NM}	σ
Tomcatv	16	75	100.0	6.27	0.12	1.90	2.66
Swim	23	193	100.0	6.46	0.02	0.57	1.97
Su2cor-matmat	1	36	23.5	5.61	0.01	0.12	0.00
Hydro2d-filter	24	120	46.8	7.70	0.11	1.84	2.77
Mgrid-resid	9	45	55.5	2.21	0.08	5.82	6.26
Applu-blts	11	20	28.3	5.28	0.05	1.25	0.22

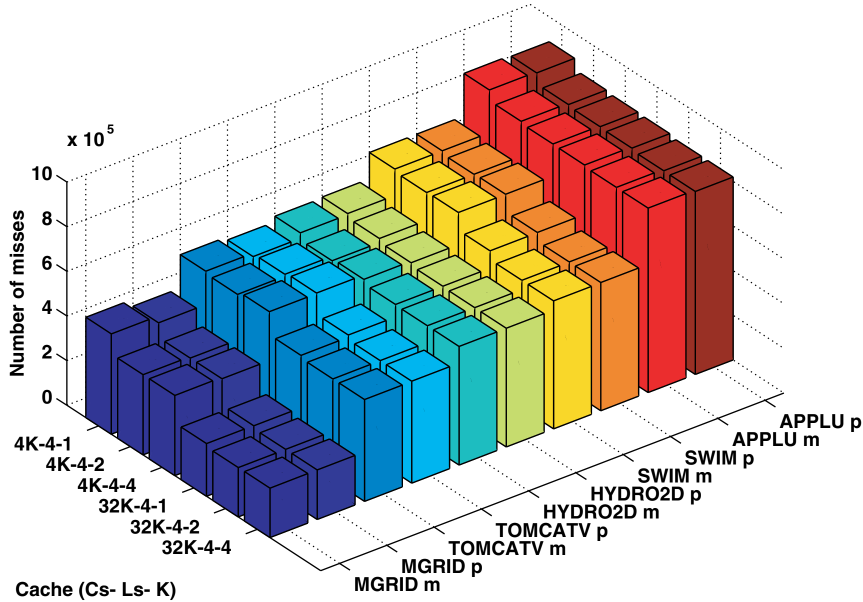

Fig. 6. Measured (m) versus predicted (p) misses for the SPECfp95 codes. Cache size (C_s) and line size (L_s) in words; K stands for the degree of associativity.

Table 7
Miss ratio errors Δ_{MR} of the model for the suite codes considering different cache configurations

Benchmark	Cache configuration (C_s–L_s–K)					
	4K–4–1	4K–4–2	4K–4–4	32K–4–1	32K–4–2	32K–4–4
Tomcatv	0.291	0.439	0.112	0.181	0.018	0.017
Swim	0.009	0.009	0.009	0.015	0.015	0.016
Su2cor-matmat	0.006	0.006	0.006	0.006	0.006	0.006
Hydro2d-filter	0.255	0.408	0.149	0.005	0.006	0.006
Mgrid-resid	0.360	0.076	0.075	0.012	0.006	0.002
Applu-blts	0.045	0.055	0.035	0.055	0.065	0.063

Table 8
Modeling times for the suite codes for different cache configurations (in seconds)

Benchmark	Cache configuration (C_s–L_s–K)					
	4K–4–1	4K–4–2	4K–4–4	32K–4–1	32K–4–2	32K–4–4
Tomcatv	0.023	0.023	0.023	0.023	0.023	0.023
Swim	0.043	0.042	0.042	0.048	0.046	0.044
Su2cor-matmat	0.004	0.003	0.004	0.003	0.003	0.003
Hydro2d-filter	0.020	0.020	0.020	0.024	0.022	0.022
Mgrid-resid	0.021	0.021	0.021	0.021	0.022	0.022
Applu-blts	0.009	0.008	0.008	0.025	0.016	0.011

it, and so on. The modelling times are at least one order of magnitude faster than the simulation, and even with respect to the execution time. As an example, the *tomcatv* amd *swim* benchmarks require, respectively, 192 and 131 s to be executed using the reference input set, and the corresponding cache simulation is at least one order of magnitude slower.

6. Automatic code optimization

An additional set of experiments using our model to check the possibility of driving optimizations in real memory hierarchy systems is described below. The high degree of accuracy obtained in the validation experiments shown in the preceding section was a support for this idea.

The matrix–matrix product with blocking shown in Fig. 4 was chosen as a case study for that purpose. We tried to derive the optimal block sizes for four representative architectures by feeding our tool with the code and the parameters describing the memory hierarchy levels of such systems (see Table 9): a PC with a PentiumII at 450 MHz (PC pII), a Digital Personal Workstation 433au with a 21164 at 433 MHz (PW 433au), an SGI Origin 200 with R10000 processors at 180 Mhz (O200 R10K) and the nodes of a Fujitsu AP3000 multicomputer with UltraSparc-II processors at 300 MHz (AP U-II). Each memory hierarchy level i of these systems was specified to our model by means of C_{s_i}, L_{s_i} and K_i, and a new parameter W_i, the relative miss penalty. We have obtained the values of W_i for the different memory levels of these machines either from the corresponding manuals or using microbenchmarking [18]. This way, the cost of a given block $BJ \times BK$ in a system with L levels would be computed as

$$\mathrm{Cost}(BJ, BK) = \sum_{i=1}^{L} W_i M(C_{s_i}, L_{s_i}, K_i, BJ, BK) \tag{5}$$

where function M provides the number of misses the model estimates for a given cache level and block. The code was later run on these machines, trying all the possible blocks whose sizes were divisors of the matrix dimensions in order to measure their execution times. Table 9 shows the difference between the execution times using the predicted optimal block and the best execution time expressed as a

Table 9
Difference between predicted and real optimal block execution times expressed as a percentage of the execution time of the real optimal block (N stands for the matrix size)

N	PC pII	PW 433au	O200 R10K	AP U-II
200	0.00	0.00	3.97	4.40
250	1.28	0.00	5.34	3.61
400	0.00	2.10	3.53	1.59
500	0.00	2.78	4.13	0.88
600	8.74	3.75	1.93	0.98

percentage of the latter. As we can see, the model proposes optimal or near optimal blocks in all the cases.

6.1. Optimal block selection

The good results achieved in the preceding set of experiments on real systems, together with the high speed of our model, confirmed that it was feasible to drive completely automatic optimizations using our tool. As an example we built a module to choose optimal block sizes (inside *Optimization Library* in Fig. 2). The estimation of the block size leading to the minimum execution time requires taking into account both the stalls due to cache misses and the computation cycles, which implies the need of a CPU model. As this is beyond the scope of our work, we took the CPU model of Delphi, as explained in Section 4. The CPU computation cycles predicted by Delphi were added to the stall cycles caused by the misses predicted by our model, to estimate the total number of execution cycles for a given code. The platform chosen for this set of experiments was an SGI Origin 200 for two reasons: the CPU parameters for the R10000 microprocessor required by Delphi were known; and a good compiler (MIPSpro version 7.3.1.1m), with many flags that allow control of the optimization and code generation process, was available.

The procedure for performing this experiment was the following: each code was first fed to the MIPSpro compiler in such a way that it decided where to apply blocking, as well as the size of the block, using the O3 optimization level. The resulting code was then analysed and rewritten by our framework, modifying the block sizes and the sentences related to them according to the predictions of our model. The set of block sizes evaluated by the tool for each dimension of size N were the original values proposed by the compiler as well as $\lceil N/i \rceil$ for $0 < i \leqslant 128$ (discarding those values that differed in less than three units). It should also be mentioned that the model was run without taking into account the relative position of the data structures, just using pure probabilities. It is obvious that this execution mode is less accurate, as it has less information to perform the modelling. This conservative approach was taken because our algorithm was not running in the MIPSpro compiler, so it really had no information at all about where it was going to locate the different data structures.

The experiment was applied to all the codes in Table 6, but the MIPSpro compiler only chose to apply blocking in three of them: *tomcatv*, *swim* and *hydro2d-filter*. We must point out that these codes can only benefit from reuse in the borders of the blocks, and not in the whole block, as is the case in the typical example of the matrix–matrix product. As a result, large improvements in the execution time cannot be expected by optimizing the size of such blocks. Moreover, the execution time of the loops modified in the last code is so small that it is very difficult to measure them alone meaningfully. This problem was overcome using the whole *hydro2d* program to make the measurements, just changing the block sizes in those places where the compiler chose to apply blocking.

Table 10 shows the percentage of improvement in execution time obtained when applying our tool with respect to that obtained with the MIPSpro compiler using the

Table 10
Improvement in the execution time for the analyzed codes

	Optimization level	
Benchmark	O2 (%)	O3 (%)
Tomcatv	1.23	1.82
Swim	0.00	1.25
Hydro2d	−0.60	0.18

O2 and O3 optimization levels. Excepting one case, our tool has always improved the execution times although, as expected, there are no big improvements, mainly due to the reasons previously mentioned. Nevertheless, we think that the improvements are quite good, taking into account that only one column and/or row of the blocks can be reused and that we are comparing it to the results of a good compiler.

There are several points that make it very difficult to compare our approach with the traditional algorithms for tile size selection [4,19]. First, such algorithms only focus on the memory hierarchy behavior, while our approach is the first one that, as far as we know, also takes into account CPU time. This happened to play an essential role, as ignoring it would lead to the choice of very small blocks that have maximum reuse, but whose management requires computing overheads that are too high. Another important point is that the referenced algorithms can consider just the parameters of one cache level, while our approach takes into account all the levels of the memory hierarchy simultaneously, in order to make the selection. Furthermore, works in [4,19] are specifically devoted to driving tile size optimization; in our case, tiling is just a particular application of a general purpose system for the prediction of the memory hierarchy behavior. Not less important is the fact that these algorithms look for block sizes that allow the keeping of the whole block in cache between reuses, but the blocks found in these codes only reuse their borders, so it is only interesting to keep the last row and/or column. Finally, these approaches only consider a block of one given matrix and pay little or no attention to its interaction with other data structures, which could even be other blocks generated by the dimension partitioning implied by blocking. Nevertheless, such interferences must be taken into account, particularly as the dimension tiling may simultaneously affect many arrays accessed in the same nesting (up to seven in these codes), giving place to many blocks interacting in the caches. This is no problem for our tool because it is oriented to a broader scope.

7. Conclusions

A fast and flexible analytical modelling technique to study the memory hierarchy performance has been validated and embedded in a compiler framework. The model is based on the concepts of a PME for each reference and nesting level, and the miss probability for an access to a given line. The miss probabilities are derived from the cache areas affected by the references between the reuses of a line, which are repre-

sented by means of the unified notation of the area vectors. This produces many structural advantages. For example, it allows the estimation of the number of misses for each reference and each given loop. The area vectors also allow the study of the relative contribution of each different data structure or even reference to the miss probability of the accesses associated to any reference. Other advantages which are not so obvious have been illustrated here, such as the ability to take into account the probability of hits due to reuses of data accessed in previous loops, which enables the analysis of non-perfect nestings and even whole programs. The model can be easily extended, thanks to its modularity, by developing the PMEs and area vectors associated to each new access pattern we need to study. These calculations may be performed either analytically or through simulations that can be sampled to reduce their computing requirements.

The systematization achieved in the model has allowed its implementation inside the Polaris parallelizing compiler, together with additional capabilities shown in Section 4. This has allowed the construction of a complete compiler framework, giving place to a powerful tool for system designers and programmers.

Validations performed through comparison with trace-driven simulations shows that although relatively large errors may arise in the number of predicted misses (when both the number of misses and the miss ratio are very small), such predictions are still very acceptable and the average accuracy of the model is very good. On the other hand, the time required by our model to make the predictions is very small in comparison with that required by simulation, and even in comparison with the compilation and execution time of the example codes. These properties make our model ideal to guide compiler optimizations. Thus, we applied it to choose optimal block sizes, as a case study, using the parameters of memory hierarchies of real systems. Successful results in a number of different hardware platforms were achieved and current commercial compilers were outperformed with real codes from the SPECfp95 suite.

References

[1] A. Lebeck, D. Wood, Cache profiling and the SPEC benchmarks: A case study, IEEE Computer 27 (10) (1994) 15–26.
[2] R. Uhlig, T. Mudge, Trace-driven memory simulation: A survey, ACM Computing Surveys 29 (2) (1997) 128–170.
[3] M. Zagha, B. Larson, S. Turner, M. Itzkowitz, Performance analysis using the MIPS R10000 performance counters, in: ACM (Eds.), Proc. Supercomputing'96 Conference, ACM Press and IEEE Computer Society Press, 1996, pp. 17–22.
[4] M.S. Lam, E.E. Rothberg, M.E. Wolf, The cache performance and optimizations of blocked algorithms, in: Proc. Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society, Santa Clara, CA, 1991, pp. 63–74.
[5] O. Temam, C. Fricker, W. Jalby, Cache interference phenomena, in: Proc. Sigmetrics Conference on Measurement and Modeling of Computer Systems, ACM Press, 1994, pp. 261–271.
[6] C. Cascaval, Compile-time performance prediction of scientific programs, Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2000.

[7] S. Ghosh, M. Martonosi, S. Malik, Cache miss equations: A compiler framework for analyzing and tuning memory behavior, ACM Transactions on Programming Languages and Systems 21 (4) (1999) 702–745.

[8] J.S. Harper, D.J. Kerbyson, G.R. Nudd, Analytical modeling of set-associative cache behavior, IEEE Transactions on Computers 48 (10) (1999) 1009–1024.

[9] X. Vera, J. Xue, Let's study whole-program behaviour analytically, in: Proc. 8th Int. Symposium on High-Performance Computer Architecture (HPCA8), 2002, pp. 175–186.

[10] S. Chatterjee, E. Parker, P. Hanlon, A. Lebeck, Exact analysis of the cache behavior of nested loops, in: Proc. ACM SIGPLAN'01 Conference on Programming Language Design and Implementation (PLDI'01), 2001, pp. 286–297.

[11] B.B. Fraguela, R. Doallo, E.L. Zapata, Automatic analytical modeling for the estimation of cache misses, in: Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'99), 1999, pp. 221–231.

[12] K.S. McKinley, O. Temam, A quantitative analysis of loop nest locality, in: Proc. Seventh Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), ACM Press, Cambridge, Massachusetts, 1996, pp. 94–104.

[13] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, P. Tu, Parallel programming with Polaris, IEEE Computer 29 (12) (1996) 78–82.

[14] J. Ferrante, V. Sarkar, W. Thrash, On estimating and enhancing cache effectiveness, in: U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), Proc. of the Fourth International Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, vol. 589, Intel Corp., Springer-Verlag, Santa Clara, CA, 1991, pp. 328–343.

[15] J. Blieberger, T. Fahringer, B. Scholz, Symbolic cache analysis for real-time systems, Real-Time Systems 18 (2/3) (2000) 181–215.

[16] J. Sanchez, A. Gonzalez, Analyzing data locality in numeric applications, IEEE Micro 20 (4) (2000) 58–66.

[17] B.B. Fraguela, R. Doallo, E.L. Zapata, Modeling set associative caches behavior for irregular computations, ACM Performance Evaluation Review (Proc. SIGMETRICS/PERFORMANCE'98) 26 (1) (1998) 192–201.

[18] R. Saavedra, A. Smith, Measuring cache and TLB performance and their effect on benchmark run times, IEEE Transactions on Computers 44 (10) (1995) 1223–1235.

[19] G. Rivera, C.-W. Tseng, A comparison of compiler tiling algorithms, in: Proc. of 8th Int. Conf. on Compiler Construction, Lecture Notes in Computer Science, vol. 1575, 1999, pp. 168–182.