Accurate prediction of the behavior of multithreaded applications in shared caches

Diego Andrade[a,*], Basilio B. Fraguela[a], Ramón Doallo[a]

[a]*Dept. of Electronics and Systems, University of A Coruña, Spain. Tel.: +34-981-167-000 ext. 1298, Fax: +34-981-167-160*}

Abstract

Multicores are the norm nowadays and in many of them there are cores that share one or several levels of cache. The theoretical performance gain expected when several cores cooperate in the parallel execution of an application can be reduced in some cases by a cache access bottleneck, as the data accessed by them can interfere in the shared cache levels. In other cases the performance gain can be increased due to a greater reuse of the data loaded in the cache. This paper presents an analytical model that can predict the behavior of shared caches when executing applications parallelized at loop level. To the best of our knowledge, this is the first analytical model that tackles the behavior of multithreaded applications on realistic shared caches without requiring profiling. The experimental results show that the model predictions are precise and very fast and that the model can help a compiler or programmer choose the best parallelization strategy.

1. Introduction

Nowadays, multicore processors have become the norm, and their design often presents one or more levels of cache shared by several cores [1]. This way, the interference of the accesses generated by the different cores in the shared cache levels can reduce the speedup of parallel applications. On the other hand, multithreaded applications can experience superlinear speedups when their threads share data in these caches. Analytical models have proved to be an useful tool capable of providing reasonable estimations of the cache performance [2, 3, 4] in order to guide successfully compiler optimizations [5, 6]. Unfortunately, they usually have a limited scope of application and, in fact, the prediction of the performance of multithreaded applications in realistic caches is still an open problem.

This paper presents an accurate analytical model of the behavior of shared caches during the execution of multithreaded applications. This model is an extension of the Probabilistic Miss Equations (PME) model [2] of the cache behavior. The extension proposed in this work considers the most common pattern of parallelization in shared memory of regular and scientific applications, which distributes blocks of iterations of parallel loops among several threads.

There are two main differences in the behavior of these applications with respect to single-threaded ones: (1) The reuse of cache lines is also possible between accesses performed by

*Corresponding author
Email addresses: diego.andrade@udc.es (Diego Andrade), basilio.fraguela@udc.es (Basilio B. Fraguela), doallo@udc.es (Ramón Doallo)

threads that run in parallel in different cores that share the same cache. (2) The attempts of reuse in the shared cache are interfered not only by the accesses from the same thread but also by the accesses performed by other parallel threads. As a result of these facts, the behavior of the shared caches present in multicores can be very different for the sequential and the parallelized version of a code. Also, the cache behavior of a parallel code can largely vary depending on its scheme of parallelization. This way, a good model of the behavior of shared caches is needed in order to take proper decisions on the parallelization of codes and drive the optimization process of multithreaded applications. The validation shows that our model reliably predicts the behavior of realistic shared caches during the execution of parallelized codes, thus being a very suitable tool to play that role. Despite the growing importance of multicore systems and the critical role of the memory hierarchy in performance, we do not know of any other model in the bibliography that can predict the behavior of shared caches during the execution of multithreaded codes without resorting to a previous profiling or simulation.

The rest of this paper is organized as follows. Section 2 defines the scope of application of our model. The model has two parts: the construction of the PMEs, which is described in Section 4, and the miss probability calculation, whose details are explained in Section 5. Section 6 shows the experimental results. Section 7 discusses the related work and Section 8 presents conclusions and future work.

2. Scope of application

Our model supports cache hierarchies composed of a number of set associative cache levels with a Least Recently Used (LRU) replacement policy. One or several of these cache levels can be shared by any number of cores. In each level, the cache size, line size and associativity can be fixed arbitrarily. This is, by far, the most common situation in modern multicore architectures.

The model uses also as input the source code to analyze, which can be composed of an arbitrary number of perfect or non-perfect loop nests, and references which can be located in any nesting level. Each data structure can be accessed by any number of references, and the indexing of the data structures must be done using affine functions of the loops control variables. The number of iterations of the loops must be known to perform the analysis.

Inter-routine cache effects are modeled using inlining, either symbolic or actual. Regarding conditional statements, they may appear in the code analyzed when they guard accesses to registers or to the latest data item accessed before the branch, as in these cases the data-dependent flow does not alter the cache behavior. This scope of application covers a large number of programs and kernels. In fact, it is the same one as in [2], where it sufficed to model complete SPECfp95 and Perfect Benchmarks codes, or at least their more significative and time-consuming routines. The model presented in this paper adds the possibility that some of the loops may be parallelized, provided they are not nested inside another parallelized loop, using any method to extract loop-level parallelism, such as for example the OpenMP [7] `for` pragma. Finally, it is assumed that the threads that participate in the parallel execution of a loop begin their execution at about the same time.

The distribution of the iterations of parallelized loops is chosen to be block-cyclic, that is, blocks of b_i consecutive iterations of loop i are distributed among t_i threads. This is a very generic approach, as the cyclic distribution is a sub-case where the block size is one, and the consecutive distribution is also a sub-case where the block size is N_i/t_i, N_i being the number of iterations of the loop. Then, block cyclic distribution of iterations is supported using both static and dynamic

```
function PMEmodel(sourceCode, C_s, L_s, k)  {
1    init_model(C_s, L_s, k)
2    misses=0
3    foreach R = reference in sourceCode     {
4        misses = misses + F_R0(RD)
5     }
6    return misses
7 }
```

Figure 1: General algorithm followed by the PME model

scheduling policies: in the static one blocks of iterations are distributed cyclically among the threads, while in the dynamic one blocks of iterations are assigned on demand to idle threads.

The extension of the model presented in this paper requires exactly the same information as the model in [2], with only two additions: the number of threads used in the parallelization and the block size used to distribute the iterations of the parallelized loops among the threads. The pieces of information required as an input to the model can be gathered through different means. For example, the number of iterations a loop can often be inferred from the code, particularly after applying standard compiler techniques such as inlining and constant propagation. It has already been proved in [8] that the information required by the sequential version of the model can be automatically extracted from regular codes like the ones considered in this paper using a compiler. Also, a compiler that could gather the data required by the PME model even in irregular codes, where this information is sometimes difficult to obtain, was presented in [9]. When this is not the case, missing information can be obtained from an analysis of the input data that can be provided by the user, for example by means of compiler directives recognized by the tool that applies the model, or obtained by means of runtime profiling.

3. The Probabilistic Miss Equations Model

The PME model [2] estimates the number of cache misses that take place during the execution of a code. The source code and the cache configuration (cache size C_s, line size L_s and associativity k) must be provided as inputs to the model. This is reflected in the arguments of the function shown in Figure 1, which is an overview of the algorithm followed by the PME model. When there are several cache levels, the behavior of each cache level is modeled independently. The predictions are precise even although the accesses do not take place directly on the shared cache, as they are filtered previously by one or two upper levels of private caches. In previous works [2, 6, 10], it has been shown that if the n-th cache level is modeled directly using the PME model and ignoring the filtering of the upper cache levels, the predictions are still accurate. Once the number of cache misses generated by each cache level has been predicted, its behavior in the overall cache performance can be estimated by weighting the number of misses with the cost of a cache miss in that level.

Loops are numbered from the outermost one and proceeding inwards starting at 0 and they are normalized to have step one. For each static reference R in the code, and loop i enclosing it, the model creates an estimator of the number of misses that a reference R generates during one complete execution of the loop called Probabilistic Miss Equation (PME), F_{Ri} (lines 3-5 of Figure 1). Finally, the PMEmodel returns the number of misses generated by the code during its execution (line 6 of Figure 1).

As caches are assumed to be initially empty, memory accesses only result in cache hits when they reuse a cache line brought to the cache by a previous access. The model predictions are

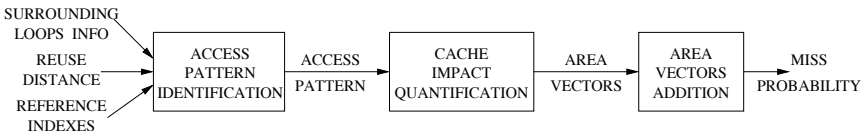


Figure 2: Procedure for estimating miss probabilities from the code

based in the calculation of the probability that each reuse attempt results in a miss. This probability depends on the cache footprint of the data accessed during the reuse distance, i.e. the piece of code executed since the previous access to the considered line.

The PME F_{Ri} classifies the accesses of R during the execution of loop i depending on whether they can exploit reuse with respect to other accesses that took place during such execution or not. The accesses that cannot exploit reuse within the loop may reuse a line already accessed in a previous iteration of an outer loop, or in a preceding loop nest. As a result these accesses may end up being compulsory misses, or they can enjoy reuse distances that are out of the scope of the loop being analyzed. Namely, their reuse distance could be a given number of iterations of an outer loop, plus/or additionally the execution of loop nests preceding the one being analyzed. This is the reason why each PME has an input parameter RD which is the reuse distance of the first-time accesses to lines within its loop (parameter RD passed to the function F_{R0} in line 4 of Figure 1). The remaining accesses can exploit reuse within the loop, as they access lines that have been brought to the cache previously in the loop, or they may end up being interference misses. Normally, each given line can be reused with different reuse distances, that is, different portions of code are executed in between different attempts to reuse the line. In the case of references found in loop nests, which is the scope of the PME model, each loop enclosing a reference gives place to a different reuse distance, which can be measured in terms of loop iterations, that (possibly) characterizes some of the reuses not captured by the inner loops.

This way, our model estimates the number of misses generated by a reference by exploring the loops that enclose it from the innermost one to the outermost one. In each loop the model builds a partial PME that adds information about the reuses whose reuse distance is associated with that loop. That PME relies on the PME of the immediately inner level to model the behavior of that inner level. The reason is that each PME discovers reuse distances between accesses that take place in different iterations of the analyzed nesting level i, thus, the reuse distances associated to the studied reference in inner loops are discovered by the PME associated to that inner level. In the innermost level containing the reference, the recursion is finished by replacing the call to the PME of the inner level by the calculation of the miss probability associated to the reuse distance. Thus, the PME associated with the outermost loop in a nest takes into account all the reuses, and its evaluation yields the number of misses generated by the reference during the execution of the loop nest. Section 4 explains the construction of the Probabilistic Miss Equations Model. Now, Section 3.1 gives an overview of the method used to calculate the miss probability associated to a given reuse distance, which is a key component of our model.

Figure 2 shows the steps the PME model follows to derive the miss probability associated to a given reuse distance.

3.1. Miss Probability Estimation

The PME model is probabilistic as it uses the probability that each access results in a miss to calculate the number of misses. In a k-way set associative cache following a LRU replacement

policy, an access to a cache line results in a miss if it is a first-time access or if since the previous access to that line, k or more different lines mapped to the same cache set have been accessed. As a result, the probability of miss in a non-first access is equal to the probability that a cache set has received k or more lines during the reuse distance. Figure 2 shows the three steps the PME model follows to derive the miss probability associated to a given reuse distance:

(1) Access pattern identification: the access patterns followed by the references involved in the reuse distance and the parameters that characterize them are inferred from the references indexing functions and the shape of the loops that enclose them. The PME model represents each pattern as a function whose output is a mathematical representation of the footprint of the access pattern on the cache. There is one function per each typical access pattern (sequential access, access with constant stride, etc.), and its arguments provide the quantitative characterization of the access pattern.

(2) Cache impact quantification: the functions obtained in the preceding step are turned into a mathematical representation of the impact of these access pattern on the cache. This mathematical representation, called area vector, contains the probability that a cache set has received a given number of lines from the pattern.

(3) Area vectors addition: the area vectors for the different access pattern are combined in order to calculate a global area vector, that represents the combined impact of all these patterns on the cache.

Section 5 explains how the PME model follows these steps. Once they are completed, the final interference probability is estimated as component 0 of the global area vector associated with the analyzed reuse distance, since it contains the ratio of sets that received k or more lines during the reuse distance, which is conversely the probability a given set has received k or more lines.

4. Probabilistic Miss Equations construction

One of the key components of the PME model are Probabilistic Miss Equations, as seen in Section 3. A different PME is built per reference and per nesting level in which this reference is enclosed in order to calculate the number of misses associated to that reference. The structure of the PME $F_{Ri}(\mathrm{RD})$ is defined depending on whether the loop i is parallelized or not as

$$F_{Ri}(\mathrm{RD}) = \begin{cases} Fs_{Ri}(\mathrm{RD}) & \text{if loop } i \text{ is not parallelized} \\ Fp_{Ri}(\mathrm{RD}) & \text{if loop } i \text{ is parallelized} \\ MissP(\mathrm{RD}) & \text{if } i = Z + 1 \end{cases} \quad (1)$$

$MissP(\mathrm{RD})$ is the miss probability associated to a reuse distance RD, and Z is the innermost loop containing the reference. When $i = Z + 1$, the PME stands for the behavior of reference R in a single access in the innermost loop containing R. The probability that access results in a miss is equal to the miss probability associated to the input reuse distance $MissP(\mathrm{RD})$. Section 5 explains how to compute miss probabilities.

4.1. PMEs for intra-reference reuse in the sequentially executed case

When loop i is not parallelized, the PMEs introduced in [2] are used. We explain here for completeness the PME for references that only carry reuse with themselves in the loop analyzed, that is, which can only reuse lines that have been accessed by them, and not by any other reference, in the considered loop. We call this situation intra-reference reuse, as the reference reuses

b_i	Block size for loop i in cyclic distribution
$D_{\mathrm{A}j}$	size of the j-th dimension of array A. Dimensions are numbered from left to right starting at 0. In a 2D matrix, the row index is dimension 0 and the column index dimension 1.
$d_{\mathrm{A}j}$	cumulative size of the j-th dimension of array A, $d_{\mathrm{A}j} = \prod_{i=j+1}^{N} D_{\mathrm{A}i}$
k	Associativity
L_s	Line size (in elements)
$L_{Ri}(Nits)$	# of iterations in a set of $Nits$ consecutive iterations of loop i that cannot exploit spatial/temporal locality
$MissP(RD)$	Miss probability associated to reuse distance RD
N_i	# of iterations of loop i
SOL	Set of lines
S_{Ri}	stride that reference R has with respect to loop i
t_i	# of threads among which the iterations of loop i are distributed

Table 1: Notation summary

lines it accessed in previous iterations of the considered loop. The PMEs for references that carry reuse with other references, i.e., for inter-reference reuse, can be found in [2]. Those PMEs allow the modeling of reuse both among references located in the same loop nest and among references located in different loop nests.

In the innermost loop that contains a reference, the number of iterations in which the reference cannot exploit reuse within the loop is equal to the number of different lines it accesses. When the loop studied is not the innermost one containing the reference, the iterations of the loop define sets of lines (SOL) accessed by R in the inner loops. For example, let us consider a bi-dimensional $M \times N$ array is accessed by columns (that is, the innermost loop of the access sweeps through the M rows of a given column) in a code in C. In the analysis of the outer loop that controls the column index of the reference, each iteration of this loop is associated to the access to the set of lines that hold the elements of a column of the matrix. As C stores matrices in row-major order, if $N \geq L_{\mathrm{s}}$, where L_{s} is the cache line size measured in elements, which is the most usual situation, each set of lines will be made up of M different lines. In what follows we will talk in general about sets of lines (SOLs), in the understanding that in the innermost loop each one of these sets consists of a single line.

The number of iterations in a set of $Nits$ consecutive iterations of loop i in which R cannot exploit either spatial or temporal locality because it accesses a new SOL is

$$L_{Ri}(Nits) = 1 + \left\lfloor \frac{Nits - 1}{\max(L_{\mathrm{s}}/S_{Ri}, 1)} \right\rfloor , \tag{2}$$

S_{Ri} being the stride that reference R has with respect to loop i. These terms and others that appear along the paper are summarized in Table 1. This stride S_{Ri} is a constant, since either the loop control variable (I_i) does not index reference R, or the index we are considering is an affine function of $\mathtt{I}_i$. In the former case, trivially $S_{Ri} = 0$. In the latter case $S_{Ri} = \alpha_{Rj} \times d_{\mathrm{A}j}$, where j is the dimension whose index depends on $\mathtt{I}_i$; the scalar that multiplies I_i in the affine function is α_{Rj}, and $d_{\mathrm{A}j}$ is the cumulative size[1] of the j-th dimension of the array A referenced by

[1] Let A be an N-dimensional array of size $D_{\mathrm{A}1} \times D_{\mathrm{A}2} \times \ldots D_{\mathrm{A}N}$, we define the cumulative size for its j-th dimension as $d_{\mathrm{A}j} = \prod_{i=j+1}^{N} D_{\mathrm{A}i}$

```
#pragma omp parallel \
  private (TMP,R,ci,cj,ck,cjj,ckk) \
  for nowait schedule(static)
for(ck = 0; ck < 8192; ck += 32)//L0
  for(cj = 0; cj < 8192;cj += 32) {// L1
    for(ci = 0; ci < 8192; ci++) {//L2

      for (ckk = ck; ckk < min(8192, ck+32); ckk++)//L3.1
        for (cjj = cj; cjj < min(8192, cj+32); cjj++)//L4.1
          TMP[cjj-cj][ckk-ck] = B[ckk][cjj];

      for (cjj = cj; cjj < min(8192, cj+32); cjj++) {//L3.2
        ACC = C[ci][cjj];
        for (ckk = ck; ckk < min(8192, ck+32); ckk++) //L4.2
          ACC += TMP[cjj-cj][ckk-ck] * A[ci][ckk];
        C[ci][cjj] = ACC;
      }
    }
}
```

Figure 3: Matrix product with tiling and buffering (MATMULTB in Table 4)

R. Dimensions are numbered from left to right starting at zero. In the case of a bi-dimensional array, the row index is dimension 0, while the column index is dimension 1. The model assumes, without loss of generality, that multidimensional arrays are stored by rows.

If $L_{Ri}(N_i) < N_i$, N_i being the number of iterations of loop i, then there are $N_i - L_{Ri}(N_i)$ iterations of loop i in which R accesses the same SOL accessed in the previous iteration of this loop. It is necessarily the same SOL, since there is a constant stride S_{Ri} between the set of addresses accessed in consecutive iterations. Thus, from the point of view of the cache behavior of the accesses of reference R there are two kinds of iterations of loop i:

- In $L_{Ri}(N_i)$ iterations a new SOL is accessed by R, giving place to first-time accesses in this loop. The potential reuse distance for such accesses is unknown in this nesting level and is the input RD to the PME F_{Ri}.
- In the other $N_i - L_{Ri}(N_i)$ iterations the reuse distance for the accesses of R is one iteration of loop i, which we denote by $\text{Iter}_i(1)$.

According to the preceding explanation, the PME $Fs_{Ri}(\text{RD})$ that computes the number of misses generated by reference R during the execution of a sequential loop i is

$$\begin{aligned} Fs_{Ri}(\text{RD}) = & L_{Ri}(N_i) \times F_{R(i+1)}(\text{RD}) + \\ & (N_i - L_{Ri}(N_i)) \times F_{R(i+1)}(\text{Iter}_i(1)) \end{aligned} \tag{3}$$

The first term of the PME captures the behavior of those iterations where the reference generates accesses to SOLs accessed for the first time in this execution of the loop. The reuse distance for these iterations is given by the input parameter of the PME, RD. The second term captures the remaining iterations as potential reuse attempts where the reuse distance is always one iteration of loop i, $\text{Iter}_i(1)$.

Example 1. *Calculation of the PME associated to a reference*: Let us consider the C code for the product of 8192×8192 matrices optimized using tiling and buffering in Figure 3 (MATMULTB

in Table 4) and a shared direct-mapped cache which can store $C_s = 1024$ elements, with lines of $L_s = 8$ elements. A tile size of $ts = 32$ is selected for two of the dimensions of the problem. The iterations of the parallel loop in the code are distributed by blocks of $b_0 = 32$ iterations among 8 threads.

Let us analyze the behavior of reference $R \equiv$`C[ci][cjj]`. The analysis starts in the innermost loop containing the reference, loop labeled as L3.2 at level 3 in Figure 3. This loop is sequentially executed, so, according to Equations (1) and (3), PME FP_{R3} is built as Fs_{R3}. Equation (2) is used for the calculation of $L_{R3}(N_3 = 32)$: the stride $S_{R3} = \alpha_{R1} \times d_{C1}$ because the loop index `cjj` indexes dimension 1 of matrix `C`, the value $\alpha_{R1} = 1$ and $d_{C1} = 1$ since in C language matrices are stored by rows; thus, $S_{R3} = 1$ and $L_{R3} = 1 + \left\lfloor \frac{32-1}{\max\{8/1,1\}} \right\rfloor = 4$. Finally, Equation (3) yields

$$Fs_{R3}(\mathrm{RD}) = 4 \times F_{R4}(\mathrm{RD}) + 28 \times F_{R4}(\mathrm{Iter}_3(1))$$

This PME means that during a full execution of the loop L3.2 R accesses 4 different lines which are reused in the remaining 28 iterations with a reuse distance of one iteration of the loop. This is sensible, since we can see that during the execution of this loop the reference accesses 32 consecutive elements, which are spread on 4 different lines because each line can hold $L_s = 8$ elements. This way, in four of the iterations the reference accesses one of these lines form the first time during this execution of the loop, while the other 28 iterations try to reuse the line accessed in the previous iteration. As for F_{R4}, since $Z = 3$ is the innermost loop containing the reference $F_{R4}(RD) = MissP(RD)$ according to Equation (1). The calculation of $MissP$ will be explained in Section 5. The two immediately outer loops (L2 and L1) are also sequential, thus following Equation (1), $F_{R2}(RD) = Fs_{R2}(RD)$ and $F_{R1}(RD) = Fs_{R1}(RD)$. The development of these two PMEs is not included here due to space reasons, as it is analogous to the one of loop 3. Finally, the outermost loop is parallelized, so Equation (1) refers us to $Fp_{Ri}(\mathrm{RD})$, whose modeling is introduced in Section 4.2.

4.2. PME for intra-reference reuse in a parallelized loop

We describe now the construction of the PME for references that only carry reuse with themselves in a parallelized loop. Appendix A deals with the PMEs for references that carry reuse with other references, located in the same loop nest, in a parallelized loop. The modeling of references that carry reuse with other references located in different loop nests does not suffer variations with respect to the strategy already described in [2]. When a loop i is parallelized, the model assumes that its N_i iterations are distributed in blocks of b_i consecutive iterations among the t_i threads available. This way, $\left\lfloor \frac{N_i}{b_i} \right\rfloor$ blocks of b_i iterations are assigned to the t_i threads. If N_i is not divisible by b_i, one of the threads executes additionally $N_i \mod b_i$ iterations. The parallel execution of the loop is organized in parallel iterations where a number of threads (typically t_i) are executing one serial iteration of the loop each concurrently.

The first term of the PME includes the $L_{Ri}(N_i)$ SOLs accessed during the execution of loop i whose associated reuse distance is the input reuse distance RD to the PME. If the loop were executed sequentially, the remaining iterations would attempt to reuse a SOL accessed in the previous iteration of the loop, as it was seen in Equation (3). However, the parallel execution of the loop opens the possibility of inter-thread reuse, that is, a SOL can be accessed by several threads. Inter-thread reuse may appear in one of two situations, if:

- Situation `Inter-thread reuse-1`: b_i is neither a multiple nor a divisor of the line size L_s and I_i, the loop index of loop i, indexes R

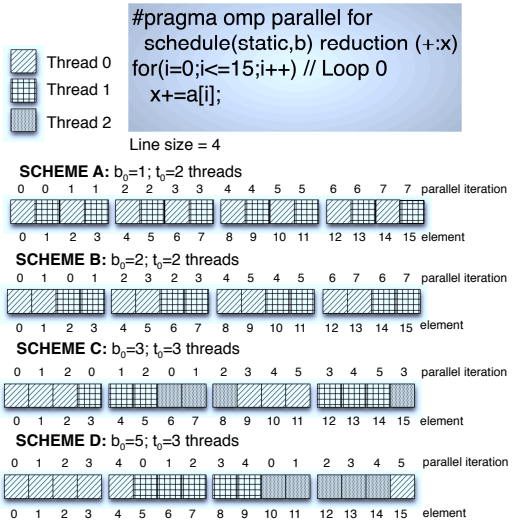


Figure 4: Distributions of the iterations of a parallel loop among the threads

- Situation `Inter-thread reuse-2`: $b_i \times S_{Ri} < L_s$.

Let us see an example where both situations arise.

Example 2. Figure 4 considers a simple loop that is parallelized using an OpenMP pragma that distributes its iterations among a number of threads in groups of b_i consecutive iterations. The four schemes in the figure represent the distribution on the cache of the 16 elements of array `a` accessed in the loop. The schemes consider a static scheduling policy which cyclically assigns the groups of b_i iterations to the available threads. Each cache line can store 4 elements of the array, and the elements of each cache line are grouped in the scheme. The elements (conversely, iterations of the loop) processed by each thread are represented using a different pattern. On top of each element there is a number that represents the parallel iteration in which the element is processed. Each parallel iteration is labeled with an integer number starting at 0. This way, all the elements labeled with the same number are processed concurrently by different threads. The number under each element is its index in the array. Each scheme in the figure indicates the block size, b_i and the number of threads t_i it considers. The distribution of the serial iterations among the threads for one of the schemes (scheme C) is shown in Figure 5. In scheme C ($b_0 = 3$ and $t_0 = 3$), there are $\left\lfloor \frac{16}{3} \right\rfloor = 5$ blocks of $b_0 = 3$ iterations and one of $16 \mod 3 = 1$ iteration to distribute cyclically among the threads. Each row contains for a given parallel iteration, which serial iteration executes each thread.

In schemes A and B the situation `Inter-thread reuse-2`, that gives place to inter-thread reuse attempt, arise. In both schemes, $b_0 \times S_{R0} < L_s$ as $S_{R0} = 1, L_s = 4$, being $b_0 = 1$ for scheme A and $b_0 = 2$ for scheme B. This situation makes that several threads access different elements from the same cache line. For example, we can see in Figure 4 that in Scheme A at parallel iteration 0, both threads are accessing two consecutive elements (0 and 1 respectively) from the same cache line. Similarly, in Scheme B at parallel iteration 0, elements 0 and 2 are accessed respectively by both threads. The PME model assumes that the progress of all the threads along

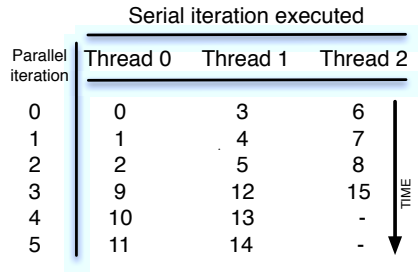


Figure 5: Correspondence among threads, parallel iterations and serial iterations for scheme C of Figure 4

the iterations is similar. This way, when two threads access simultaneously memory positions located at the same cache line, only the accesses of one of the cores is going to produce cache misses as the other cores are going to take advantage of the line that has been already loaded on the cache. Thus, in both schemes this cache line is reused by one of both threads, with a reuse distance of 0 parallel iterations.

In scheme C, both situations, `Inter-thread reuse-1` and `Inter-thread reuse-2`, are present at the same time, $b_0 \times S_{R0} < L_s$ as $b_0 = 3$, $S_{R0} = 1$ and $L_s = 4$, while $b_0 = 3$ is neither a multiple nor a divisor of the line size $L_s = 4$. These situations causes that, like in schemes A and B, different elements from the same cache line are accessed by different threads. For example, at parallel iteration 0, elements 0 and 3 are accessed simultaneously by threads 0 and 1, thus, this cache line is reused by one of both threads, with a reuse distance of 0 parallel iterations. No all the inter-thread reuses happen in the same parallel iteration. For example we can see that at parallel iteration 2, thread 2 accesses element 8, while at parallel iteration 3 thread 0 accesses element 9 that belongs to the same cache line. In this case, one of the threads is attempting to reuse a cache line that was accessed in the previous parallel iteration by the other thread. This attempt of reuse has thus a reuse distance of one parallel iteration.

Finally, in scheme D only situation `Inter-thread reuse-1` takes place, as $b_0 = 3$ is neither a multiple nor a divisor of the line size $L_s = 4$. For example, thread 0 accesses element 4 at parallel iteration 4, while the other three elements from the same cache line (elements 5, 6 and 7) has been accessed by thread 1 at parallel iterations 0, 1 and 2, respectively. The most recent access to the cache line, when thread 0 accesses it for the first time in the 4th parallel iteration, was done by thread 1 in parallel iteration 2, thus, in this case the reuse distance is 2 parallel iterations ■

The modeling of these situations is covered now. During the execution of the loop, a total of $L_{Ri}(N_i)$ SOLs are accessed. Now, these iterations are distributed among t_i threads in $\lfloor N_i/b_i \rfloor$ groups of b_i and eventually one group of $N_i \mod b_i$. The number of SOLs accessed by several threads is equal to the difference between the number of first time accesses to SOLs, considering isolatedly the iterations executed by each thread, and the number of accessible SOLs. This value nrs is calculated as

$$\text{nrs} = \left(L_{Ri}(b_i) \times \left\lfloor \frac{N_i}{b_i} \right\rfloor \right) + L_{Ri}(N_i \mod b_i) - L_{Ri}(N_i) \tag{4}$$

These SOLs are accessed in the first iteration of one of the blocks of iterations assigned to a given thread and can be reused in an iteration that belongs to the previous block, assigned to

another thread, and which is run in a later parallel iteration. There are $b_i - 1$ iterations between the first iterations of two consecutive blocks, and on average the number of reuses within the same thread of a SOL shared by two blocks (threads) is $\lfloor L_s/S_{Ri} \rfloor - 2$ as it is the mean of all the possible number of reuses of this kind $(1 \ldots (L_s/S_{Ri}) - 1)$. Subtracting, the reuse distance rd for the subsequent access, expressed in parallel iterations of loop i, is calculated as

$$rd = max(b_i - \lfloor L_s/S_{Ri} \rfloor + 1, 0) \tag{5}$$

Example 3. *Calculation of* rd: Let us consider the scheme D in Figure 4, in which $t_0 = 3$ threads participate in the parallel execution of the loop distributing its iterations in chunks of $b_0 = 5$ consecutive iterations per thread. During its execution, all the lines but the first one are accessed by two threads. This way, one of the threads may reuse the lines brought to the cache by another one. The number of these reuses is nrs $= \left(L_{Ri}(5) \times \left\lfloor \frac{16}{5} \right\rfloor\right) + L_{Ri}(16 \mod 5) - L_{Ri}(16) = (2 \times 3) + 1 - 4 = 3$ (see Equation (4)). The average reuse distance of these reuse attempts is rd $= max\left(5 - \left\lfloor \frac{4}{1} \right\rfloor + 1, 0\right) = 2$ (see Equation (5)) parallel iterations of the loop. In Figure 4 we can see that these 3 reuses are between:

- (1) The accesses to element 4 by thread 0 and element 5 by thread 1 at parallel iterations 4 and 2, respectively (reuse distance is 2).
- (2) The accesses to element 9 by thread 1 and element 11 by thread 2 at parallel iterations 4 and 1, respectively (reuse distance is 3).
- (3) The accesses to elements 14 by thread 2 and element 15 by thread 0 at parallel iterations 4 and 5, respectively (reuse distance is 1) ■

If $b_i \times S_{Ri} < L_s$ several threads access simultaneously the same line (or SOL), so the reuse distance of these reuses is 0 iterations of the loop. As we saw, the N_i iterations of the loop are distributed in groups of b_i iterations. This way, the N_i iterations are divided into groups of $t_i \times b_i$ iterations where each thread executes a block of b_i iterations. It must be calculated how many of these groups of $t_i \times b_i$ iterations fit in one SOL (ngrt) and how many threads can access concurrently a SOL (nst). Once we have calculated these values, we can calculate that there are

$$\text{rso} = \text{ngrt} \times max((\text{nst} - 1), 0) \times (b_i - 1) \times L_{Ri}(N_i) \tag{6}$$

iterations where the reuse distance is 0, as the same SOL is being accessed simultaneously by other threads. For each group, one thread of the nst threads whose b_i iterations fit in one SOL is not considered in this term because it is regarded as the first one that accesses the SOL. For the other threads, for each group of b_i iterations, only the last $b_i - 1$ iterations are considered here, as the reuse of the first iteration of each group of b_i iterations has already been considered in the term nrs (see Equation (4)). Finally, such reuses happen in each one of the $L_{Ri}(N_i)$ SOLs that reference R defines in loop i.

Regarding the calculation of ngrt and nst, if $S_{Ri} = 0$, ngrt $= \frac{N_i}{t_i \times b_i}$ and nst $= t_i$ because in the N_i iterations of the loop the same SOL is accessed. When $S_{Ri} > 0$, ngrt $= min\left(\frac{N_i}{t_i \times b_i}, \frac{L_s}{t_i \times b_i \times S_{Ri}}\right)$ and nst $= min\left(\frac{L_s}{b_i \times S_{Ri}}, t_i\right)$. These two cases are summarized in the following expressions:

$$\text{ngrt} = min\left(\frac{N_i}{t_i \times b_i}, \frac{L_s}{t_i \times b_i \times S_{Ri}}\right) \tag{7}$$

$$\text{nst} = min\left(\frac{L_s}{b_i \times S_{Ri}}, t_i\right) \quad (8)$$

Example 4. *Calculation of* rso: Considering schemes A, B and C of Figure 4, as $b_i \times S_{Ri} < L_s$ several threads access simultaneously the same SOL. For example, in scheme C, rso = 3 as:

- (1) At parallel iteration 0, threads 0 and 1 access elements 0 and 3, which belong to the same cache line.
- (2) At parallel iteration 1, threads 1 and 2 access simultaneously elements 4 and 7, which belong to the same cache line.
- (3) At parallel iteration 3, threads 1 and 2 access elements 12 and 15, which belong to the same line.

This can be also visualized for scheme C in Figure 5.

Regarding scheme B, one group of $t_0 \times b_0$ iterations fits in one cache line. Thread 1 reuses the lines that are accessed concurrently by thread 0. The first access to each line by thread 1 is accounted by the term nrs with associated reuse distance rd = 0. The second access to each line by thread 1 is considered by term rso = ngrt $\times max((\text{nst} - 1), 0) \times max((2 - 1), 0) \times L_{R0}(16)$ (see Equation (6)). As ngrt = $min\left(\frac{16}{2\times2}, \frac{4}{2\times1\times2}\right) = 1$ group (see Equation (7)) of $t_0 \times b_0 = 4$ iterations, and the iterations of nst = $min\left(\frac{4}{2\times1}, 2\right) = 2$ (see Equation (8)) threads fits in one cache line, then, according to Equation (6), rso = $1 \times (2 - 1) \times 1 \times 4 = 4$. These 4 reuses take place due to the simultaneous access to elements 1 and 3, 5 and 7, 9 and 11, and finally 13 and 15, by threads 0 and 1, respectively. Notice that the simultaneous access to elements 0 and 2, 4 and 6, 8 and 10, and finally 12 and 14, which correspond to the reuse of the first iteration of each group of b_i iterations , have already been considered in the term nrs (see Equation (4)) ■

If more than one group of $t_i \times b_i$ iterations fit in a SOL (ngrt > 1), the first thread that accesses a SOL may access that same SOL again in a different block of iterations associated to that same thread. In these iterations, the thread is reusing the same SOL it accessed in the immediately previous iteration. However, these iterations were initially included in the term nrs, which corresponds to accesses with a reuse distance of rd iterations. The number of these iterations is

$$\text{ngrtC} = max((\text{ngrt} - 1), 0) \times L_{Ri}(N_i) \quad (9)$$

This fact must be taken into account by subtracting these iterations from nrs and adding them to a term where the reuse distance is one iteration of the loop.

Example 5. *Calculation of* ngrtC: Considering scheme A of Figure 4, nrs = 12 and rd = 0. But the second access to each line by thread 0 has a reuse distance of one iteration of the loop. The number of these iterations is calculated using Equation (9) as ngrtC = $max((2-1), 0) \times L_{R0}(16) = 4$, because ngrt = $min\left(\frac{16}{2\times1}, \frac{4}{2\times1\times0}\right) = 2$. We are assuming that all the threads progresses at the same speed along the iterations of the parallel loop. In addition, we assume that, when both threads access simultaneously memory positions located at the same cache line, thread 1 takes advantage of the line loaded in the cache by thread 0, these ngrtC = 4 iterations correspond to the accesses of elements 2, 6, 10 and 14 of the array. As it is shown in Figure 4, these iterations access the same cache line brought to the cache by the simultaneous access performed by threads 0 and 1 in the immediately previous parallel iteration and thus their reuse distance is one parallel iteration, not the rd = 0 iterations computed for nrs ■

The iterations not accounted in the previous terms attempt to reuse a line accessed in the immediately previous iteration. The number of iteration of this kind (rem) is calculated by subtracting the iterations already considered in the other terms from the N_i total iterations.

$$\text{rem} = Ni - (\text{nrs} + \text{rso} + L_{Ri}(N_i) - \text{ngrtC}) \tag{10}$$

Example 6. *Calculation of* rem: Considering scheme A of Figure 4: $L_{Ri}(N_i) = 4$ which corresponds with the processing of elements $0, 4, 8$ and 12; nrs $= 12$ which corresponds to the processing of the remaining elements of the array, and ngrtC $= 4$ which corresponds to the processing of elements $2, 6, 10$ and 14, as it was stated in Example 5. Let us recall that term ngrtC is substracted from nrs. rso $= 0$ because $b_0 = 1$, so finally rem $= Ni - (\text{nrs} + \text{rso} + L_{Ri}(N_i) - \text{ngrtC}) = 16 - 12 = 4$ as $N_0 = 16$. This last term correspond to the processing of elements $2, 6, 10$, and 14 ■

This term takes into account the ngrtC iterations that attempt to reuse a SOL accessed in the previous parallel iteration, and the remaining iterations that, not being first-time accesses to SOL or inter-thread reuses, reuse a SOL accessed in the previous iteration by the same thread. This way the PME to model the behavior of a reference R that carries no reuses with other references in a parallel loop i is

$$\begin{aligned} Fp_{Ri}(\text{RD}) = & L_{Ri}(N_i) \times F_{R(i+1)}(RD) + \\ & (\text{nrs} - \text{ngrtC}) \times F_{R(i+1)}(\text{Iter}_i(\text{rd})) + \\ & \text{rso} \times F_{R(i+1)}(\text{Iter}_i(0)) + \\ & \text{rem} \times F_{R(i+1)}(\text{Iter}_i(1)) \end{aligned} \tag{11}$$

Example 7. *Calculation of the PME associated to a reference*: Let us consider again the code in Figure 3 and a shared direct-mapped cache which can store $C_s = 1024$ elements, with lines of $L_s = 8$ elements. Let us recall that the tile size is $ts = 32$ for both dimensions and that the iterations of the parallel loop in the code are distributed in blocks of $b_0 = 32$ iterations among 8 threads.

Example 1 was devoted to analyze the behavior of reference $R \equiv$`C[ci][cjj]`. In the three innermost loops containing the reference, which are sequential. We build here the PME that models of the behavior of that reference in the outermost loop at level 0 (L0). As that loop is parallelized, Equation (1) refers us to Equation (11), and $S_{R0} = 0$, since its loop index `ck` does not index any dimension of `C`.

L_{R0} is calculated using Equation (2), the loop has $N_0 = 8192/32 = 256$ iterations, so $L_{R0}(256) = 1$. nrs $= 7$ according to Equation (4), ngrtC $= 0$ (Equation (9)), rd $= 0$ (Equation (5)), rso $= 217$ (Equation (6)) and rem $= 31$ (Equation (10)). This way, $Fp_{R0}(\text{RD})$ is calculated as,

$$\begin{aligned} Fp_{R0}(\text{RD}) = \quad & 1 \times F_{R1}(RD) + 7 \times F_{R1}(\text{Iter}_0(0)) + \\ & 217 \times F_{R1}(\text{Iter}_0(0)) + 31 \times F_{R1}(\text{Iter}_0(1)) \end{aligned}$$

Let us examine the meaning of this expression. The same SOL is accessed in all the iterations of the loop. Thus, it is accessed once for the first time by one of the threads (reuse distance RD). In the same parallel iteration, the remaining 7 threads access the same line simultaneously (reuse distance 0). The same behavior is repeated in the remaining $256/8 - 1 = 31$ parallel iterations. One thread accesses the SOL for the first time (31 iterations now with a reuse distance of one iteration) and the other 7 threads reuse that SOL simultaneously ($31 \times 7 = 217$ iterations with a reuse distance of 0 iterations) ■

5. Miss probability calculation

Once the PMEs have identified the reuse distance of each access, the miss probability associated to those reuse distances must be calculated to compute the estimation of misses. This is achieved in three stages (access pattern identification, cache impact estimation, and area vectors union) which are discussed now in turn.

5.1. Access pattern identification

This step identifies the access pattern followed by each reference executed during the reuse distance. This access pattern is a formal characterization of the set of memory positions accessed by the reference. The access pattern followed by a reference can be used to derive the memory region defined by the accesses generated by that reference. The process to identify the access pattern that references follow during a reuse distance consisting of n iterations of the loop at nesting level h has two steps. They are discussed now in turn.

5.1.1. Indexing identification

In this step, for each reference R the indexes of each dimension and the number of iterations of each loop during this reuse distance are examined. The output of this analysis is a D_{A}-tuple $\mathcal{R}_R(h,n)$, where D_{A} is the number of dimensions of the array `A` referenced by R. Each element of these tuples is in its turn a 2-tuple $\mathcal{R}_{Rj}(h,n) = (M_j, S_j)$, where M_j is the number of points accessed along dimension j, and S_j is the distance between each two consecutive points. The algorithm followed to calculate the 2-tuple associated to dimension j of reference R during n iterations of the loop at nesting level h is as follows.

The indexing considered is always an affine function $\alpha_{Rj} \times \mathtt{I}_i + \delta_{Rj}$ of some loop index $\mathtt{I}_i$. The set of points accessed in this dimension by R can be represented as the tuple $(\mathrm{Iters}_i(h,n), S_{Ri})$, where S_{Ri} is the stride between the points accessed in two consecutive iterations of loop i, and $\mathrm{Iters}_i(h,n)$ stands for the number of different values that $\mathtt{I}_i$ takes during n iterations of the loop in nesting level h. This value is calculated as

$$\mathrm{Iters}_i(h,n) = \begin{cases} 1 & \text{if } i < h \\ n & \text{if } i = h \\ N_i & \text{if } i > h \end{cases}$$

The D_{A}-tuple $\mathcal{R}_R(h,n)$ may be simplified by applying some transformations between pairs of 2-tuples $\mathcal{R}_{Rj}(h,n)$ that describe the access pattern in different dimensions of the arrays such as:

$$\begin{array}{lcl} ((1, S_k), (M_j, S_j)) & = & (M_k, S_k) \\ ((M_k, M_j \times S_j), (M_j, S_j)) & = & (M_j \times M_k, S_j) \end{array} \tag{12}$$

Both simplifications refer to accesses associated to bidimensional matrices where the elements of the matrix are stored by rows. In the first simplification, the D_{A}-tuple represents the access to M_j elements of one row separated by a distance S_j. This access may be simplified as the access to M_j elements of the matrix separated by a distance S_j. In the second simplification the D_{A}-tuple represents the access to M_k rows of the matrix separated by a distance $M_j \times S_j$. In each row, M_j columns separated by a distance S_j are accessed. The distance between a pair of the rows accessed is equal to the distance between the first and the last element accessed of each row. This way, the access may be simplified as an access to $M_j \times M_k$ elements of the matrix separated by a distance S_j.

An additional vector $\overrightarrow{Q_R}$ is required to describe the region accessed by each reference R. If the reference is not indexed by a loop index of any parallel loop directly or indirectly, $\overrightarrow{Q_R}$ has a single component, which is the offset with respect to the beginning of the array of the first element accessed by R. If R is indexed by the loop index $\mathtt{I}_i$ of a parallel loop directly or indirectly (when a loop index that indexes R depends on $\mathtt{I}_i$) vector $\overrightarrow{Q_R}$ may contain several components. In this case, R generates concurrent accesses in each one of the t_i threads that participate in the execution of the loop. Each thread executes $\left\lceil \frac{N_i}{b_i \times t_i} \right\rceil$ groups of b_i consecutive iterations, where b_i is the block size. The distance between the elements that belongs to consecutive groups is $stride_i = b_i \times S_{Ri}$, where S_{Ri} is the stride that reference R has with respect to loop i. This way, if we are considering the reuse distance of n iterations of loop h, the components of the $\overrightarrow{Q_R}$ vector are calculated as follows,

- If $i > h$, $\overrightarrow{Q_R}$ has a single component, which is the offset with respect to the beginning of the array of the first element accessed by R. The reason is that the N_i iterations of the parallel loop are executed.
- If $i < h$, $\overrightarrow{Q_R}$ has t_i components because R has generated one access per thread. Component Q_{R_0} contains the offset with respect to the beginning of the array of the first element accessed by the first thread. The remaining components are calculated as $Q_{R_t} = Q_{R_0} + t \times \text{stride}_i$ where $t = 1 \ldots (t_i - 1)$.
- If $i = h$, the number of parallel iterations executed of the loop i is equal to the number of iterations of the reuse distance n. $\overrightarrow{Q_R}$ has $n \times t_i$ components. Component Q_{R_0} is calculated as in the previous case and the remaining values are calculated using the following expression

$$Q_{R_{(n*t+s)}} = Q_{R_0} + \lfloor s/b_i \rfloor \times b_i \times t_i + s \bmod b_i + t \times \text{stride}_i$$

  where $s = 0 \ldots (n - 1)$ and $t = 0 \ldots (t_i - 1)$.

5.1.2. Access pattern function identification

Rather than this description of the memory region accessed, the output of the access pattern identification step is a function that characterizes the access pattern. The most common kinds of access patterns identified are $\text{Reg}_\text{s}(S_\text{R})$, the access to S_R consecutive elements, and $\text{Reg}_\text{r}(M_\text{R}, L_\text{R}, T_\text{R})$, the access to M_R groups of L_R consecutive elements separated by a distance T_R. This way, when the result of the previous step is a single 2-tuple (M, S), depending on the values of M and S, two kinds of access pattern functions can be identified in this step. If $S = 1$, it is an access to M consecutive elements, $\text{Reg}_\text{s}(M)$. Otherwise it is an access to a set of M groups of one element separated by a constant stride S, $\text{Reg}_\text{r}(M, 1, S)$.

Sometimes more than one tuple appears in the region description. For example, when $\mathcal{R} = ((M_1, S_1), (M_2, 1))$, the access pattern can be represented by function $\text{Reg}_\text{r}(M_1, M_2, S_1)$. When $M_2 = S_1$, then the access pattern is $\text{Reg}_\text{s}(M_1 \times M_2)$.

As a result of the indexing identification step, there may be either several $\mathcal{R}_R(h, n)$ tuples associated to references to the same data structure, tuples with a $\overrightarrow{Q_R}$ parameter containing more than one component or both things simulatenously. These descriptors must be post-processed to: (1) identify overlapping or adjacent regions and (2) derive the overall access pattern that describes the memory positions associated to these references. This post-processing considers simultaneously all the components of all these $\overrightarrow{Q_R}$ parameters. The PME model provides simple

algorithms to merge the components of several $\overrightarrow{Q_R}$. This way, lines that are accessed by different references are not taken into account several times as source of interferences and several memory regions associated to the same data structure may be merged to be characterized as a unique one. For example, in the parallel case due to the regular distribution of the parallelized loop iterations among the threads and the affine indexing considered in this paper, each two components of the $\overrightarrow{Q_R}$ vector may differ a fixed value $stride_i$. This way, for example, if the access has been identified as sequential, $\text{Reg}_\text{s}(M)$, then, considering the combined effect of all the threads, the access is represented by $\text{Reg}_\text{r}(N, M, stride_i)$, i.e. the access to N groups of M elements separated by a distance $stride_i$.

Example 8. *Calculation of the interference region associated to a reference during a reuse distance*: Examples 1 and 7 built the PMEs for reference `C[ci][cjj]` in the code of Figure 3. Let us consider that the number of threads used in the parallelization is $t_0 = 8$ and the block size $b_0 = 32$. We explain here part of the access pattern identification step of the reference `A[ci][ckk]` during the reuse distance $\text{Iter}_3(1)$, i.e., one iteration of loop 3 (L3.2 for this reference).

The first tuple of the 2-tuple $\mathcal{R}_R(3,1)$ that characterizes the region it accesses in the reuse distance is $(\text{Iters}_2(3,1), S_{R2})$, because the first dimension of this reference is indexed by the loop index of loop 2. $\text{Iters}_2(3,1) = 1$, as $(i = 2) < (h = 3)$, and $S_{R0} = \alpha_{R0} \times d_{\text{A}0} = 1 \times 8192 = 8192$. Thus, the first component of $\mathcal{R}_R(3,1)$ is the tuple $(1, 8192)$. The second component of $\mathcal{R}_R(3,1)$ characterizes the access to the second dimension in the reference and it is the tuple $(\text{Iters}_4(3,1), S_{R4})$, as this dimension is accessed by the loop at level 4 (L4.2). The value $\text{Iters}_4(3,1) = N_4 = 256$ as $(i = 4) > (h = 3)$, while $S_{R4} = \alpha_{R1} \times d_{\text{A}1} = 1 \times 1 = 1$. This way, the second component of $\mathcal{R}_R(1,1)$ is the tuple $(256, 1)$.

The parallel loop (loop $i = 0$) indexes indirectly the second dimension of reference `A[ci][ckk]` because this loop selects the different tiles in which this dimension is divided. Its $\overrightarrow{Q_R}$ parameter has $t_0 = 8$ components as $(i = 0) < (h = 3)$. Component Q_{R_0} contains the offset with respect to the beginning of the array of the first element accessed and the remaining 7 components are generated using the expression $Q_{R_t} = Q_{R_0} + t \times \text{stride}_0$ where $t = 1 \ldots ((t_0 = 8) - 1)$ and $\text{stride}_0 = (b_0 = 32) \times (S_{R0} = 256) = 8192$. $S_{R0} = \alpha_{R1} \times d_{\text{A}1} = 256 \times 1 = 256$ as loop 0 has stride 256 and must be normalized to have stride one. As a result, the values contained in the 8 components of $\overrightarrow{Q_R}$ are separated by a constant stride 8192.

The two components of the $\mathcal{R}_R(3,1)$ tuple can be simplified as $((1, 8192), (256, 1)) = (256, 1)$ according to Equation (12). As a result, the access can be represented by a single tuple $(M = 256, S = 1)$. As $S = 1$, it is an access to $M = 256$ consecutive elements, $\text{Reg}_\text{s}(256)$. Then, considering the combined effect of all the threads, and as the 8 components of $\overrightarrow{Q_R}$ are separated by a constant $\text{stride}_0 = 8192$, the access is represented by $\text{Reg}_\text{r}(8, 256, 8192)$ ■

5.2. Cache impact estimation

The cache impact estimation step quantifies the impact on the cache of each access pattern found using a mathematical representation of the distribution of the lines of the access pattern on the cache sets, called area vector. An area vector V has $k + 1$ components, k being the associativity of the cache. Component V_0, represents the ratio of cache sets that receive k or more lines from the access, while component V_i, $0 < i \leq k$, stands for the ratio of cache sets that receive $k - i$ lines. There is a method to calculate the area vector associated to each kind of access pattern. The methods to calculate the area vector associated to the most common access

patterns have already been described in [2]. For example, the sequential access to n consecutive words $\mathrm{Reg_s}(S_\mathrm{R})$ generates an area vector AV_s:

$$\begin{array}{l} AV_{\mathrm{s}_{(k-\lfloor l \rfloor)}}(S_\mathrm{R}) = 1 - (l - \lfloor l \rfloor) \\ AV_{\mathrm{s}_{(k-\lfloor l \rfloor - 1)}}(S_\mathrm{R}) = l - \lfloor l \rfloor \\ AV_{\mathrm{s}_i}(S_\mathrm{R}) = 0 \qquad 0 \leq i < k - \lfloor l \rfloor - 1, k - \lfloor l \rfloor < i \leq k \end{array} \tag{13}$$

where $l = \min\{k, (S_\mathrm{R} + L_\mathrm{s} - 1)/(L_\mathrm{s} N_K)\}$ is the minimum of k and the average number of lines placed in each set. In this expression, L_s stands for the line size and N_K for the number of cache sets. The number of cache sets N_K can be calculated as $C_\mathrm{s}/L_\mathrm{s}k$. The term $L_\mathrm{s} - 1$ added to n stands for the average extra words brought to the cache in the first and last accessed lines.

Example 9. The impact of a sequential access to 256 elements, $\mathrm{Reg_s}(256)$, on a direct mapped cache that can store 1024 elements distributed in lines of 8 elements, is represented by the area vector with the values $AV_\mathrm{s} = (0.256836, 0.743164)$ because $l = \min\{k, (S_\mathrm{R} + L_\mathrm{s} - 1)/(L_\mathrm{s} N_K)\} = \min\{1, (256+8-1)/(8\times128)\} = 0.256836$ as $Nk = C_\mathrm{s}/L_\mathrm{s}k = 1024/(8\times1) = 128$. Once the value of l is replaced in Equation (13), $AV_{\mathrm{s}_{(k-\lfloor l \rfloor)}}(S_\mathrm{R}) = AV_{\mathrm{s}_{(1-0)}}(256) = 1-(l-\lfloor l \rfloor) = 1-(0.256836-0) = 0.743164$ and $AV_{\mathrm{s}_{(k-\lfloor l \rfloor-1)}}(S_\mathrm{R}) = AV_{\mathrm{s}_{(1-0-1)}}(256) = l - \lfloor l \rfloor = 0.256836 - 0 = 0.256836$. This means that 25.6836% of the cache sets receives 1 or more lines from the access pattern, while the remaining 74.3164% receives none. ■

5.3. Area vectors union

The preceding step generates an area vector per access pattern found during a reuse distance. Each component of one of these area vectors is the ratio of cache sets that receives a given number of lines from those that comprise the memory region accessed by the corresponding access pattern, which is conversely the probability a cache set holds a given number of lines from those lines. In this last step these area vectors are added in order to get a global interference area vector which represents the total impact on the cache of all the accesses that take place during the considered reuse distance. The component 0 of this area vector is the miss probability we are trying to estimate. The reason is that this component is the ratio of sets that have received a number of different lines equal or greater than the associativity (k) during the reuse distance, and in a cache with LRU replacement policy a reuse attempt results in a miss if and only if this number of lines is mapped to the cache set considered during the reuse distance. The algorithm to add area vectors has already been described in [2]. Given two area vectors V_A and V_B, their addition, represented by the operator $\cup$, is calculated as

$$\begin{array}{l} (V_\mathrm{A} \cup V_\mathrm{B})_0 = \sum_{j=0}^{K} \left(V_{\mathrm{A}_j} \sum_{i=0}^{K-j} V_{\mathrm{B}_i} \right) \\ (V_\mathrm{A} \cup V_\mathrm{B})_i = \sum_{j=i}^{K} V_{\mathrm{A}_j} V_{\mathrm{B}_{(K+i-j)}} \qquad 0 < i \leq K \end{array}$$

This method is based on the addition as independent probabilities of the area ratios, which means that it does not take into account the relative positions of the program data structures in memory. This approach allows our model to provide reasonable estimations in many situations in which the base addresses of the data structures are not known at compile time (e.g. physically-addressed caches, which is in fact the case of all the actual shared caches we know of), something that, as far as we know, no other model supports.

Example 10. Using the method previously described, the union of two area vectors $V_\mathrm{A} = (0.25, 0.75)$ and $V_\mathrm{B} = (0.15, 0.85)$ is $(V_\mathrm{A} \cup V_\mathrm{B}) = (0.3625, 0.6375)$. The rationale is that if

Processor	Level	Cache size	Line size	Associativity
Intel Core i7	1 (private)	32KB	64B	8
	2 (private)	256KB	64B	8
	3 (shared)	8MB	64B	16
Intel Core 2 Duo	1 (private)	32KB	64B	8
	2 (shared)	4MB	64B	16

Table 2: Characteristics of the cache hierarchy of two different multicore systems

each region places one ore more cache lines in a 25% and in 15% of the cache sets, respectively, and the location of these cache sets is unknown, the percentage of cache sets that receives one or more cache lines from these two regions is $25 \times 15 + 25 \times 85 + 75 \times 15 = 36.25\%$. That is, a cache set receives at least one cache line from both regions, A and B, if it receives at least one cache line from A and B, or at least one cache line from A but none from B, or at least one cache line from B and none from A. The remaining 63.75% cache sets do not receive a cache line from any of both regions ■

6. Experimental results

The accuracy of our cache prediction has been first validated against the SESC cycle-accurate simulator [11]. In a second set of experiments, it is verified that the model accurately reflects the changes in the behavior of the cache due to the existence of a different number of threads and block sizes in the parallelization of loops. In this second set of experiments, the predictions of the model are validated against the SESC simulator and against the actual execution time of the code in real platforms. Let us notice that neither the simulations nor the actual executions enforce a lock-step execution of the iterations by the threads involved in the parallelization.

The experiments consider two cache hierarchies with the parameters of two well-known multicore processors, Intel Core 2 Duo and Intel Core i7 [12]. The main characteristics of these memory hierarchies are outlined in Table 2. For each processor and each cache level the table specifies whether the cache is private to each core or shared, the cache size, the line size, and the associativity. The characteristics of the L1 instruction cache are not included, as its effects on the shared cache levels during the execution of the codes used in this section are negligible. We were able to confirm this fact using the SESC simulator. Unless otherwise stated, the number of threads used in the experiments for each configuration is equal to the number of cores that exist in the processor (2 for the Intel Core 2, and 4 for the Intel Core i7).

The accuracy of the model has been validated using thirteen codes (see first column of Table 3). MATMULIJK performs the parallel product of two matrices in IJK order. These three letters indicate the order of the loop indexes in the code, the first index is the one for the outer loop and the last index is the one for the innermost loop in the nest. I and K select the row and the column of the first matrix to be multiplied, respectively, and J selects the column of the second matrix to be multiplied. MATMULJIK performs also a parallel matrix product but using order JIK; MATMULTB, shown in Figure 3, performs the same operation as MATMULIJK but applying tiling (to loops J and K), and buffering for the tile of matrix `b`; TRANS performs the parallel transposition of a matrix. SWIM is a parallelized version of the complete SWIM benchmark from the SPECfp95 suite already used in [2]. SU2COR-MATMAT and HYDRO2D-FILTER are parallelized versions of the most-time consuming routines in the SU2COR and HYDRO2D

	# Loops	# omp for pragmas	# parallelized loops	# References	Ex. time %
MATMULIJK	3	1	3	5	100%
MATMULJIK	3	1	3	4	100%
MATMULTB	7	1	7	6	100%
TRANS	2	1	2	2	100%
SWIM	23	7	23	193	100%
SU2COR-MATMAT	1	1	1	36	23.5%
HYDRO2D-FILTER	24	14	24	120	46.8%
SP-LHSY	14	5	14	77	12.8%
SP-TZETAR	4	1	4	21	6.8%
SP-COMPUTE-RHS	68	21	53	289	36.4%
LU-JACLD	2	1	2	284	31%
MG-COMM3	6	3	6	12	1%
MG-RESID	4	1	4	21	47%

Table 3: Characteristics of the codes analyzed

benchmarks of the same suite. Finally, the analysis of SP-LHSY, SP-TZETAR, SP-COMPUTE-RHS, LU-JACLD, MG-COMM3 and MG-RESID routines covers a significative percentage of the execution time of the SP, LU and MG benchmarks of a version of the NAS Parallel Benchmarks parallelized using OpenMP (NPB-omp).

In MATMULIJK, MATMULJIK, MATMULTB and TRANS, the outermost loop has been parallelized using the OpenMP `for` pragma. A similar strategy was used to parallelize all the parallelizable loops of the codes from the SPECfp95 suite (SWIM, SU2COR-MATMAT and HYDRO2D-FILTER). Finally, the codes from the NPB-omp suite are already parallelized and thus, they have been analyzed as provided in this suite. The complexity of these codes is summarized in Table 3. For each code analyzed, the table contains the number of loops (# Loops), the number of OpenMP `for` pragmas used in the parallelization (# omp for pragmas), the number of loops whose execution is nested inside one of the pragmas (# parallelized loops), the number of array references (# References), and the percentage of the execution time of the whole benchmark consumed by the analyzed routine (Ex. time %). All the references in all the codes are inside parallel regions, so they are executed simultaneously by different threads.

Table 4 compares the number of misses predicted by the model with the number of misses observed in the simulator in the shared cache of the two processors considered, using both static and dynamic policies for the scheduling of the parallel execution. Regarding the problem size, for MATMULIJK, MATMULJIK, MATMULTB and TRANS, each dimension of each one of the data structures involved has size 1000. In the case of MATMULTB, the tile size in both dimensions has been fixed to 100. In these codes the iterations of the parallel loop are distributed by blocks of 4 iterations. In the benchmarks from SPECfp95 suite the default problem size was used and the iterations of the parallelized loops were distributed among the threads in blocks of 25 iterations. Finally, for the benchmarks from the NPB-omp suite, the class C size and a block size of 16 iterations were used for the parallelized loops.

The predictions of the number of misses in the private cache levels were also computed using the PME model in [2], and they accurately reflected the values obtained by the simulator. They are not shown in the table as their computation does not require the extension of the model presented in this paper. For each case, the table also includes the miss rate (MR) with respect to the total number of accesses that produces the code during its execution. The predictions are precise even although the accesses do not take place directly on the shared cache, as they are

Code	Intel Core i7 (4 threads)					
	mod		sim (static)		sim (dynamic)	
	misses	MR	misses	MR	misses	MR
MATMULIJK	1971369	0.09%	2160860	0.10%	1931723	0.09%
MATMULJIK	7220821	0.36%	7007347	0.35%	7133528	0.35%
MATMULTB	677659	0.01%	774540	0.01%	774591	0.01%
TRANS	250000	12.5%	250000	12.5%	249461	12.47%
SWIM	354249154	1.51%	257314373	1.1%	257309505	1.1%
SU2COR-MATMAT	6156	4.17%	6153	4.17%	6153	4.17%
HYDRO2D-FILTER	65675	1.45%	64775	1.43%	64831	1.43%
SP-LHSY	14956562	7.61%	14516653	7.38%	14517172	7.38%
SP-TZETAR	6412800	9.42%	6221760	9.14%	6221760	9.14%
SP-COMPUTE-RHS	89480848	7.39%	96357543	7.95%	96343323	7.95%
LU-JACLD	559649	7.46%	388011	5.15%	386541	5.15%
MG-COMM3	853762	27.14%	784796	24.94%	784796	24.94%
MG-RESID	84801106	3.42%	83661839	3.37%	83661846	3.37%
	Intel Core 2 Duo (2 threads)					
	mod		sim (static)		sim (dynamic)	
	misses	MR	misses	MR	misses	MR
MATMULIJK	62750000	3.13%	62750000	3.13%	62750000	3.13%
MATMULJIK	63500000	3.17%	63376500	3.16%	63625500	3.17%
MATMULTB	1191320	0.02%	1003850	0.02%	1006375	0.02%
TRANS	250000	12.5%	250000	12.5%	250000	12.5%
SWIM	547768496	2.34%	516181431	2.21%	516145330	2.21%
SU2COR-MATMAT	6156	4.17%	6153	4.17%	6153	4.17%
HYDRO2D-FILTER	65675	1.45%	64839	1.43%	64863	1.43%
SP-LHSY	14956562	7.61%	14516868	7.38%	14516874	7.38%
SP-TZETAR	6412800	9.42%	6221760	9.14%	6221760	9.14%
SP-COMPUTE-RHS	98836762	7.54%	98824503	8.16%	6607318	8.16%
LU-JACLD	561276	7.48%	386840	5.15%	386840	5.15%
MG-COMM3	853762	27.14%	784788	24.94%	784788	24.94%
MG-RESID	84800831	3.42%	83665677	3.38%	83660279	3.37%

Table 4: Number of misses and miss rate (MR) of the shared level of the cache predicted by the model (mod) and observed in the simulator (sim) respectively using both static and dynamic scheduling. Each dimension of each one of the data structures involved has size 1000. In MATMULTB, a tile size of 100 has been used in both dimensions.

filtered previously by one or two upper levels of private caches. In previous works [2, 6, 10], it has been shown that if the n-th cache level is modeled directly using the PME model and ignoring the filtering of the upper cache levels, the predictions are still accurate.

The estimations of the model are scheduling policy-independent, and the simulation results confirm that the cache behavior using both policies does not change substantially. The reason is that in the static scheduling, the blocks of b_i iterations are assigned cyclically to the threads, while in the dynamic scheduling the order of assignment depends of the occupancy of the threads and all the threads have a very similar load. The load is similar because all the iterations of the loops require the same amount of computation. The cache behavior does not depend on the order used to assign the blocks of iterations to the threads. The prediction of the model matches the cache behavior observed in the simulator. Another interesting characteristic of our model is that the predictions are obtained very fast. The average modeling time is 0.56 seconds, the minimum modeling time is 0.01 seconds and the maximum one is 2.3 seconds. The modeling times were obtained using an Intel Core 2 Duo at 2.4 Ghz with 2GB of RAM.

Figure 6 illustrates how application behavior can largely change depending on the cache

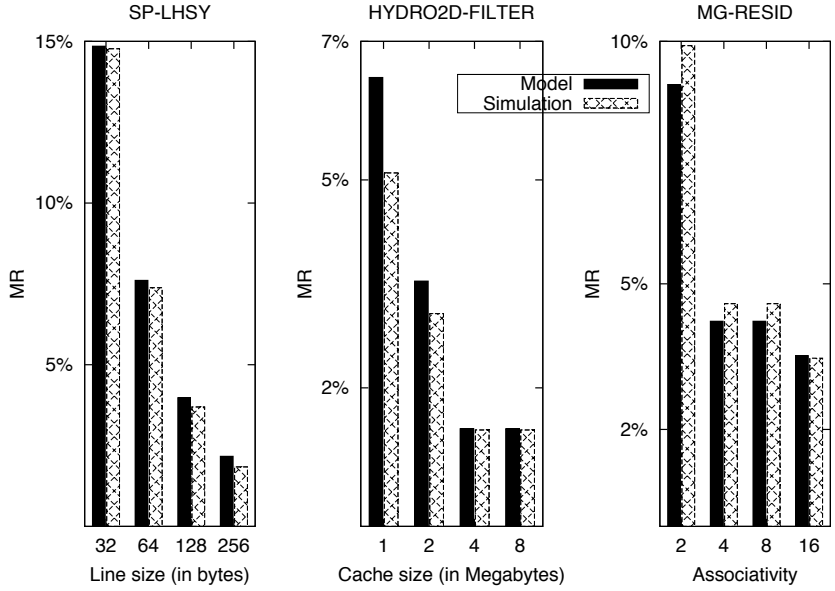


Figure 6: Miss rate prediction vs. miss rate observed in the simulation changing the line size, cache size and associativity of the Intel i7 cache

configuration and how our model reliably predicts these changes. The figure shows the miss rate predicted and the miss rate obtained in the simulator changing the line size, cache size and associativity of the Intel i7 shared cache level, for the SP-LHSY, HYDRO2D-FILTER and MG-RESID codes parallelized using 4 threads, respectively. In the SP-LHSY and MG-RESID codes, the class C size has been used. We can see that changes in any of the three basic configuration parameters of the shared cache can give place to strong variations of the performance, and that our model accurately predicts them.

Figure 7 shows a comparison of the miss rate predicted and the one observed in the simulation using three different parallelization schemes (changing the number of threads t_i and block size b_i) for the MATMULJIK code and the Intel i7 cache for different problem sizes. The predictions of the model are very accurate, as they match the behavior observed in the simulator. When the problem size is small with respect to the cache (500x500 and 750x750) the cache performance is not influenced by the parallelization scheme. However, for larger problem sizes the cache performance changes depending on the parallelization scheme and the model perfectly captures this behavior. Thus, it is important to study the influence of the number of threads and the block size on the performance of shared caches. This way, the second target of our experiments is to validate that the model precisely reflects the changes in the behavior of the code due to the existence of a different number of threads and block sizes in the parallelization of loops. The results of these experiments are shown in Sections 6.1 and 6.2 respectively. In these sections, the predictions of the model are validated against the simulator. Additionally, it is also verified that these changes in the cache behavior affect the actual execution time of the code. Finally, Section 6.3 shows how the predictions of the model can be used to guide compiler optimizations.

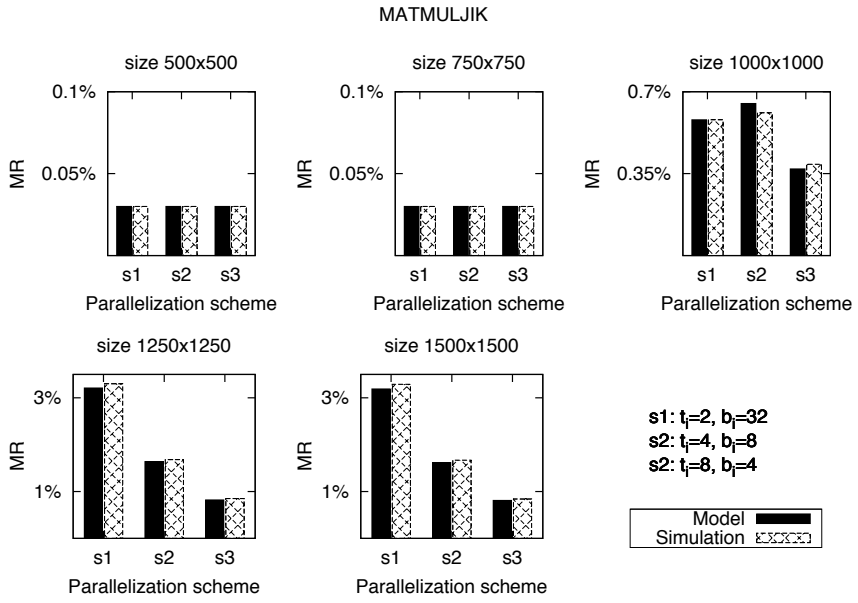
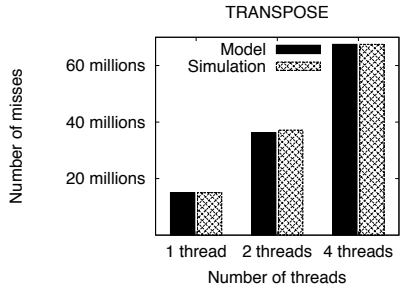


Figure 7: Miss rate prediction vs. miss rate observed in the simulation using different parallelization schemes and problem sizes for the MATMULJIK code on the Intel i7 cache

6.1. Influence of the number of threads in the shared cache performance

The first factor whose effect in the shared cache performance is studied, is the number of threads used in the parallelization. Thus, Figure 8(a) shows the number of misses predicted by the model and the number of misses observed in the simulator in the shared cache present in an Intel Core i7 when transposing with TRANS a 60000×1000 matrix using 1, 2 or 4 threads. Each one of the t threads is assigned a block of $(N = 60000)/t$ consecutive iterations, where $N = 60000$ is the number of iterations of the parallel loop. The reason why in this case a square matrix has not been used is that it has been detected analytically that the cache performance of the transposition of a cache with 60000 elements per row could be largely influenced by the variation of the number of threads in this kind of cache. However, each 60000×60000 matrix of elements of type double occupies around 26 Gigabytes, so the performance of this code in these conditions would have been negatively influenced by the impossibility to fit the data structures in the main memory of our system. The problem sizes used in the other experiments have also been selected on purpose to be illustrative of the influence of different factors in the shared cache performance. Table 8(b) shows the number of misses predicted by the model for each one of the data structures involved. The ability to provide the number of misses per reference (and even per nesting level) is an interesting characteristic of this model, as this information may be used in optimization processes. The results show that the usage of several threads in the code influences the number of misses of matrix `b` while the number of misses of matrix `a` remains unaltered. The last column of the table shows the actual execution time of the code on a system with a Intel Core i7. The large increase of the number of misses predicted is reflected in the execution times, even although the work is distributed among more threads. Let us analyze the possible causes of these

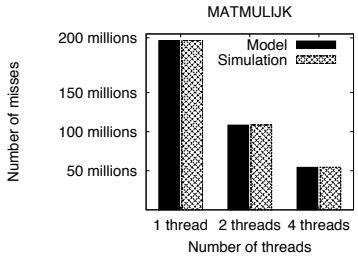


(a) Number of misses

threads	Execution time		Time
	Matrix a	Matrix b	(ms)
1	7500000	7500000	689
2	7500000	29679788	940
4	7500000	37179788	1839

(b) Number of misses predicted by the model and execution time

Figure 8: Cache behavior of a shared cache of an Intel Core i7 when executing TRANS on a 60000 × 1000 matrix using different number of threads



(a) Number of Misses

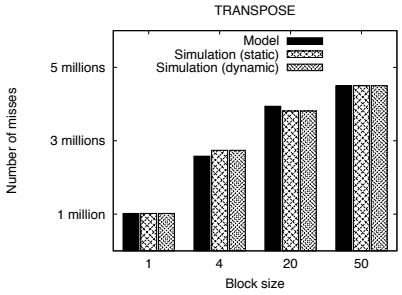
Threads	Misses			Execution time
	Matrix a	Matrix b	Matrix c	(ms)
1	180	216000	180000	14618
2	180	108000	180000	7375
4	180	54000	180000	3919

(b) Number of misses (in thousands) predicted by the model and execution time

Figure 9: Cache behavior of the shared cache of an Intel Core i7 when executing MATMULIJK on two 1200 × 1200 matrices using different number of threads

results. This code consists of two nested loops where one matrix `a`, the matrix to be transposed, is accessed by rows and another matrix `b`, the matrix where the result is stored, is accessed by columns. The iterations of the outermost loop are distributed among t_i threads. The reuse in matrix `a` is between consecutive iterations of the innermost loop. During this reuse distance, t_i elements of `a` and `b` are accessed so the miss probability associated to this reuse distance is very small even when using 4 threads. The reuse in matrix `b` takes place between iterations of the outermost loop. During this reuse distance, t_i rows of `a` and t_i columns of `b` are accessed. The miss probability associated to this reuse distance is high, and the usage of several threads increases the number of misses.

Figure 9(a) shows the results of a similar experiment for the multiplication of two 1200 × 1200 matrices on the shared cache of an Intel Core i7 using the MATMULIJK code. The distribution of the iterations among the threads is consecutive, therefore, each one of the t threads is assigned a block of $(N = 1200)/t$ consecutive iterations, where $N = 1200$ is the number of iterations of the parallel loop. In this case, as the number of threads increases we observe a reduction of the number of misses. Table 9(b) shows the number of misses per structure predicted by the model and the execution time of the real code. The cache performance of the accesses to



(a) Number of misses

Block	Misses		Execution time (ms)
	Matrix a	Matrix b	
1	500000	520000	152
4	500000	2080000	186
20	500000	3440000	210
50	500000	4000000	215

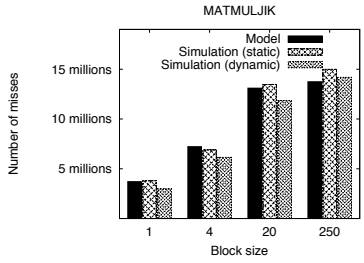
(b) Number of misses predicted by the model and execution time

Figure 10: Cache behavior of the shared cache of an Intel Core 2 Duo when executing TRANS on a 40000×100 matrix with 2 threads using different block sizes in the parallelization

matrices `a` and `c` remains unaltered, while the number of misses produced by the accesses to matrix `b` decreases with the number of threads used. This improvement in the memory performance reduces the execution time in the target architecture, as the access to matrix `b` generates most of the cache misses in this code. The reduction of the execution time in this case is due both to: the better memory performance and the distribution of the work among an increasing number of threads. According to the results in Table 4 (where 1000×1000 matrices were considered), the miss rate in this code is below 1%, so it can be inferred that the usage of an increasing number of threads that do not interfere among them is the main reason for the improvement of the execution time. Let us analyze the causes of this behavior. This code performs the matrix product $a = b \times c$ in sucha a way that matrices `a` and `c` are accessed by rows, while `b` is accessed by columns. Thus, the reference that accesses `b` is likely the main source of cache misses in this code. The reason is that in different iterations of the outermost loop, the same positions of `b` are accessed in the same order, and as this loop is parallel, several iterations of this outermost loop are executed by different threads concurrently. If the progress of all the threads along the iterations of the different loops is similar, the access to each element of `b` in one thread takes advantage of the access to the same element that takes place simultaneously in another thread.

6.2. Influence of the block size in the shared cache performance

The second factor which can influence the cache performance of a shared cache is the block size, that is, the number of consecutive iterations that are assigned to each thread. Thus, Figure 10(a) represents the number of misses predicted by the model and the number of misses observed in the simulator using both a static and a dynamic strategy to distribute the iterations among the threads, when transposing with TRANS a 40000×100 matrix in the shared cache of an Intel Core 2 Duo. Block sizes of 1, 4, 20 and 50 iterations have been used to distribute the iterations of the parallel loop among 2 threads. The predictions of the model are good and match the behavior observed in the simulator when both static and dynamic strategies are used to distribute the iterations among the threads. The cyclic distribution, where the block size is equal to one iteration, is the one that produces the smallest number of misses. Table 10(b) shows



(a) Number of misses

Block	Misses			Execution time (ms)
	Matrix a	Matrix b	Matrix c	
1	3362196	172910	172910	11947
4	6607024	306898	306898	12306
20	11737707	680558	680558	12368
250	11957005	894765	894765	12559

(b) Number of misses predicted by the model and execution time

Figure 11: Cache behavior of the shared cache of an Intel Core i7 when executing MATMULJIK on two 1000×1000 matrices using different block sizes in the parallelization

the number of misses per matrix predicted by the model and the execution time of the real code in the target architecture. The number of misses of `b` and the execution time of the real code increase with the block size. The reason is that in this code, the destination matrix `a` is accessed by rows, while the source one `b` is accessed by columns. A cyclic distribution of the iterations facilitates the reuse of lines between the accesses to different columns of matrix `b` that take place in different threads.

Finally, Figure 11(a) shows the number of misses predicted by the model and observed in the simulator using both a static and a dynamic strategy to distribute the iterations among the threads, when executing MATMULJIK on two 1000×1000 matrices using different block sizes for the parallelization of the outermost loop in the shared cache of an Intel Core i7. Block sizes of 1, 4, 20 and 250 iterations have been used to distribute the iterations of the parallel loop among 4 threads. The cyclic distribution is the one that generates the fewest number of misses. The code performs the matrix product $a = b \times c$. Table 11(b) shows that the number of misses produced by the access to the three matrices involved in the code increase with the block size. As a consequence, the execution time also increases for larger block sizes. The reason is that many of the reuses (or potential misses) may take place among consecutive iterations of the outermost loop of this code, where consecutive elements of the same row of `b` and `c` are accessed, and the access to the whole matrix `a` is repeated. This way, in one iteration of this loop, the whole matrix `a` and four columns (one per thread) of `b` and `c` are accessed. The number of lines brought to the cache by the access to these four columns increases with the block size b_i, as they may share the same line or instead they may be located in different lines. In the studied case, for the smallest block sizes, the lines brought to the cache in one iteration of the outermost loop fit in the cache. This is not the case for the larger ones, which avoid reuses and consequently increase largely the number of misses.

6.3. Usage of the model to guide compiler decisions

The previous results show that the model is very accurate, as it matches the behavior observed in a simulator. Besides, it provides its predictions in less than three seconds independently of the analyzed code. This turns our model in a very valuable tool to guide compiler optimizations. For example, the first two rows of Table 4 show that the MATMULIJK ordering of a matrix product has more locality than the JIK ordering. However, the difference among both implementations is less important in the cache of an Intel Core 2 Duo. Anyway, according to the results in this table

MATMULTB should be chosen as the best implementation of a matrix product from the cache point of view as it produces fewer cache misses and a smaller miss rate. Thus, our model can be used to statically asses the locality of different variants of the same kernel.

Figures 8(a) and 9(a) show that the model correctly tracks the evolution of the number of misses with respect to the number of threads used for the parallelization in different scenarios. This way, Fig. 8(a) shows that the CPU performance gain expected due to the usage of more threads can be spoiled by the existence of a high number of collisions in the shared caches. The opposite happens in Fig. 9(a) where the benefit of using more threads is increased due to the intense reuse of cache lines in the shared cache among different threads. This information can be used by the compiler to select the optimal number of threads used to paralellize a loop.

Finally, Figures 10(a) and 11(a) show that the model also correctly tracks the effect of the block size in the number of misses. In both cases, the number of misses increases with the block size. This information would be useful at compile-time to select the optimal block size.

7. Related work

Cache modeling has been extensively studied in sequential single core environments. For example, Beyls and D'Hollander [13] use a set of reuse distance metrics to tune the cache hints values available in the memory operations of the EPIC architectures. Unlike our work, they can only predict the number of capacity misses of non-parallel codes and they do not take into account shared caches.

Cascaval and Padua in [4] use a stack histogram model to predict the cache performance. These predictions are used to evaluate memory optimizations like tiling, data shackling and the product-space transformation, or to estimate the optimal tile size. As in the preceding paper, they only use sequential codes and they do not consider the existence of shared caches. The accuracy of their approach is only relatively good for fully associative caches, while our approach accurately predicts the behavior of cache size with arbitrary associativities. They also explicitly state that the accuracy of their approach can be significatively lower for larger data sets, while our experimental results show high accuracy independently of the problem size.

Other works focus on the cache behavior on multicore and many-core environments, but only for single-threaded applications. This is the case of Chauhan and Shei [14], who present an efficient algorithm to compute static reuse distance at source level (without requiring a trace of the memory addresses) using an extended version of dependence graphs. They accurately analyze the behavior of complete MATLAB programs, which rely on third-party libraries to perform certain computations in multicore or many-core environments, but both their model and their validation only consider applications with only one active thread. As a result, as in the preceding papers, they do not consider the impact on the cache of multithreaded codes. Besides, their model ignores cache associativity, which limits the accuracy and applicability of the approach. Finally, unlike our model, which provides its estimations in a few seconds, this approach is very time-consuming for nested loops, which may be fine when you target codes in MATLAB but not for applications written in general-purpose languages.

Many works in the bibliography study the impact of cache sharing on co-scheduled threads, assuming that these threads execute different applications that do not share any address space. Kim et al. [15] study the fairness in cache sharing between threads and propose different metrics to measure the degree of fairness in cache sharing, some of which strongly correlate with execution time fairness. Using these metrics they propose static and dynamic methodologies to

partition the cache maximizing fairness. These techniques are tested on a set of co-scheduled benchmarks and an improvement of $4x$ is observed in the fairness. The paper concludes that the improvement of the fair cache sharing also improves throughput. The problem considered by this work is different from that considered by us as it aims to reduce fairness in the execution of threads associated to different applications on a multicore systems.

West et al. [16] estimate analytically the cache occupancy of set-associated caches with a LRU replacement policy by individual threads using the hardware counters. They do not consider the effect of data sharing among different threads, which has a significant impact on the cache performance as we have proved in this paper. On top of this analytical model they build a method to construct online miss rate curves which are used to make informed resource management decisions.

Chandra et al. [17, 18] propose three performance models that predict the impact of L2 cache sharing on co-scheduled threads that do not share any address space. They use as an input the L2 cache distance or circular sequence profile of each thread, and predict the number of extra L2 cache misses for each thread due to cache sharing. The input information is obtained through profiling. The models have been validated against a simulation of fourteen pairs of benchmarks (mostly from SPEC). The most accurate model achieves an average error of 3.9%. This model also considers the execution of threads associated to different applications on a multicore environment which is a different target than the one considered by ours.

Xu et al. [19] present CAMP, a model that calculates a metric of the cache contention among two different applications in shared caches using as an input reuse distance histograms, cache access frequencies and the relationship between the throughput and the cache miss rate of each process. The model has been validated considering the execution of pairs benchmarks from SPEC CPU2000 on a dual core machine, so the target of this model is also different from ours.

Other works consider thread-parallel numeric codes like those considered in our work. This way, Song et al. [20] propose an analytical model for predicting the number of misses of shared caches assuming that they are fully associative, which can strongly restrict its accuracy in real memory hierarchies. Besides, the model requires the trace of accesses to the shared cache. The model is validated against simulations on fully-associative caches of three typical scientific codes obtaining an average relative error between 2% and 12%. However, our model is general enough to consider any associativity and it only uses as an input the source code and the cache configuration. The usage of traces is very time consuming and is a potential source of inaccuracy as the trace reveals the characteristics of the execution of one code under certain circumstances.

Schuff et al [21] model the effect of multithreaded applications on private and shared caches by using a variant of the well-known reuse distance measurement method. Their reuse distance approach requires the instrumentation of the code to obtain the trace of addresses accessed by the program. The authors use sampling to reduce the overhead introduced by this instrumentation. A parallelized analysis completes a technique. This analysis is 80 times to 265 times slower than native execution. Let us recall that our technique allows to obtain estimations in less than one second no matter which is the execution time of the real code. The experimental results show the accuracy of this sampled technique with respect to that one which uses the full trace of the addresses. The technique demonstrate effective for an example application of selecting important portions of code which cause cache misses.

Kandemir et al. [22] propose a compiler based code restructuring scheme that assigns chunks of iterations to each core and decides in which order to run those chunks seeking to optimize the locality in the shared caches. This way, they improve the locality of the data accessed on each core. The application of this technique to Splash-2 and Parsec benchmarks reduced their

execution time by 29% on average. Unlike our model, this technique does not predict the cache behavior, thus it cannot guide general compiler transformations such as loop reordering, tile size selection, etc.

Kandemir et al. [23] develop an automated source-to-source transformation tool that uses a cache-aware tile scheduling strategy customized for on-chip cache topologies of multicore machines. The method takes into account horizontal and vertical data reuses in on-chip caches while exploiting intra-tile and inter-tile reuses. This technique produces a reduction of a 27.9% of the cache misses a 13.5% of the execution time over PLUTO [24]. This work is focused on the application of a specific cache optimization technique, while the purpose of our work is to provide a tool to guide the application of arbitrary optimizations to improve the cache behavior.

Jian et al. [25] define the concurrent reuse distance, an extension of the traditional reuse distance that aims to characterize the behavior of caches shared by concurrent threads. Their proposal, which can only predict the cold and the capacity misses, requires the memory traces of the threads. As a result, either profiling or simulation must be performed before the behavior can be predicted. The error of the reuse histograms predicted is around 25% for the real applications tested.

Other works are focused on the improvement of the performance of shared caches that can be obtained through cache partitioning. This way, Lu et al. [26] develop Soft-OLP (Software-based Object-Level cache Partitioning), a tool that collects per-object reuse distance and inter-object interference histograms through memory sampling to determine the locality pattern of data objects. Data objects are classified into several locality types and the cache is partitioned among those objects using heuristics to reduce cache misses. This strategy obtains up to a 78% L2 cache misses reduction with respect to a standard LRU L2 cache, for a set of single and multithreaded programs, including SPEC CPU2000 and NAS benchmarks.

Lin et al. [27] present an OS-based cache partitioning tool for multicore processors. This tool confirms several key findings already obtained from simulation-based studies while discovering new insights not obtained by simulation. These approaches are different from ours, as they are focused on the usage of cache partitioning as a method to improve the memory performance of shared caches.

Soares et al [28] also propose a method (ROCS) that tries to improve the performance of shared caches at OS-level. Cache unfriendly pages are dynamically mapped to a pollute buffer in the cache. Cache unfriendly pages are detected using the hardware counters at run-time. This buffer for non-reusable page is implemented through page-coloring. Experimental results show that ROCS improves the performance of some SPEC CPU 2000 and NAS benchmarks by up to 34% and 16%, respectively.

8. Conclusions

This paper presents the first model that accurately predicts the behavior of realistic shared caches when executing scientific parallel programs parallelized at loop level without requiring profiling. Regarding the distribution of the iterations of the parallel loops among the threads, the model is general and allows to consider a block cyclic strategy where the block size is configurable.

The validation has been conducted using from small kernels to complete benchmarks from the SPECfp95 and NPB-omp suite, or at least their most significative and time-consuming routines, which proves the large applicability of the model. The predictions of the model for these

codes have been validated against a simulator using a varying number of threads, loop block sizes, and scheduling policies on the cache hierarchies of two state-of-the-art multicore processors. In addition, we have conducted experiments that show that the accuracy of the model is high using a wide variety of cache configurations and problem sizes. The experiments show that the model accurately reflects the influence in the cache behavior of the scheme of parallelization followed (number of threads and block size), no matter if a static or a dynamic scheduling policy is used. The evolution of the number of misses tracked by the model is also reflected in the actual execution time of the actual code on the real computer. Finally, the predictions of the model are always obtained in less than three seconds, even for the most time-consuming benchmarks, which enables its usage in automated optimization processes.

A future line of research is the evaluation of the usage of the model complemented with a CPU model to guide compiler optimizations on parallel applications.

Acknowledgments

This work has been supported by the Galician Government under projects Consolidation of Competitive Research Groups (ref 2010/6), INCITE08PXIB105161PR and UDC/GI-000265, and the Ministry of Education and Science of Spain, FEDER funds of the European Union (Project TIN2010-16735). We also thank the anonymous reviewers for their suggestions, which helped improve the paper. The authors are members of the HIPEAC network.

References

[1] R. Balasubramonian, N. P. Jouppi, N. Muralimanohar, Multi-Core Cache Hierarchies, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2011.

[2] B. B. Fraguela, R. Doallo, E. L. Zapata, Probabilistic Miss Equations: Evaluating Memory Hierarchy Performance, IEEE Trans. on Comp. 52 (3) (2003) 321–336.

[3] J. Xue, X. Vera, Efficient and accurate analytical modeling of whole-program data cache behavior, IEEE Trans. Comput. 53 (5) (2004) 547–566.

[4] C. Cascaval, D. Padua, Estimating cache misses and locality using stack distances, in: Proc. 17th Intl. Conf. on Supercomputing, 2003, pp. 150–159.

[5] X. Vera, N. Bermudo, J. Llosa, A. Gonzalez, A fast and accurate framework to analyze and optimize cache memory behavior, ACM Trans. on Prog. Lang. and Systems 26 (2) (2004) 263–300.

[6] B. B. Fraguela, Y. Voronenko, M. Püschel, Automatic tuning of discrete fourier transforms driven by analytical modeling, in: Intl. Conf. on Parallel Arch. and Compilation Techniques, 2009, pp. 271–280.

[7] OpenMP Architecture Review Board, OpenMP Program Interface Version 3.0 (May 2008).

[8] B. B. Fraguela, R. Doallo, J. Touriño, E. L. Zapata, A compiler tool to predict memory hierarchy performance of scientific codes, Parallel Computing 30 (2) (2004) 225–248.

[9] D. Andrade, M. Arenaz, B. B. Fraguela, J. Touriño, R. Doallo, Automated and accurate cache behavior analysis for codes with irregular access patterns, Concurrency and Computation: Practice and Experience 19 (18) (2007) 2407–2423.

[10] B. B. Fraguela, M. G. Carmueja, D. Andrade, Optimal tile size selection guided by analytical models, in: Proc. of Parallel Computing, Vol. 33, 2005, pp. 565–572, publication Series of the John von Neumann Institute for Computing (NIC).

[11] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, P. Montesinos, SESC simulator (January 2005).

[12] I. Corporation, Intel 64 and ia-32 architectures. software developers manual. volume 3a.system programming guide, part 1 (June 2010).

[13] K. Beyls, E. H. D'Hollander, Generating cache hints for improved program efficiency, J. Syst. Archit. 51 (4) 223–250.

[14] A. Chauhan, C.-Y. Shei, Static reuse distances for locality-based optimizations in matlab, in: Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10, 2010, pp. 295–304.

[15] S. Kim, D. Chandra, Y. Solihin, Fair cache sharing and partitioning in a chip multiprocessor architecture, in: Proc. 13th Intl. Conf. on Parallel Architectures and Compilation Techniques, 2004, pp. 111–122.
[16] R. West, P. Zaroo, C. A. Waldspurger, X. Zhang, Online cache modeling for commodity multicore processors, SIGOPS Oper. Syst. Rev. 44 (2010) 19–29.
[17] D. Chandra, F. Guo, S. Kim, Y. Solihin, Predicting inter-thread cache contention on a chip multi-processor architecture, in: Proc. 11th Intl. Symp. on High-Performance Computer Arch., 2005, pp. 340–351.
[18] Y. Solihin, F. Guo, S. Kim, Predicting cache space contention in utility computing servers, Intl. Parallel and Distributed Processing Symposium 11 (2005) 226b.
[19] C. Xu, X. Chen, R. Dick, Z. Mao, Cache contention and application performance prediction for multi-core systems, in: Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on, 2010, pp. 76 –86.
[20] F. Song, S. Moore, J. Dongarra, L2 cache modeling for scientific applications on chip multi-processors, in: Intl. Conf. on Parallel Processing, 2007, pp. 51 –51.
[21] D. L. Schuff, M. Kulkarni, V. S. Pai, Accelerating multicore reuse distance analysis with sampling and parallelization, in: Proceedings of the 19th international conference on Parallel architectures and compilation techniques, PACT '10, ACM, New York, NY, USA, 2010, pp. 53–64.
[22] M. Kandemir, S. P. Muralidhara, S. H. K. Narayanan, Y. Zhang, O. Ozturk, Optimizing shared cache behavior of chip multiprocessors, in: Proc. 42nd Intl. Symposium on Microarchitecture, 2009, pp. 505–516.
[23] J. Liu, Y. Zhang, W. Ding, M. Kandemir, On-chip cache hierarchy-aware tile scheduling for multicore machines, in: Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on, 2011, pp. 161 –170.
[24] M. M. Baskaran, N. Vydyanathan, U. Bondhugula, J. Ramanujam, A. Rountev, P. Sadayappan, Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors, in: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, 2009, pp. 219–228.
[25] Y. Jiang, E. Z. Zhang, K. Tian, X. Shen, Is reuse distance applicable to data locality analysis on chip multiprocessors?, in: Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction, CC'10/ETAPS'10, 2010, pp. 264–282.
[26] Q. Lu, J. Lin, X. Ding, Z. Zhang, X. Zhang, P. Sadayappan, Soft-olp: Improving hardware cache performance through software-controlled object-level partitioning, in: PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 2009, pp. 246–257.
[27] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, P. Sadayappan, Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems, in: High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on, 2008, pp. 367 –378.
[28] L. Soares, D. Tam, M. Stumm, Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer, in: Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on, 2008, pp. 258 –269.

Diego Andrade received the M.S. and the Ph.D. degrees in computer science from the University of A Coruna, Spain, in 2002 and 2007, respectively. He is a lecturer in the Department of Electronics and Systems of the University of A Coruña since 2006. His research interests focuses in the fields of performance evaluation and prediction, analytical modeling and compiler transformations.

Basilio B. Fraguela received the M.S. and the Ph.D. degrees in computer science from the University of A Coruna, Spain, in 1994 and 1999, respectively. He is an associate professor in the Department of Electronics and Systems of the University of A Coruna since 2001. His primary research interests are in the fields of performance evaluation and prediction, analytical modeling, programmability of parallel systems, design of high performance processors and memory hierarchies, and compiler transformations.

Ramón Doallo, Ph.D in Physics (Univ. Santiago de Compostela) is a Full Professor in Computer Architecture and Technology, and the Head of the Computer Architecture Research Group at University of A Coruña, Spain. He has 22 years of experience in research and development in the area of High Performance Computing (HPC), covering a wide range of topics such as compilers and programming languages for HPC, parallel and distributed algorithms and applications, management of HPC infrastructures, cluster and grid computing, processor architecture, computer graphics, and distributed Geographic Information Systems. He has published more than 120 technical papers on these topics.

```
1function PMEs(S) {
2          D = ∅
3          for it = 0 to ⌈N_i/t_i⌉ − 1 {
4                     foreach reference R ∈ S {
5                               for t = 0 to min{t_i, ⌈(N_i − ⌊it/b_i⌋ × b_i × t_i − it mod b_i)/b_i⌉} − 1 {
6                                         let seqit = ⌊it/b_i⌋ × b_i × t_i + it mod b_i + b_i × t
7                                         let tag = ⌊(seqit × S_Ri + δ_Rj × d_Aj)/L_s⌋
8                                         if ∃p ∈ D | p.tag = tag {
9                                                   F_Ri(RD) += F_R(i+1)(It_i(it − p.it) + RD(p.R, R))
10                                                  D = D − p
11                                        } else {
12                                                  F_Ri(RD) += F_R(i+1)(RD)
13                                        }
14                                        D = D ∪ (tag, it, R)
15                              }
16                    }
17         }
18         return F_Ri ∀R ∈ S
19}
```

Figure A.12: Algorithm to compute the PMEs for a set S of references in translation

Appendix A. PME for inter-reference reuse in the parallelized case

The PME model supports the modeling of the reuses among references in translation to the same array. These are references that have the same stride S_{Ri} with respect to each loop, but which can differ in the added δ constants used in the indexing of one or more dimensions. Most reuse in scientific codes comes from this kind of reference groups. The model sorts the references in translation in descending order of the position they access. The first reference is modeled in all the loops disregarding the other references, i.e., using the intra-reference reuse PMEs. The reason is that it carries no reuse with the other references, as it is the first one to access any line. For each one of the following references, all the loops are modeled in the same way but the one associated with the greatest dimension whose indexing changes with respect to the preceding reference. The reason is that this one will be the dimension in which the reuse takes place. This approach is accurate for references that only differ with the preceding one in the indexing of one dimension. When the references differ in the indexing of several dimensions, it simplifies the problem while providing good estimations in most cases. Such cases are those in which the variations introduced by the differences in the smaller dimensions are much smaller than those produced by the greatest dimension. In the loop where the reuse is modeled a PME that takes into account the reuse with respect to the preceding reference is applied.

Our extension follows the approach in [2] to model references in translation with a modification. This change is applied when the loop that indexes the largest dimension, among those in which the indexing of the references in translation differ, is parallelized. Finding all the possible reuse distances for each reference when there are several references to the same data structure executed in parallel in different threads, using a formula would be very complex. Thus in this case, instead of applying the PME for reuses among references explained in [2], the PME is built applying the algorithm in Figure A.12. The algorithm takes as input an array S containing all the references for which the largest dimension, among those whose indexing is different from the one of the previous reference in the ordering, is indexed by the variable of control of a parallelized loop. The references are stored in S in the order in which they are found in the loop, which is thus the order in which they are executed.

The algorithm emulates an execution of the parallelized loop at nesting level i in $\lceil N_i/t_i \rceil$ steps (line 3). In each step each reference is considered in the order in which it is found in the code (line 4), and up to t_i parallel accesses for the reference are calculated (line 5). The algorithm computes in line 6 the index of the serial iteration corresponding to the current parallel iteration and thread considered. A tag associated to the SOL accessed by the reference is computed in line 7, where S_{Ri} is the stride of all the accesses in S with respect to the considered loop i, δ_{Rj} is the constant in the j-th dimension of R, which is indexed by $\mathtt{I}_i$, and d_{Aj} is the cumulative size of the j-th dimension of the array $\mathtt{A}$ that R accesses. Set D is made up of triplets whose components are a tag representing a SOL accessed during the execution of the loop, the number of parallel iterations, it, in which the most recent access to the SOL took place, and the reference, R, that accessed the SOL in that latest access. If no triplet with the computed tag is found in D, the PME for R at this nesting level $F_{Ri}(RD)$ is updated adding a term $F_{R(i+1)}(RD)$ associated to potential compulsory misses (line 12). If a triplet p is found, then the PME is updated in line 9 with a term $F_{R(i+1)}(\mathrm{It}_i(it - p.it) + RD(p.R, R))$. This means that the PME for the immediately inner level is evaluated with a reuse distance associated to $it - p.it$ iterations of loop i, modified taking into account the reuse distance between references $p.R$ and R. This is achieved by considering $it - p.it + 1$ iterations for the computation of the interference generated by the references found between $p.R$ and R if $p.R$ precedes R in the loop (and, thus, in S), and considering $it - p.it - 1$ iterations for them otherwise. The algorithm updates D with the triplet associated to the current access in line 14. Its output is the set of PMEs for nesting level i for the references in S, returned in line 18.