Address Independent Estimation of the Boundaries of Cache Performance

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

Worst-case (WCET) and best-case (BCET) execution times must be estimated in real-time systems. Worst-case memory performance (WCMP) and best-case memory performance (BCMP) components are essential to estimate them. These components are difficult to calculate in the presence of data caches, since the data cache performance depends largely on the sequence of memory addresses accessed. These addresses may be unknown because the base address of a data structure is unavailable for the analysis or it may change between different executions. This paper introduces a model that provides fast and tight valid estimations of the BCMP, despite ignoring the base address of the data structures. The model presented here, in conjunction with an existing model that estimates the WCMP, can provide base-address independent estimations of the BCMP and WCMP. The experimental results show that the base addresses of the data structures have a large influence in the cache performance, and that the model estimations of the boundaries of the memory performance are valid for any base addresses of the data structures.

Keywords: Real-time systems, BCET, WCET, cache memory

1. Introduction

Execution time must be bounded [1] in real-time systems. Namely, the worst-case execution time (WCET) estimates an upper limit of the execution time of a task and it may be used to perform any schedulability analysis, to ensure meeting deadlines and to assess resource needs for real-time systems. The best-case execution time (BCET) estimates the lower limit and it may be used to assess code quality and resource needs for non- or soft real-time systems, and to ensure that live lines and minimum sampling intervals are met. The presence of data caches [2] complicates the estimation of the memory performance components of the WCET and the BCET, which are the worst-case (WCMP) and best-case memory performance (BCMP), respectively. The reason is that the memory performance in the presence of caches depends largely on the exact

*Corresponding author

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

sequence of memory addresses accessed by the program, and these addresses determine the placement of each piece of data in the cache. This sequence may be unavailable at compile-time due to the lack of the base address information of the data structures and/or the presence of irregular access patterns. The base addresses may also not be obtainable because of the usage of program modules or libraries compiled separately, stack variables or dynamically allocated memory. Furthermore, these addresses may change between different executions of the program.

The model presented in [3] is, to our knowledge, the only one to tackle a base address-independent prediction of the WCMP in the presence of data caches. It is based on the Probabilistic Miss Equations (PME) model [4] and it can estimate rapidly and precisely the WCMP of codes with strided accesses (regular codes). This paper complements [3] with the ability to provide base address independent predictions of the BCMP. This would turn the model, if integrated with a CPU model, into a powerful tool to perform statically a thorough timing analysis.

The rest of this paper paper is organized as follows. Section 2 describes the basics of the PME model and its scope of application. The following two sections explain the changes required to estimate the BCMP. Namely, Sections 3 and 4 contain the core contribution of the paper as they are devoted to describe the method to calculate the BCMP. Section 5 highlights the main differences between the BCMP approach presented in this paper, and the WCMP approach presented in [3]. Then, Section 6 shows some experimental results, Section 7 is devoted to the related work and Section 8 concludes.

2. The Probabilistic Miss Equations model

The Probabilistic Miss Equation (PME) model [4] predicts the behavior of set-associative caches following a Least Recently Used (LRU) replacement policy. Before the model is introduced, let us start with some notions on cache memories. Caches are associative memories [5], located in the top levels of the memory hierarchy, that contain a subset of the data present in the lower levels and that are searched by the memory address provided by the processor. Caches are divided into cache lines, a line corresponding to the minimum amount of information that can be placed in the cache. Cache lines are grouped in cache sets, each set having the same number of lines, which is called the cache associativity. Under a CPU request only the lines in a set are searched. Also, a line can only be placed in a predetermined set, although any of the set lines is eligible for its placement.

The basic operation of a cache memory starts with the processor emitting a memory address, which may need to be translated into a physical address before accessing the cache if we consider a system with virtual memory and a physically-indexed cache. A part of this memory address, called index, is used to select the cache set where the cache line containing the requested address should be located. Another part of the address, called tag, is used to find out if the line is in the cache set. If it is present, the access turns into a cache hit. Then the cache retrieves the data associated to the address requested inside the line using another part of the address called displacement and it sends this data item to the processor. If the line is not present, the access turns into a

miss and the data must be brought from the lower levels of the memory hierarchy to the corresponding cache set. Then, the access is treated as a hit.

When a new line is loaded into its corresponding cache set, the cache set may be full. In that case, one of the lines of the cache set must be replaced by the new line. The selection of which line is replaced is taken by the replacement policy. The most popular replacement policy is the Least Recently Used (LRU) one, that selects the line that has not been accessed for a longer time.

The PME model estimates the number of cache misses generated by the execution of a code. The model processes the static references of the analyzed code one by one. For each reference and each nesting level containing it, a separated Probabilistic Miss Equation (PME) is generated. Each static reference generates several dynamic accesses. Each access affects one data item which is located in a given memory line. This access can result in a cache hit if the line is already loaded in the cache or a miss otherwise. Cache misses take place compulsorily the first time a memory line is accessed. The remaining accesses to memory lines are reuse attempts, given that a preceding access already loaded the line of interest in the cache. A reuse attempt on a memory line results in a miss if the lines brought to the cache during the reuse distance have evicted the line. This eviction happens with a given probability, called miss probability, which depends on the reuse distance. A PME formula classifies the accesses generated by the reference it models within the considered loop according to their reuse distance and computes the miss probability associated to each one of the reuse distances found. Then, this PME estimates the number of misses generated by the reference in the loop by adding the number of accesses with each given reuse distance weighted by their associated miss probability.

$$F_{Ri}(\mathrm{RD}_{\mathrm{input}}) = NAcc(\mathrm{RD}_{\mathrm{input}}) \times MissP(\mathrm{RD}_{\mathrm{input}}) + \sum_{i=1}^{NRD} NAcc(\mathrm{RD}_i) \times MissP(\mathrm{RD}_i) \quad (1)$$

Equation 1 represents the general form of the PME associated to reference R and nesting level i. NRD is the number of different reuse distances found. $NAcc(\mathrm{RD})$ is the number of accesses generated by reference R whose reuse distance is RD and $MissP(\mathrm{RD})$ is the miss probability associated to that reuse distance. The PME also considers the first-time accesses of R to lines during one execution of the loop. While these accesses cannot exploit a RD within the loop, they may enjoy reuse with respect to accesses to the same line which took place in previous iterations of outer or preceding loop nests. Since such RDs cannot be found in the analysis of loop i, every PME F_{Ri} has an input. The number of misses generated by these accesses is accounted by the first term of the formula.

Example 2.1. *Let us consider a simple code like the following*

```
for(ind=0; ind<16; ind++) {
     a[ind] = b[ind];
}
```

and a direct-mapped cache that can store 16 elements of any of both arrays which is

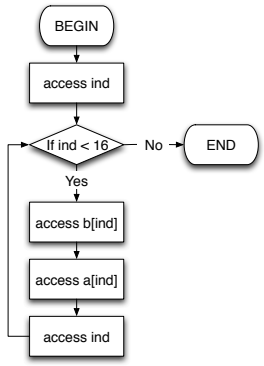

Figure 1: Control flow of the accesses of Example 2.1

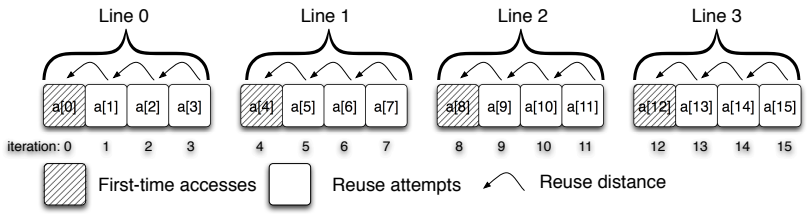

Figure 2: First-time accesses and reuse attempts of the acccess to `a[ind]` in Example 2.1

divided into 4 lines that can store 4 elements of the arrays each. Figure 1 contains a control flow of the accesses generated by the example code. Let us assume that scalar accesses are usually mapped to processor registers, even if it is not the case, that is a reasonable best-case assumption. Thus, only the accesses to arrays `a` *and* `b` *have an impact on the cache.*

Let us see how Eq. 1 applies to the modeling of reference $R \equiv a[ind]$ *in the scope of this loop at nesting level* $i = 0$*. Figure 2 represents the positions of* `a` *accessed through this reference. Accesses to positions located in the same cache line are grouped and, first-time accesses, reuse attempts and reuse distances are identified. The first position of the array* `a[0]` *is located at the beginning of one line, thus, the loop accesses exactly* $16/4 = 4$ *different cache lines. So,* $NAcc(\mathrm{RD}_{\mathrm{input}}) = 4$ *because 4 accesses visit a line for the first time in this loop and its associated reuse distance is* $\mathrm{RD}_{\mathrm{input}}$*, which is the RD for the first-time accesses of R in the loop. The remaining 12 iterations are reuse attempts of a line accessed in a previous iteration of the loop. In fact since the access is sequential, the reuse attempts always take place on the immediately previous iteration of the loop. So, all the reuse attempts are associated to a single reuse distance, thus,* $NRD = 1$*. This reuse distance* RD_0 *is concretely one iteration of the loop 0 we are considering, denoted as* $Iter_0(1)$*. This way,* $F_{R0}(\mathrm{RD}_{\mathrm{input}}) = 4 \times MissP(\mathrm{RD}_{\mathrm{input}}) + 12 \times MissP(\mathrm{RD}_0)$*.*

The example can be completed intuitively although the method to calculate the miss

probability associated to a given reuse distance has not been explained yet. If the lines from array a *have not been accessed in a previous loop, the miss probability associated to the* $\mathrm{RD}_{\mathrm{input}}$ *reuse distance is* 1*, which means that these* $NAcc(\mathrm{RD}_{\mathrm{input}}) = 4$ *accesses turn into misses. Regarding the 12 accesses with reuse distance* $\mathrm{RD}_0 = Iter_0(1)$*, during one iteration of the loop one line from array* a *and one line from array* b *are accessed. The line from* a *is the reused line, so it does not interfere with itself. The line from* b *is going to be placed in one of the 4 sets of the cache, each one of them consisting of a single line. If it is placed in the same set as the reused line from* a*, it will eject it from the cache; otherwise there will be no interference. This way on average, there is a probability of* $1/4 = 0.25$ *that the reused line is ejected from the cache, and this is conversely the miss probability associated to the reuse distance* RD_0 *we are considering. As a result, the average number of misses generated by the studied reference is* $F_{R0}(\mathrm{RD}_{\mathrm{input}}) = 4 \times 1 + 12 \times 0.25 = 7$ ■

Let us notice that this model has two main stages. The first one consists in estimating which is the reuse distance for each access of a reference in order to build the PMEs. The second one estimates the miss probability associated to each reuse distance.

The pseudo-code of Fig. 3 gives an overview of the PME model. In the top-level function, *analyze_code*, the references that appear in each loop nest of the source code are studied one by one. The function *number_of_misses* computes the formula that calculates the number of misses produced by that reference in each nesting level. The pseudo-code of the *number_of_misses* function, and the remaining functions included in this pseudo-code, will be explained later.

The aim of the model in [4] is to estimate an average expected number of misses for the execution of a code. Therefore its formulas and heuristics estimate the average reuse distance associated to each access and the average miss probability for each reuse distance. The model extension presented here for BCMP prediction estimates the smallest possible number of misses a code could generate. Thus it estimates best-case reuse distances, i.e., reuse distances guaranteed to be not larger than those in the actual executions. This extension also computes best-case miss rates for those reuse distances, rather than the average miss probabilities of the original model. This way, Section 3 deals with the construction of best-case PMEs based on best-case reuse distances, and Section 4 with the estimation of the best-case miss rate for a reuse distance.

2.1. Scope of Application

The scope of application of the model introduced in this paper is limited to set-associative caches. The BCMP model introduced in this paper, just as those in [3][4], requires as inputs a representation of the source code to analyze and the cache configuration. The only restriction on the cache is that it must have a LRU replacement policy. The representation of the code can be the source, or if there are optimization steps involved in the generation of the executable, the Abstract Syntax Tree (AST) generated by the compiler after those transformations. This AST reflects the final structure of the executable and contains also all the information required by the model. If the analysis is not performed within the compiler and no such internal representation could be made available for the analysis, optimizations would have to be disabled to ensure the safeness of the prediction.

```
function analyze_code() {
1     foreach loop_nest of the code {
2        foreach reference in the loop_nest {
3           misses+ = number_of_misses(reference, outermost_loop(loop_nest), R_full)
4        }
5     }
6    return misses
}

function number_of_misses(reference, loop, region) {
1    if is_innermost_loop_containing(loop, reference) {
2       return L_R(loop) * bmiss_rate(region) + (N_loop − L_R(loop)) * bmiss_rate(interference_region(loop, 1))
3    } else {
4       misses=0.0
5        foreach inner_loop in inner_loops_containing(loop, reference) {
6           misses+ = L_R(loop) * number_of_misses(reference, inner_loop, region)
7                     +(N_(loop) − L_R(loop)) * number_of_misses(reference, inner_loop, interference_region(loop, 1))
8        }
9       return misses
10    }
}

function interference_region(loop, its) {
1    region_set = ∅
2    foreach reference in the loop {
3       case ... {
4          ...→ region_set = region_set ∪ Reg_s(M)
5          ...→ region_set = region_set ∪ Reg_r(N_R, T_R, S_R)
6          ...
7       }
8    }
9    return region_set
}

function bmiss_rate(region_set) {
1    global_AV = empty_AV()
2    union_AV = empty_AV()
3    foreach region in the region_set {
4       case region {
5          Reg_s(M) → minUnionAV(global_AV, AV_s(M), union_AV)
6          Reg_r(N_R, T_R, S_R) → minUnionAV(global_AV, AV_r(N_R, T_R, S_R), union_AV)
7          ...
8          global_AV = copy(union_AV)
9       }
10    }
11    return union_AV_0
}
```

Figure 3: The PME model algorithm

The model supports codes with any number of perfectly or non-perfectly nested loops with an arbitrary number of references, which must be indexed using affine functions of the loop indices. There may be several references to the same data structure at any nesting level. Actually, any memory reference can be found at any nesting level. The number of iterations of the loops, or at least a lower bound for the BCMP prediction, must be known to perform the analysis. This information can be inferred from the code, provided by the user, or found through profiling. Inter-routine cache effects are modeled using inlining, either symbolic or actual.

Conditional statements may appear in the code analyzed when they guard accesses to registers or to the latest data item accessed before the branch. In these cases the data-dependent flow does not alter the cache behavior. Similar restrictions are found in other analytical models of the data cache behavior [6][7][8]. Pieces of code containing irregular access pattern, due to the usage of pointers, indirections or more complex conditional statements, can be analyzed by locking the cache during its execution. Cache locking can be used to improve cache predictability, which is necessary to estimate a safe WCET [7]. In multitasking environments, predictability can be improved by dividing the cache into disjoint partitions which are assigned to different concurrent tasks [9]. As a proof of the large applicability of the method proposed in this paper in real-world situations, these are exactly the same assumptions of the original PME model [4], which sufficed to model complete SPECfp95 and Perfect Benchmarks applications, or at least their most significative and time-consuming routines.

The approach presented in this paper calculates the best-case behavior of the cache independently of the performance of the rest of the system. The success of this approach depends on that the architecture is fully timing compositional [10], as the model does not take into account those timing anomalies [11] where a cache hit may result in longer execution times than a miss for a given path due to overlapped structural resource conflicts.

3. Best-case PME construction

Our model builds a PME F_{Ri} to estimate the number of misses that each static reference R generates during one execution of each loop i that encloses it. Building a PME consists in multiplying the number of dynamic accesses generated by R that can enjoy each potential reuse distance (RD) within loop i by the miss probability associated to that RD, as Eq. 1 summarized.

Let us now explain the construction of PMEs. Loops are normalized for the analysis to have step 1 and they are numbered from 0, the outermost one, to Z, the innermost loop containing reference R. In order to simplify the explanation, all the strides and sizes will be expressed in number of elements of the data structure considered rather than in bytes. As we have explained, the PME for each nesting level captures all the reuse distances within that level. In particular, the construction of F_{Ri} discovers the reuse distances that are specifically associated to loop i. If i is not the innermost loop containing R, the PME is written in terms of the PME for the immediately inner level $F_{R(i+1)}$, which carries the reuse distances within the inner loops. In the innermost loop Z containing R, we define $F_{R(Z+1)}(\mathrm{RD}_{\mathrm{input}})$ as the PME for an individual access of R. So, it is in this PME where the recursion finishes, being $F_{R(Z+1)}(\mathrm{RD}_{\mathrm{input}})$ equal to the

miss probability associated to $\mathrm{RD}_{\mathrm{imput}}$, $MissP(\mathrm{RD}_{\mathrm{input}})$. This way, although PMEs are built recursively, their final appearance is that of the general form explained in Eq. 1. In the calculation of the BCMP, rather than miss probabilities, best-case miss rates are computed following the function developed in Section 4. This way, in the BCMP model $F_{R(Z+1)}(\mathrm{RD}_{\mathrm{input}}) = BMissR(\mathrm{RD}_{\mathrm{input}})$, the best-case miss rate associated to RD. This best-case miss rate is calculated by considering the placement of the data structures in the cache which minimizes the miss rate.

The first step to build F_{Ri} in each loop containing R, $0 \leq i \leq Z$, is to verify whether R can exploit reuses with a RD associated to this loop with respect to other references to the same data structure (inter-reference reuses) or not. The model can analyze inter-reference reuse among uniformly generated references, that is, references with the same affine indexing functions except possibly an added constant (i.e. `A[i][2*j]` and `A[i][2*j+5]`). The techniques used to model best-case inter-reference reuse are analogous to those employed in [3] in the calculation of the WCMP. If there is no inter-reference reuse, the PME is built taking into account the potential reuses of R with respect to its own accesses (intra-reference reuses). The subsequent explanation focuses on these reuses.

The PME is built based on the fact that R has a constant stride S_{Ri} with respect to any enclosing loop i. The reason is that each dimension j of R is indexed by an affine function $\alpha_{Rj} \cdot \mathtt{I}_m + \delta_{Rj}$ of the control variable of loop m, $\mathtt{I}_m$, where α_{Rj} and δ_{Rj} are constants. Therefore,

$$S_{Ri} = \begin{cases} 0 & \text{if } \mathtt{I}_i \text{ does not index } R \\ |\alpha_{Rj}| \cdot D_{aj} & \text{if } \mathtt{I}_i \text{ indexes dimension } j \text{ of } R \end{cases} \tag{2}$$

where `a` is the data structure referenced by R and D_{aj} is the cumulative size of dimension j of array `a`. If `a` is a n-dimensional array of size $d_{a1} \times d_{a2} \times \cdots \times d_{an}$ with row-major layout (as in the C language), $D_{aj} = \prod_{i=j+1}^{n} d_{ai}$. Notice that S_{Ri} is non-negative because the absolute value of α_{Rj} is used to compute it. This simplifies the treatment for negative strides. Table 1 depicts these parameters and others that will be referenced during the explanation of the model.

The concept of iteration point of reference R at level i is crucial to understand the rest of the explanation.

Let A be the memory address accessed by R during the first iteration of loop i for a given IP_{Ri}. Then, since R has constant stride S_{Ri} with respect to loop i, in the subsequent iterations of loop i the addresses accessed by R for this IP_{Ri} will be $A + S_{Ri}$, $A + 2 \cdot S_{Ri}, \ldots, A + (N_i - 1) \cdot S_{Ri}$, where N_i is the number of iterations of loop i. This reasoning holds for any IP_{Ri}. Thus, the minimum number of different lines that R can access in any IP_{Ri} during the execution of loop i is

$$L_{Ri} = 1 + \lfloor (N_i - 1) / \max\{L_s / S_{Ri}, 1\} \rfloor , \tag{3}$$

L_s being the size of a cache line in elements of the considered access. The actual number of lines accessed depends on the alignment of the first address accessed (A) with a cache line. Expression (3) assumes that this address is mapped to the beginning of a cache line, that is, $A \bmod L_s = 0$. This assumption minimizes the number of different lines accessed. This is one of the differences between the BCMP and the

C_s	Cache size
d_{aj}	size of the j-th dimension of array a
D_{aj}	cumulative size of the j-th dimension of an $n-$dimensional array a, $D_{aj} = \prod_{l=j+1}^{n} d_{al}$
D_{Ri}	minimum # of different lines an iteration point of a reference R can potentially access during the execution of the loop at nesting level i
k	Associativity of the cache
L_{Ri}	minimum # of different lines accessed by an iteration point of a reference R during the execution of the loop at nesting level i
L_s	Line size
N_i	minimum # of iterations of loop at nesting level i, whose index is $\mathtt{I}_i$
S	Number of cache sets
S_{Ri}	stride of reference R with respect to the loop at nesting level i, $S_{Ri} = \alpha_{Rj} \cdot d_{aj}$, where j is the dimension of array a referenced by R indexed by $\mathtt{I}_i$
Z	innermost loop containing reference R

Table 1: Notation used (all the strides and sizes are expressed in number of elements of the data structure considered rather than in bytes).

WCMP approaches. In the WCMP approach, this equation assumes the alignment that maximizes the number of different lines accessed. In each iteration of the innermost loop containing the reference, one iteration point is accessed in each iteration of the loop. In the remaining nesting levels, a set of iteration points are accessed in each iteration of the loop. In L_{Ri} iterations all the IP_{Ri} access a line different from that accessed by the corresponding iteration point of the previous iterations of loop i. Thus, the potential reuse distance RD for such accesses is unknown in this nesting level. They may reuse lines already accessed in previous iterations of outer loops or they become cold misses.

When $L_{Ri} < N_i$, in the remaining $N_i - L_{Ri}$ iterations of loop i, each IP_{Ri} attempt to reuse a line accessed by the corresponding iteration point of the previous iteration of loop i. This is necessarily the case because R has a constant stride with respect to loop i. This implies that the RD for these accesses is one iteration of the loop at nesting level i. As a result,

$$F_{Ri}(\mathrm{RD}_{\mathrm{input}}) = L_{Ri} \cdot F_{R(i+1)}(\mathrm{RD}_{\mathrm{input}}) + (N_i - L_{Ri}) \cdot F_{R(i+1)}(\mathrm{Iter}_{Ri}(1)) \tag{4}$$

where $\mathrm{Iter}_{Ri}(1)$ is a reuse distance of one iteration of the loop at nesting level i. Now we can see why this is a best-case PME. The longer a RD is, the larger the miss rate it generates is. Since the $\mathrm{RD}_{\mathrm{input}}$ for the first-time accesses within the loop, which is the input argument of the PME, is necessarily longer than $\mathrm{Iter}_{Ri}(1)$, our minimization of L_{Ri} ensures associating the minimum possible number of reuses to the longest RD. Let us notice that this PME matches the general form presented in Eq. 1, once the PMEs for the inner nesting levels have been composed and $F_{R(Z+1)}(\mathrm{RD}_{\mathrm{input}})$ has been substituted by $BMissR(\mathrm{RD}_{\mathrm{input}})$ in the innermost loop containing the reference.

The *number_of_misses* function in the pseudo-code of Fig. 3 provides an imple-

```
for(ind=0; ind<4; ind++)   // Level 0
  for(j=0; j<4; j++)       // Level 1
    for(i=0; i<4; i++)     // Level 2
      a[j][i] += b[i][j] + c[ind][i]
```

Figure 4: Example code

mentation for the construction of the PMEs based on the recursive form of Eq. 1 . For simplicity, this pseudo-code only considers intra-reference reuses but the model also supports inter-reference reuses. The two remaining functions (*bmiss_rate* and *interference_region*) will be explained later.

Example 3.1. *Let us analyze reference* `b[i][j]` *in the code in Fig. 4. Let us consider a direct-mapped (*$k = 1$*) which can store 32 elements of any of the matrices (*$C_s = 32$*) and where each line can store 2 elements (*$L_s = 2$*). The matrices are* 4×4*, and they are stored by rows. First, the equation* F_{R2} *for the innermost loop for that reference is derived. Since* $N_2 = 4$ *and Eq. 2 yields* $S_{R2} = 4$*, then according to Eq. (3) we have* $L_{R2} = 4$*. The resulting equation is obtained substituting these values in Eq. 4*

$$F_{R2}(\mathrm{RD}_{\mathrm{input}}) = 4 \cdot F_{R3}(\mathrm{RD}_{\mathrm{input}}) + 0 \cdot F_{R3}(\mathrm{Iter}_{R2}(1)) \tag{5}$$

that is, the reference accesses new lines in the four iterations of this loop. As for loop in nesting level 1, since here $N_1 = 4$ *and* $S_{R1} = 1$*, then from Eq. (3)* $L_{R1} = 2$ *and*

$$F_{R1}(\mathrm{RD}_{\mathrm{input}}) = 2 \cdot F_{R2}(\mathrm{RD}_{\mathrm{input}}) + 2 \cdot F_{R2}(\mathrm{Iter}_{R1}(1)) \tag{6}$$

that is, in the best case, this loop generates accesses to two different lines as well as two reuses for each iteration point within loop 1. In the outermost loop, $N_0 = 4$ *and* $S_{R0} = 0$*, and from Eq. (3)* $L_{R0} = 1$*. Thus,*

$$F_{R0}(\mathrm{RD}_{\mathrm{input}}) = 1 \cdot F_{R1}(\mathrm{RD}_{\mathrm{input}}) + 3 \cdot F_{R1}(\mathrm{Iter}_{R0}(1)) \tag{7}$$

that is, in the best case, this loop generates accesses to one line and three reuses for each iteration point within loop 0. When the equations are composed and $F_{R3}(\mathrm{RD})$ *is replaced by* $BMissR(\mathrm{RD})$*, the final number of misses for reference* `b[i][j]` *can be calculated as*

$$F_{R0}(\mathrm{RD}_{\mathrm{input}}) = 8 \cdot BMissR(\mathrm{RD}_{\mathrm{input}}) + 24 \cdot BMissR(\mathrm{Iter}_{R0}(1)) + 32 \cdot BMissR(\mathrm{Iter}_{R1}(1)) \tag{8}$$

This equation indicates that 8 of the accesses of `b[i][j]` *in this loop cannot exploit reuses within the loop. Thus their reuse distance* $\mathrm{RD}_{\mathrm{input}}$ *depends on previous loop nests. Other 24 accesses attempt to reuse a line accessed in the previous iteration of loop 0. Thus their miss rate depends on the memory regions accessed during such iteration* $\mathrm{Iter}_{R0}(1)$*. Finally, the remaining 32 accesses try to reuse a line accessed in the previous iteration of loop 1 and its associated reuse distance is* $\mathrm{Iter}_{R1}(1)$ ■

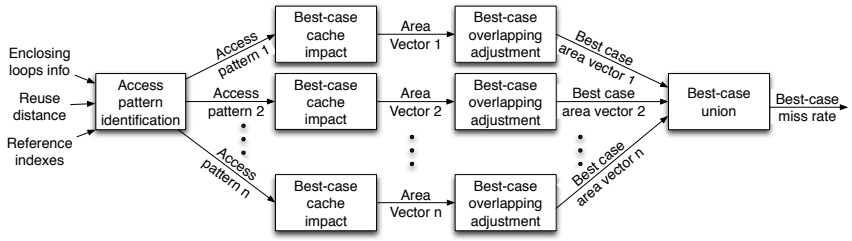

Figure 5: Best-case miss probability estimation for a reuse distance

4. Best-case estimation of the miss rate for a reuse distance

As Section 2 explains, the estimation of the BCMP requires calculating a lower bound of the miss rate associated to each reuse distance (RD) found in a PME. This miss rate corresponds to the minimum rate of the lines to reuse that may have been evicted from the cache by other lines loaded during the RD. The estimation of the best-case miss rate consists of several steps: the access pattern identification, the best-case cache impact estimation, the best-case overlapping between the reused and the interfering regions and the best-case area vectors union. These steps are illustrated in Figure 5 and they are now discussed in turn.

4.1. Access pattern identification

Let us define in advance the concepts of access pattern and memory region that will be used throughout this section. This step derives in 3 stages the access patterns followed by the references found within the RD from the indexing functions of these references and the shape of the loops enclosing them:

1. The first stage computes for each reference and for each dimension of the data structure the reference it refers to, the number of elements accessed during the RD, as well as the stride between each two consecutive elements. As a result, this stage computes the memory region accessed by each reference.
2. When there are several references to the same data structure, the regions they access often overlap. Regions that overlap are merged in order to avoid considering the overlaps several times as source of interference.
3. From the shape of each memory region, an associated access pattern is inferred. From this point we will use indistinctively the terms memory region and access pattern: the shape of a memory region defines an access pattern, and an access pattern defines the memory region comprised by the elements it accesses. Two access patterns have been found in the codes considered in [4] and in this paper: the sequential access to M elements, denoted as $\mathrm{Reg_s}(M)$, and the access to N_R groups of T_R consecutive elements each, separated by a constant stride S_R, denoted as $\mathrm{Reg_r}(N_\mathrm{R}, T_\mathrm{R}, S_\mathrm{R})$.

The estimation of the BCMP instead of the average-case memory performance or the WCMP does not require changes for this step. A more detailed description of this step is found in [12].

The pseudo-code of the *interference_region* function in Fig. 3 shows a simplified implementation of the access pattern identification step described in this section. As the details of this method, already explained in [12], have been omitted in this paper, the implementation of this function also omits the corresponding parts of the method. The function explores each reference in the reuse distance and it identifies the access pattern followed by that reference in that context. These access patterns are stored in the *region_set* data structure in Fig. 3.

Example 4.1. *Example 3.1 derived Eq. 8 to estimate the number of misses generated by reference* `b[i][j]`. *The only unknown quantities of this equation are* $BMissR(\text{Iter}_{R0}(1))$ *and* $BMissR(\text{Iter}_{R1}(1))$*, the best-case miss rate associated to one iteration of loops 0 and 1 respectively. The calculation of this miss rate requires to find out the access patterns followed by references to matrices* `a`*,* `b` *and* `c` *during one iteration of loops 0 and 1 respectively. Let us recall that the three matrices have* 4×4 *elements.*

Regarding the reuse distance $\text{Iter}_{R1}(1)$*, the reference* `a[j][i]` *generates the access to 4 consecutive elements. This way, it implies a sequential access to 4 elements,* $\text{Reg}_s(4)$*. Reference* `b[i][j]` *generates the access to 4 elements of a column, and this access is identified as* $\text{Reg}_r(4, 1, 4)$ *because* $L_s \leq 4$*, that is, the access to 4 groups of 1 element each, separated by a constant stride 4. Finally, the reference* `c[ind][i]` *generate the access to 4 consecutive elements,* $\text{Reg}_s(4)$.

Regarding the other reuse distance $\text{Iter}_{R0}(1)$*, references* `a[j][i]` *and* `b[i][j]` *generates the access to the 16 elements of matrices* `a` *and* `b` *respectively. So, both access patterns are identified as* $\text{Reg}_s(16)$*. Let us notice that although one access is performed by columns and the other by rows, the impact of both matrices on the cache is similar. Finally, reference* `c[ind][i]` *generates the access to 4 consecutive elements,* $\text{Reg}_s(4)$ ■

4.2. Best-case cache impact estimation

This second stage quantifies the impact of the cache lines loaded by each access pattern on the miss rate associated to a reuse distance by means of a vector of ratios called area vector (AV). This is a generic definition of the concept of AV. The group of lines considered depends on the kind of estimation to obtain. For example, in the model presented in [4], the average-case AV is calculated considering the group of all the existing memory lines. This is equivalent to calculating the ratio of cache sets that receives a given number of lines from the studied access pattern. The reason is that all the memory lines are uniformly distributed along all the cache sets. The best-case miss rate presented used for the BCMP estimation requires considering a more limited group of lines. Namely, the lines accessed by the access pattern associated to the reference *R* for which the PME is calculated. In a first stage, explained in this section, the group of all the existing memory lines and the best-case alignment are considered. Section 4.3 processes this AV to take into account only the limited group of lines associated to *R*. So, it selects the best-case placement of the access pattern considered with respect to the lines of *R*.

Section 4.1 explained that the two most common access patterns found in regular codes were: the access to consecutive elements (sequential access), and the access to groups of consecutive elements, separated by a constant stride (strided access). The

estimation of the best-case cache impact for these two access patterns expressed as an AV is explained now in turn.

□ *Best-case cache impact estimation for the sequential access* . The best-case AV associated to the access to M consecutive elements, $\mathrm{Reg_s}(M)$, must reflect the alignment of the M elements with respect to the cache lines that involves fewer lines. This is another part of the model where the BCMP and WCMP approaches differ. The WCMP approach reflects the alignment that involves more lines, and thus, it maximizes the impact of these regions on the cache. In the best-case approach, the placement that involves fewer lines as in the case of the L_{Ri} value discussed in Section 3, is the one where the first of the M elements accessed is the first of a cache line. This way, the access to the M elements involves $\lceil M/L_s \rceil$ cache lines. The average number of lines placed in each set for this best-case alignment is $l = \min\left\{k, \frac{\lceil M/L_s \rceil}{S}\right\}$ since there are S sets in the cache and each one of them cannot hold more than k lines (the associativity). Based on this, the best-case area vector $\mathrm{AV_s}(M)$ for this access pattern can be computed as

$$\begin{array}{l} \mathrm{AV}_{\mathrm{s}_{(k-\lfloor l \rfloor)}}(M) = 1 - (l - \lfloor l \rfloor) \\ \mathrm{AV}_{\mathrm{s}_{(k-\lfloor l \rfloor - 1)}}(M) = l - \lfloor l \rfloor \\ \mathrm{AV}_{\mathrm{s}_i}(M) = 0 \qquad 0 \leq i < k - \lfloor l \rfloor - 1, k - \lfloor l \rfloor < i \leq k \end{array} \tag{9}$$

where the i-th element of the AV is the ratio of sets that have received $k - i$ lines of this access pattern. The exception is the component 0, which is the ratio of sets that have received k or more lines. These values can be also interpreted as the ratio of all the memory lines that must compete with a given number of lines of the considered access pattern. As l is not necessarily an integer, there is a ratio of $l - \lfloor l \rfloor$ cache sets that have received $\lfloor l \rfloor + 1$ lines from this pattern, and the remaining sets (a ratio of $1 - (l - \lfloor l \rfloor)$) receive $\lfloor l \rfloor$ lines from this pattern. The remaining components of the AV are zeroed, as they do not correspond to any ratio of lines.

□ *Best-case cache impact estimation for the strided access.* The best-case AV associated to the access to N_R groups of T_R consecutive elements each, separated by a constant stride S_R, $\mathrm{Reg_r}(N_\mathrm{R}, T_\mathrm{R}, S_\mathrm{R})$, is calculated using a method analogous to the one described in [3] for the worst-case AV. This method considers all the possible relative offsets of the first element of the access pattern in a line and calculates its associated AV. All these AVs are tried in the next stages of the miss rate calculation. Finally, the AV of the relative offset with the largest miss rate is selected. The best-case AV is calculated analogously, but the offset that gives place to the smallest miss rate is selected. So, this is other part of the model where the BCMP and the WCMP approaches are different.

Example 4.2. *The Example 4.1 continued Example 3.1 and it identified the access patterns of the references accessed during the* $\mathrm{Iter}_{R0}(1)$ *and* $\mathrm{Iter}_{R1}(1)$ *reuse distances.*

Let us focus in the $\mathrm{Iter}_{R0}(1)$ *RD (one iteration of the outermost loop). During this RD, 16 consecutive elements of matrices* `a` *and* `b` *are accessed (*$\mathrm{Reg_s}(16)$*), and 4 consecutive elements of matrix* `c` *(*$\mathrm{Reg_s}(4)$*).*

As both access patterns are sequential, their impact on the cache can be estimated using Eq. 9. Let us consider the cache configuration used in Example 3.1. The calculation of the impact of the access to 4 consecutive elements, $\mathrm{Reg_s}(4)$ *starts with*

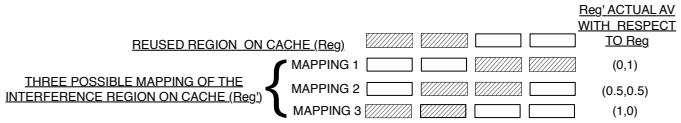

Figure 6: AV depending on the relative positions of the reused and the interfering memory regions

the calculation of the average number of lines placed on each set for this best-case alignment $l = \min\left\{k, \frac{\lceil M/L_s \rceil}{S}\right\}$*,* k *being 1 (the cache associativity),* $M = 4$*,* $L_s = 2$ *and* $S = C_s/(L_s \times k) = 16$*. Thus,* $l = \min\left\{1, \frac{\lceil 4/2 \rceil}{16}\right\} = 0.125$*. Then, when* l *is replaced in Eq. 9 the resulting AV is* $(0.125, 0.875)$*.*

In the case of the estimation of the impact of $\mathrm{Reg}_s(16)$*,* $l = \min\left\{1, \frac{\lceil 16/2 \rceil}{16}\right\} = 0.5$ *as now* $M = 16$ *and when* l *is replaced in Eq. 9 the resulting AV is* $(0.5, 0.5)$ ■

4.3. Best-case overlapping between reuse and interfering regions

The success of an attempt of reuse of several lines, which we will call reuse region `Reg` in what follows, depends on the relative placement of these lines with respect to those of each interfering region `Reg'` found in the considered reuse distance (RD). The procedures described in Section 4.2 provide an AV for each `Reg'` where its components are ratios of all the memory lines that compete with a given number Y of lines from this region, no matter these sets contain lines from `Reg` or not. This is a fair average estimation, as the relative placement in the cache of the lines from different memory regions is unknown to the model. Nevertheless, the actual ratios of interference between `Reg` and `Reg'` can be much smaller depending on the actual placement, and corresponding overlapping, of both sets of lines on the cache. Thus, the components of the AV associated to each region `Reg'` must be processed to contain the ratio of lines of the group compounded by the lines of `Reg` that compete in their cache sets with Y lines from `Reg'`.

Example 4.3. *Figure 6 represents the mapping on a one way (direct mapped) cache with four cache sets of a reuse region* `Reg` *and three possible mappings of an interference region* `Reg'`*. Since* `Reg'` *fills two of the four cache sets and leaves empty the other two, its AV is (0.5, 0.5). This suggests an average 50% rate of conflict. However, the actual interference with the lines of* `Reg` *depends on the relative mapping of both sets of lines in the cache and is represented by the AV placed on the right side of each mapping in the figure. In this AV, component 0 is the ratio of lines of* `Reg` *that collide with* `Reg'` *for this mapping, and component 1 is the ratio of lines that do not collide. This way, the first mapping does not interfere with the reuse of* `Reg`*, the second mapping only interferes with the reuse of one of its lines, and the third one avoids both reuses, which leads to 0, 50 and 100% miss rate, respectively* ■

As we see the AVs must be modified to provide conflict rates, i.e., ratios of lines of the reuse region `Reg` that collide in their set with a given number of lines from `Reg'`. These conflict rates must be computed considering the best-case overlapping of both

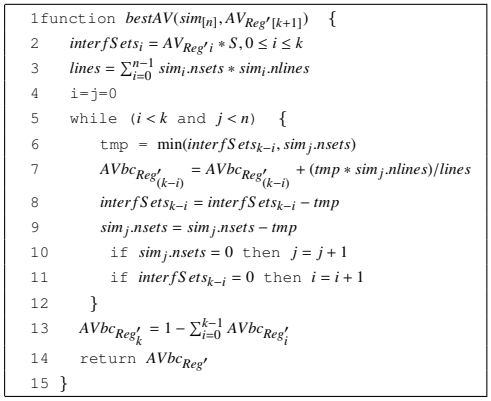

Figure 7: Calculation of best-case AV

regions in the cache. So, this part of the model is performed differently in the BCMP and the WCMP approaches, as the WCMP one considers the worst-case overlapping. The best-case will be the one in which the largest possible number of lines from Reg have to compete with the smallest possible number of lines from Reg' in their cache set. That is, it is the situation in which the largest possible number of lines from Reg are mapped to the $AV_{Reg'_k} \cdot S$ sets that receive 0 lines from Reg', where $AV_{Reg'}$ is the AV associated to Reg', k is the associativity, $AV_{Reg'_k}$ the $k-$th component of $AV_{Reg'}$ and S is the number of sets in the cache. Component k of $AVbc_{Reg'}$, the AV for Reg' considering this best-case alignment, is then the ratio of lines of Reg mapped to these sets without lines of Reg'. Once those sets without lines from Reg' are exhausted, then the largest possible number of lines from Reg are mapped to the $AV_{Reg'_{(k-1)}} \cdot S$ sets that receive 1 line from Reg'. These lines contribute to the $AVbc_{Reg'_1}$ component. The remaining lines from Reg are processed analogously. As we see this algorithm requires $AV_{Reg'}$ as well as the distribution of lines of Reg per set in order to match them with the lines from Reg'. Unfortunately AV_{Reg} does not suffice for this because its component 0 does not provide the exact number of lines per set, just that there are k or more lines, which is not enough to estimate the ratios. Let us see this with an example. In Fig. 6 if region Reg had occupied three sets with a single line each, its AV would have been $(0.75, 0.25)$, and since one of its lines would have collided in the best case with Reg', $AVbc_{Reg'}$ would have been $(0.33, 0.66)$. Now, if Reg had mapped two lines to set 0, another two to set 1 and another line to set 2, its AV would have also been $(0.75, 0.25)$, but since 1 out of its 5 lines could collide with Reg' in the best case, $AVbc_{Reg'}$ would have been $(0.2, 0.8)$. Thus the calculation of $AVbc_{Reg'}$ requires a simulation of the distribution of the lines of Reg on the cache. The simulation output is a vector of pairs *sim* where each pair i has two components $sim_i.nsets$ and $sim_i.nlines$, such that *nsets* cache sets received *nlines* cache lines from region R. The elements of the vector are sorted in decreasing order of $sim_i.nlines$.

The algorithm bestAV in Fig. 7 calculates $AVbc_{Reg'}$ from vector *sim* and $AV_{Reg'}$ following the procedure explained. In this function, array indexing is 0-based and the input vectors are subindexed with their size. This way, n is the size of vector *sim* while

k stands for the associativity of the cache, $k+1$ being the size of the input AV. The algorithm matches the cache sets that receive the maximum number of lines of the reuse region, as *sim* is processed in decreasing order of $sim_i.nlines$, with the cache sets that receive the minimum number of lines from the interference region, as $AV_{Reg'}$ is processed from component k to component 0.

If the reuse region `Reg` changes its relative position in the cache in different iterations of outer loops, it may be impossible that the best-case overlapping takes place in all the iteration of those loops. The same holds if the interfering region is the one to change its position with the iterations of outer loops. Thus tightness could be improved taking into account the freedom of placement of `Reg` and `Reg'` due to the change of their relative position in the cache in different iterations of outer loops. However, the experimental results showed that the tightness of the BCMP prediction is already very good without taking into account this freedom of movement, therefore it has not been considered in the model.

Example 4.4. *The example 4.2 calculated the impact of the references accessed during the RD corresponding to one iteration of the outermost loop (*$\text{Iter}_{R0}(1)$*). This RD is one of those identified in example 4.1 when the PME associated to references* `b[j][i]` *was derived. The AVs obtained in this process were* $(0.5, 0.5)$ *for the* `a[i][j]` *and* `b[j][i]` *references, and* $(0.125, 0.875)$ *for the* `c[ind][i]` *reference. Now the AVs derived in this previous example must be turned into AVs that reflect the best-case overlapping between the reused region and the interference region. This conversion is done using the algorithm in Fig. 7.*

Let us see how this process is done for the region associated to reference `a[i][j]`*, whose AV is* $(0.5, 0.5)$*. Let us consider the cache configuration used in Example 3.1. The reused region (generated by reference* `b[j][i]`*) is composed of 16 consecutive elements. In the best-case, these 16 elements are spread along 8 cache lines, thus, the structure sim has two components* $n = 2$*. In the first component,* $sim_0.nsets = 8$ *and* $sim_0.nlines = 1$*, and in the second component* $sim_1.nsets = 8$ *and* $sim_1.nlines = 0$*. The other input of the algorithm is the AV of the interfering region,* $AVReg' = (0.5, 0.5)$*. Each component of this AV is multiplied by the number of cache set (*$S = 16$*) in line 2 of the algorithm yielding a vector* $interfSets = (8, 8)$*. The number of lines occupied by the reused region, calculated in line 3, is* 8*. As the cache has 16 cache sets (or lines as it is direct-mapped), the loop between lines 5 and 12 matches the 8 cache sets (or lines) containing one line of the the reused region and the 8 sets containing no lines from the interfering region. As a result, the final* $AVbc_{Reg'}$ *is* $(0, 1)$*.*

In the case of reference `c[ind][i]`*, its AV is* $(0.125, 0.875)$*. Vector sim has the same contents as the reused region is the same and* $AVReg' = (0.125, 0.875)$*. As a result,* $interfSets = (2, 14)$*. In this case, the 2 cache sets that receive 1 line from the interfering region are matched with two of the 8 cache sets receive no lines from the reused region. Out of the 14 cache sets that do not receive any line from the interfering region, 8 are matched with the 8 sets that receive the region to reuse (generating no interference, as they have no lines), while the other 6 are matched with the remaining 6 cache sets that receive no lines from the reused region. As a results, the final* $AVbc_{Reg'}$ *is* $(0, 1)$ ■

4.4. Best-case area vectors union

The previous steps for calculating the best-case miss rate for a reuse distance (RD) generated one AV per memory region accessed during that RD. Those AV represent the contribution of each region to the miss rate associated to the RD. This final step of the process summarizes the effects of all the memory regions defined by the accesses found within the RD by merging their AVs into a global one. Then, component 0 of this AV will be the ratio of lines to reuse located in a set where k or more lines (from other memory regions) have been placed during the RD, k being the associativity of the cache. Since in a k-way cache with LRU replacement, a line is evicted from its set before a subsequent reuse if and only if k or more different lines are placed in its set before such reuse, component 0 of the global AV is the miss rate associated to the RD. The process to compute the global AV considers how the different memory regions involved may overlap in the cache and it varies depending on the purpose of the model. The model in [4] uses a method based on independent probabilities to calculate the average-case miss probability. This method processed the AVs two by two but it is not suitable to calculate the best-case miss rate, because it simply computes the probability that a given set receives certain number of lines. In the method to calculate the worst-case miss rate presented in [3], all the AVs had to be considered at the same time to generate the largest interference. This approach, or an analogous one, is not valid to calculate the best-case AVs union. Instead, we present a novel approach which combines the AVs in groups of two untill one global best-case AV is obtained. This way, again, this part of the model is different in the BCMP and the WCMP approaches.

Example 4.5. *Let us consider a 1-way cache and a reuse distance in which two memory regions have been found, the AVs for both being (0.5, 0.5). This means that, for both regions, half of the lines to reuse collide in their set with one or more of their lines. Now, depending on the relative position of both interfering memory regions and the lines to reuse, there are two extreme scenarios. In the WCMP situation, one of the memory regions collides with one half of the lines to reuse, and the other region with the other half. This would mean that it would be impossible to reuse any line. The global AV would be (1, 0) and the miss rate, 1 (100%). In the BCMP situation the lines of the two interfering regions would be mapped to the same cache sets, colliding then with the same half of the lines to reuse. As a result the other half could be successfully reused. The global AV would be (0.5, 0.5) and the miss rate 0.5 (50%).*

4.4.1. Operation `minUnionAV`

We have developed a `minUnionAV` operation that combines the components of two input AVs in order to estimate the best-case resulting global AV. This will be the AV with the smallest possible component 0, i.e., miss rate. The operation is commutative and associative, and it is applied repetitively on the AVs found within a RD when there are more than two. We first explain the rationale for the algorithm and illustrate it with an example.

The strategy to derive the best-case global AV of two input AVs A and B is to process the elements of these AVs one by one. Since the AVs are defined on the same cache, each component of A has to overlap (i.e, correspond to the same cache sets) with some component(s) of B that add up the same ratio and viceversa. Each one of these

Figure 8: Example of mapping considering the best alignment

overlapped ratios will be added to a component of the resulting AV that represents the combined effect of the AVs A and B, which we will call U through this explanation. This way, if a ratio of cache sets p of A_j overlaps with the same ratio of cache sets of B_t, this ratio p must be subtracted from both A_j and B_t. According to the definition of area vector, the ratio of cache sets that hold s lines is located in the component $max(k-s,0)$ of the AV. Since A_j and B_t provide $k-j$ and $k-t$ lines respectively to that ratio p of cache sets, a total of $2k-j-t$ lines are mapped to each one of those sets. As a result this ratio p must be attributed to component $max(k-(2k-j-t),0)$ of the resulting AV U.

The best-case algorithm must find the overlapping that minimizes the component 0 (ratio of full cache sets) of the resulting AV U. The strategy to achieve this is to process the ratios of each input AV starting from the leftmost one to the rightmost one in two stages. The first stage tries to avoid contributing to U_0 by overlapping the ratio considered with the righmost components of the other AV, that is, those that represent sets with the fewer possible number of lines. Concretely, during the processing of the component in position j in an AV, we first consider its overlap with the ratios in the components k to $k-j+1$ in the other AV, which bring to the set from 0 to $l-1$ lines, respectively. The overlappings are performed beginning with the rightmost component and proceeding leftwards in order to contribute ratios to the rightmost possible components in U. That is, we promote overlappings that generate sets with the minimum possible number of lines. In the second stage, what is left from the ratio considered has to be overlapped with ratios from the other AV placed to the left of component $k-j+1$, that is, ratios representing cache sets that hold j or more lines of the region represented by the other AV. These overlappings will necessarily contribute to U_0. In this case we consider the overlapping with ratios from the other AV starting with component 0 and proceeding right till component $k-j$. The rationale is that since this stage cannot avoid contributing to U_0, it does it in such a way that it consumes the rightmost ratios from the other AVs. This reduces sets of lines with many lines to be processed in later steps, and thus the ratio of lines that contribute to U_0.

Example 4.6. *Figure 9 shows a step-by-step example of the union operation of the AVs associated to the regions represented in Fig. 8 that leads to a global AV with the minimum first component, i.e, miss rate generated, which is also shown in Fig. 8. Each step represents the processing of a component of one of the input AVs and its overlapping with one or several components of the other input AV. Each step shows the state of each AV before it is processed and the state afterwards in a smaller italic font. The arrows labeled with the ratio to be overlapped start at the processed element and finish at the element of the resulting AV where the corresponding ratio will be added. The component of the other input AV used in that overlapping is highlighted using a*

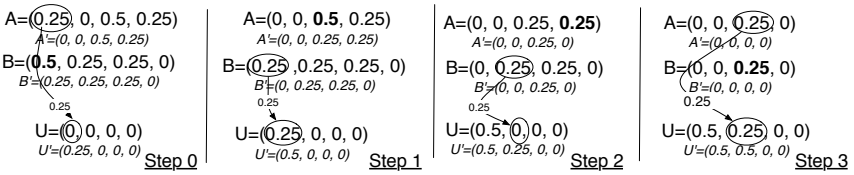

Figure 9: Example of the algorithm union operation considering the best alignment

bold font.

Step 0: *This step shows the processing of* A_0*. The cache sets represented by* A_0 *are already full, thus, this ratio must be overlapped with those of the other AV (B) that contain the maximum number of lines (leftmost components). As this ratio of cache sets is already filled by lines of A, the target is to overlap it with those cache sets of B that contains the highest number of lines. This way, they do not contribute in later overlaps to increase the resulting ratio of full sets. Thereby, the* $A_0 = 0.25$ *is overlapped with a* 0.25 *ratio from* B_0*. The* 0.25 *ratio must be subtracted from* A_0 *and* B_0 *respectively, and contribute to the component 0 of the resulting AV* U_0*, as it stands for cache sets that are full.*

Step 1: *This step shows the processing of the remaining* 0.25 *of* B_0*. Again, the cache sets represented by* B_0 *are already full and they must be overlapped with a* 0.25 *of* A_1 *as it is the non-zero component of A that contains the highest number of lines (leftmost non-zero component), following the same strategy than in step 0. That* 0.25 *ratio is subtracted from* B_0 *and* A_1 *and added to* U_0*.*

Step 2: *The processing of component 1 of A is skipped in this example as it is already 0. This step shows the processing of* B_1*. Since* $B_1 = 0.25$*, that means that* 25% *of the cache sets receive* 2 *lines from* `B`*, so they are not full. Then, it is overlapped with the ratio* 0.25 *of cache sets represented in* A_3*, as they hold 0 lines. This overlapping contributes with a ratio of* 0.25 *to* U_1*. The overlapping with any other ratio from A would contribute at least one extra line to the sets, resulting in full sets that would increase* U_0*, the ratio of full sets.*

Step 3: *The processing of components 2 of both input AVs, represented in this step, is quite simple, as they are the only remaining non-zero values in their respective AVs. They must be overlapped and added to* U_1 *since those sets receive two lines: one from A and another one from B. As a result, the final joint AV is* `U`*=(0.5,0.5,0,0) which could be depicted as the mapping shown in Fig. 8.*

4.4.2. The `minUnionAV` *algorithm*

The strategy just described to calculate the best-case union operation is implemented by the `minUnionAV` algorithm in Fig. 10(a), which combines two input AVs *A* and *B* in order to estimate the best-case resulting global AV *U*. This is the AV with the smallest possible component U_0, the portion of cache sets that have received *k* or more lines. The algorithm is written assuming the indexing of the vectors and arrays begins at 0. The vectors that are input to a routine are subindexed with their size. All

```
1 routine minUnionAV(A[k+1], B[k+1], U[k+1])  {
2   for j = 0 to k − 1 {
3       overlap(A_j, j, B, U, k, k − j + 1, −1)
4       overlap(B_j, j, A, U, k, k − j + 1, −1)
5       overlap(A_j, j, B, U, 0, k − j, 1)
6       overlap(B_j, j, A, U, 0, k − j, 1)
7   }
8   U_k = A_k
9}
```

(a) Main function

```
1 routine overlap(r, j, E[k+1], U[k+1], init, end, s)  {
2   for i = init to end step s {
3       maxovlp = min(r, E_i)
4       E_i = E_i − maxovlp
5       r = r − maxovlp
6       U_max(0,j+i−k) = U_max(0,j+i−k) + maxovlp
7   }
8 }
```

(b) Helper function overlap

Figure 10: Best-case area vector union algorithm

the routine arguments are passed by reference. In our algorithm, each pair of components A_j and B_j of the input AVs are processed in each iteration of the loop in lines $2-7$ in Fig. 10(a), with the exception of the last components A_k and B_k.

The ratio of cache sets represented in A_j must be overlapped preferably with the ratios in those components of B in conjunction with which they do not fill any cache set. These are the components B_k to B_{k-j+1}, in this order. If $j = 0$ no overlapping takes place. This strategy minimizes the final number of lines in the ratio of cache sets targeted by the overlapping. Line 3 of Fig. 10(a) updates the values of A_j, B and U performing that overlapping using the helper function `overlap` shown in Fig. 10(b). Line 4 performs the symmetric operation for B_j. This is the strategy followed in steps 2 and 3 of Example 4.6.

The ratio of cache sets represented in A_j not yet processed at this point, has to be unavoidably overlapped with ratios of components of B in conjunction with which they fill cache sets. These components are those from B_0 to B_{k-j}, in this order. Following this order leaves unprocessed the ratios of sets with the smallest numbers of lines, which have better chances of generating non full sets, when combined with ratios of other AVs. Line 5 of Fig. 10(a) updates the values of A_j, B and U performing this overlapping strategy using the helper function `overlap` shown in Fig. 10(b). Line 6 of Fig. 10(a) performs the symmetric operation for B_j. Steps 0 and 1 of Example 4.6 correspond to this part of the algorithm.

After the components 0 to $k-1$ are processed in the input AVs, only A_k and B_k may be nonzeros, and they will have the same value. This is a ratio of cache sets that are empty, so, it is stored in component U_k in line 9 of Fig. 10(a).

The helper function `overlap`, shown in Fig. 10(b), overlaps a ratio r from component j of one of the input AVs with ratios of the other input AV, which is the parameter E. The components of E are processed one by one from positions *init* to *end* using step s (lines 2-6). Line 3 calculates the ratio of r that can be overlapped with component i of E. Lines 4 and 5 update E_i and r subtracting this ratio. Line 6 updates the position of the resulting AV where the ratio processed in this iteration will be added.

4.4.3. Validity of the union operation

This algorithm ensures the generation of the smallest possible U_0 component by construction. We will reason on how A_j contributes to U_0 in Fig. 10(a). The reasoning

is analogous for B_j. Each ratio r added to U_0 results from overlapping a ratio r from A_j and the same ratio r from B_i, $i \leq k - j$, in line 5. A smaller U_0 could be obtained only if A_j were overlapped with ratios from B_i, $i > k - j$, but that is exactly what line 3 has already done, depleting A_j as much as possible.

The other source of concern is whether B_i, $i > k - j$, have the largest possible values after the processing in the previous iterations of the algorithm, as this minimizes the contribution of A_j to U_0 thanks to the mentioned overlapping in line 3. Each B_i may have lost part of its ratio in each iteration $l < j$ of the main loop either in line 3 or in line 5. In the first case, that ratio was used to avoid A_l contributed to U_0, so saving it for A_j would have made no difference. In the second case, it had to be overlapped with A_l to contribute to U_0 after exhausting the overlap of A_l with B_i, $i > k - l$, in line 3. Line 5 overlaps what is left of A_l with B processing it from left to right. Thus it only resorts to B_i if all the $B_{i'}$, $i' < i$ have already been overlapped and A_l is still nonzero. This way line 5 only overlaps B_i if it is absolutely necessary, and it maximizes the value of the rightmost components of B not overlapped because of the order in which it processes B.

The pseudo-code of the *bmiss_rate* function in Fig. 3 provides an implementation for the last two steps of the best-case miss rate calculation process. Namely, the cache impact estimation and the area vector union steps. The AVs associated associated to each access pattern identified in the previous $interference\_region$ function are combined using the *minUnionAV* function to obtain one final *global_AV*. Component 0 of this global AV is the best-case miss rate that will be used by the PMEs in the innermost level containing the reference. The calculation of the best-case overlapping between reuse and interfering regions described in Section 4.3 is not reflected in this pseudo-code by simplicity.

It is important to clarify that while the union operation guarantees the generation of the smallest possible U_0 component, and thus a safe lower bound for the best-case miss rate, this value may be below the actual value generated by the best-case alignment of the data structures involved in the union operation. The reason is that an AV reflects the distribution of the lines of a given access pattern on the cache sets but it does not encode the information about the relative positions of cache sets, and thus neither does the union algorithm. The best-case union operation overlaps ratios of cache sets of one of the AVs with ratios of cache sets from the other AV, but the combination of ovelappings generated by the union operation may not reflect accurately the actual relative positions of the cache sets.

The example in Figure 11 illustrates this situation. It shows the union of two AVs representing the impact of two regions, RA and RB, on a 3-way associative cache with 4 cache sets. Both regions fill two cache sets with their cache lines, but in one region the full cache sets are contiguous and in the other one they are not. When the best-case union operation is performed, our algorithm overlaps the 0.5 ratio of full cache sets in both AVs and the overall miss rate is 0.5, as half of the cache sets are full. But this situation does not reflect accurately the relative positions observed in the input regions, as in one region both full cache sets are contiguous and in the other region they are not contiguous, so the actual best-case situation is represented in the figure where the actual overall miss rate is 0.75, since 3 out of the 4 sets are full.

Regarding the scalability, a more precise method would require that the informa-

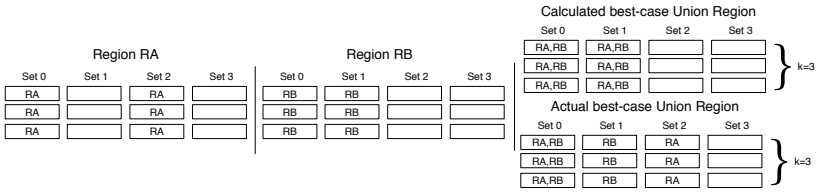

Figure 11: Example of mapping considering the best alignment calculated by the union operation and the actual best alignment

tion about the relative positions of the cache sets would be encoded in each AV, which would generate extra memory requirements. The best-case union of two AVs could not be derived analytically as once a couple of cache sets from two different AVs are overlapped by the union algorithm, this would determine the overlapping of the remaining cache sets represented in those AVs. Thus, the miss rate calculation process should be performed, starting at the best-case overlapping step, taking into account all the possible overlappings among all the memory regions, and then, the overlapping that would derive the lowest miss rate would be selected. The scalability of that method would be very poor as the addition of larger caches and more references to the analyzed code would increase largely the memory requirements of the model and the number of combinations of overlappings to be tested. The precise results reflected in Section 6, obtained in less than one second each, shows that our approach provides a good balance between precision and scalability.

5. BCMP vs WCMP

This section summarizes the differences between the model presented in [3] to calculate the WCMP and the model for the estimation of the BCMP introduced in this paper in order to emphasize the contribution of the current paper. The pseudocode of Figure 3 in Section 2 shows that the PME model processes the references of the analyzed code separately. Algorithm 1 summarizes the algorithm followed to estimate the number of misses generated by a reference R in loop i. The algorithm highlights in a bold font the four parts of the model where the BCMP and WCMP model differ. In line 1 the algorithm computes the number of accesses generated by reference R in loop i associated to each possible reuse distance RD. The BCMP model is designed to ensure that the actual distance for a reuse will be greater than or equal to the one used in the estimation. The WCMP model however is designed to ensure that it is shorter or equal.

Lines 3-9 describe the process to calculate the miss rate associated to a given reuse distance. This process is divided into four steps.

The first step (line 4) identifies the access pattern followed by a given reference and presents no differences in both models.

The second step (line 5) calculates the impact of that access pattern on the cache using a mathematical representation called area vector. The methods used for the WCMP and BCMP estimations are different. For sequential accesses patterns the worst-case

```
1  Compute worst/best-case number of accesses of reference R in loop i associated
     to each possible reuse distance RD ;
2  forall reuse distance RD found do
3  |  forall reference R' in RD do
4  |  |  Indentify access pattern AP of R' in RD;
5  |  |  Compute AV for worst/best-case cache impact of AP;
6  |  |  Adjust AV for worst/best-case overlapping of AP with the reuse region
   |  |    of R;
7  |  end
8  |  Merge all the AVs computed into a single one considering the
   |    worst/best-case situation;
9  |  Take as miss rate for the RD element 0 of the final AV;
10 end
11 Evaluate PME with the miss rate of each RD;
```

Algorithm 1: Worst/best-case construction of the PME for reference R in loop i

approach maximizes the number of lines covered by the pattern, while the BCMP approach minimizes it. In the strided access pattern, the impact on the cache can be represented by one of L_s (L_s being the cache line) area vectors. Each area vector is calculated assuming that the first position accessed by the pattern has a different offset with respect to the beginning of one cache line. The miss rate is calculated assuming each possible AV and the minimum miss rate is selected in the BCMP approach, or the maximum in the WCMP approach.

The area vectors obtained in the previous step represent the impact of a given access pattern on all the cache sets, but the estimation actually depends on its impact on the cache sets where the lines of the reused region are placed. The worst/best-case overlapping step (line 6) turns the area vector obtained in the previous step into an area vector that represents the impact of a given pattern on the cache sets occupied by the reused region. These algorithms are different between the BCMP and the WCMP cases, as the WCMP one promotes that the lines of the pattern are placed in the cache sets occupied by the reused region, while the BCMP promotes that the lines of the pattern are placed in different cache sets from those occupied by the reused region.

Finally, the area vectors union algorithm (line 8) is very different in the WCMP and the BCMP models. Although the targets pursued by both approaches are conceptually complementary the algorithms used to accomplish them are totally different. For example, the BCMP algorithm needs to consider area vectors in groups of two, while the WCMP approach merges all the area vectors at the same time following a completely different strategy thoroughly motivated and described in [13].

Example 5.1. *Example 3.1 derived the PME that predicts the number of misses generated by reference* `b[i][j]` *in the code in Fig. 4,*

$$F_{R0}(\mathrm{RD}_{\mathrm{input}}) = 8 \cdot BMissR(\mathrm{RD}_{\mathrm{input}}) + 24 \cdot BMissR(\mathrm{Iter}_{R0}(1)) + 32 \cdot BMissR(\mathrm{Iter}_{R1}(1))$$

This best-case PME can be compared to the worst-case PME that would be derived following the strategy described in [3],

$$F_{R0}(\mathrm{RD}_{\mathrm{input}}) = 12 \cdot BMissR(\mathrm{RD}_{\mathrm{input}}) + 40 \cdot BMissR(\mathrm{Iter}_{R0}(1)) + 12 \cdot BMissR(\mathrm{Iter}_{R1}(1))$$

This latter result is very different because it ensures that the reuse distances experimented by each access are smaller or equal than those predicted by the model.

Example 4.2 calculated the impact of two memory regions, $\mathrm{Reg_s}(16)$ *and* $\mathrm{Reg_s}(4)$*, on the cache described in Example 3.1. The best-case AV associated to the* $\mathrm{Reg_s}(4)$ *access pattern is* $(0.125, 0.875)$*. If the worst-case approach is followed to obtain this AV, the result is* $(0.1875, 0.825)$*. In the case of the AV associated to the* $\mathrm{Reg_s}(16)$ *access pattern, the best-case AV is* $(0.5, 0.5)$*, while the worst-case one is* $(0.5625, 0.4375)$*. The differences observed are due to the fact that the methods used in the worst-case maximizes the impact of these access pattern on the cache.*

Example 4.4 continued example 4.2 to calculate the best-case overlapping between the interfering regions whose impact on the cache is represented by these AVs and the reused region. This process turned the $(0.125, 0.875)$ *AV associated to the* $\mathrm{Reg_s}(4)$ *access pattern into a* $(0, 1)$ *AV which represented the impact of this region on the cache sets occupied by the reused region. It also turned the* $(0.5, 0.5)$ *AV associated to the* $\mathrm{Reg_s}(16)$ *into* $(0, 1)$*. Following the worst-case approach, however, the* $(0.1875, 0.825)$ *is turned into* $(0.33, 0.67)$ *and* $(0.5625, 0.4375)$ *becomes* $(1, 0)$*. The large difference observed between the results obtained using both strategies, is due to the fact that the worst-case strategy promotes that the cache lines from the interfering region are placed in the same cache sets than the reused region.*

Example 4.6 shows a step by step example where the best-case union of the AVs $(0.25, 0, 0.5, 0.25)$ *and* $(0.5, 0.25, 0, 0.25)$ *is performed. The resulting AV is* $(0.5, 0.5, 0, 0)$*, which implies a best-case miss rate of 50%. If the worst-case approach described in [13] is applied the resulting AV is* $(1, 0, 0, 0)$*, with an associated miss rate of 100%. The difference between the results obtained using both models is because the worst-case strategy ensures that the interfering regions fill the maximum number of cache sets among those where the reused region is placed.*

6. Experimental results

We have validated our model using trace-driven simulations. The model is integrated in a compiler framework and provides its predictions always in less than one second. It was applied automatically to ten codes: the average, sum and difference of the values stored in two arrays (ST); a 1D stencil calculation (STENCIL); the sum of all the values in a matrix (CNT); a matrix transposition (TRANS); the calculation of the first `N` fibonacci numbers (FIBONACCI) and five codes from the DSPStone benchmark suite [14]: convolution, fir, lms, matrix1 (a matrix product) and n_real_updates. Pointer-based memory accesses were replaced with equivalent array accesses and functions were inlined. These codes have been gathered from similar works in the bibliography [3, 15, 7, 16] and they include perfectly and non-perfectly nested loops, uniformly generated references to the same data structure and references following both sequential and strided access patterns.

System	C_s	L_s	k	Hit	Miss
MicroSPARC II-ep	8KB	16B	1	1	10
PowerPC 604e	16KB	32B	4	1	38
MIPS R4000	16KB	16B	1	1	40
IDT79RC64574	32KB	32B	2	1	16

Table 2: Characteristics of the caches used in the experiments: C_s cache size, L_s line size, k associativity, and Hit and Miss are the hit and miss time in cycles, respectively.

The experiments were performed for each code considering a data size of 500 elements per dimension. The complexity of the matrix1 code, and thus its simulation time, is $O(n^3)$, so a smaller number of elements per dimension (200) was used in this case. Each code was tested using the cache configurations present in a MicroSPARC II-ep [17], a PowerPC 604e [18], a MIPS R4000 [19] and a IDT79RC64574 [20], which have been used in [7] too. Table 2 summarizes the main characteristics of these caches, including the cache hit and miss times.

The purpose of our model is the estimation of a base address-independent WCMP and BCMP. Thus its validation is based in simulating each code for each cache configuration using all the possible combinations of relative positions with respect to the cache of its data structures. Our simulation environment facilitates this, as it allows to specify the base addresses of the data structures and it is highly optimized. The variations in the cache behavior in the simulations are due exclusively to the changes in the base addresses of the data structures, since the data-dependent conditionals modeled cannot modify the cache behavior, as Section 2.1 explains. The number of combinations of relative cache offsets of the data structures in a code is very large (for example, in a direct mapped cache of 16 KB, each vector of elements of 4 bytes can present 16 KB/4=4096 different offsets) and it grows exponentially with the number of data structures. Thus two kinds of validations have been performed. For the codes with up to three data structures, *all* the relative address combinations were simulated systematically. For ST and n_real_updates, the only codes including more data structures, simulations using random offset combinations were run for 3200 hours (≈4 and 1/2 months) in a 1.6 GHz Itanium Montvale processor, since the simulations considering all the possible relative address combinations are not feasible. These two codes appear in the validation tables with an asterisk.

Table 3 contains for each code and cache configuration, the average memory performance observed along the simulations expressed in cycles, $\overline{\text{MP}}$, the BCMP estimated by the model presented in this paper, BCMP_{mod}, and the BCMP observed in the simulations, BCMP_{sim}. The memory performance is calculated as $NM \times mt + (ACCS - NM) \times ht$, NM being the number of misses, $ACCS$ the number of accesses, and ht and mt the hit cycles and miss cycles of the studied cache, extracted from Table 2. The BCMP_{mod} value is always smaller than BCMP_{sim}, which supports the validity of the prediction, while the difference between both values is small which shows the tightness of the prediction.

Table 4 shows similar statistics to Table 3 but referred to the WCMP prediction. The $\overline{\text{MP}}$ column is the same, while WCMP_{mod} and WCMP_{sim} reflects the WCMP estimated

Code	MicroSPARC II-ep			PowerPC 604e		
	$\overline{MP}$	$BCMP_{mod}$	$BCMP_{sim}$	$\overline{MP}$	$BCMP_{mod}$	$BCMP_{sim}$
ST_*	8256	8125	8125	14155	14044	14044
STENCIL	4286	4250	4259	6699	6625	6662
CNT	812500	812500	812500	1406250	1406250	1406250
TRANS	2045300	1972625	2036129	6151565	5078824	6145312
FIBONACCI	1625	1625	1625	2794	2794	2794
convolution	3276	3250	3250	5689	5625	5662
fir	4276	4250	4259	6689	6625	6662
lms	8244	8125	8125	14386	14044	14229
matrix1	44290990	42220000	43993684	61416512	61410000	61410000
$n\_real\_updates_*$	6618	6500	6500	11365	11250	11324
Code	MIPS R4000			IDT79RC64574		
	$\overline{MP}$	$BCMP_{mod}$	$BCMP_{sim}$	$\overline{MP}$	$BCMP_{mod}$	$BCMP_{sim}$
ST_*	27159	26875	26875	7225	7180	7180
STENCIL	11867	11750	11789	3905	3875	3890
CNT	2687500	2687500	2687500	718750	718750	718750
TRANS	6292982	5375000	6266111	1762070	1506875	1754705
FIBONACCI	5375	5375	5375	1430	1430	1430
convolution	10836	10750	10750	2901	2875	2890
fir	11836	11750	11789	3901	3875	3890
lms	27280	26875	26875	7318	7180	7210
matrix1	107316817	102820000	106660135	39195460	39190000	39190000
$n\_real\_updates_*$	22045	21500	21500	5799	5750	5780

Table 3: $\overline{MP}$, $BCMP_{mod}$ and $BCMP_{sim}$ for four different cache configurations.

by the model in [3] and observed in the simulations respectively. The $WCMP_{mod}$ value is always greater than $WCMP_{sim}$, which supports the validity of the prediction, while the difference between both values is small which shows the tightness of the prediction. Observe that for the TRANS code the predictions of the WCMP are not very tight in two of the caches. The reason is that the overlapping adjustments applied by the model consider worst-case overlappings that actually do not take place for these specific data sizes and cache configurations.

Our model is the only one to our knowledge that can provide safe and tight estimations when the base addresses of the data structures are unknown. As a result, it is not possible to make a comparison of the estimations of our model with those of previous works in the bibliography. We will review the most related work in Section 7. In this situation, the other models can only predict the WCMP/BCMP for one specific possible combination of the base addresses of the data structures in the cache out of the millions that are possible. The actual memory performance for a given base-address combination can be very far from the one obtained with other combinations. This is indicated by the large difference between memory worst/best-case memory performance in Tables 3 and 4 ($BCMP_{sim}$ and $WCMP_{sim}$) for some codes. Other codes may not have shown a high variability between the results of Tables 3 and 4, but the behavior of some of these codes can vary largely when different problem sizes or cache configurations are used. For example, Table 5 shows the best-case ($BCMP_{sim}$), worst-case ($WCMP_{sim}$) and average-case ($\overline{MP}$) memory performance for the TRANS and matrix1 codes of the

Code	MicroSPARC II-ep			PowerPC 604e		
	$\overline{MP}$	$WCMP_{mod}$	$WCMP_{sim}$	$\overline{MP}$	$WCMP_{mod}$	$WCMP_{sim}$
ST_*	8256	25000	25000	14155	95000	95000
STENCIL	4286	12134	12134	6699	6736	6736
CNT	812500	820285	812500	1406250	1434000	1406250
TRANS	2045300	2057000	2054597	6151565	7242769	6156560
FIBONACCI	1625	1634	1625	2794	2868	2794
convolution	3276	10000	10000	5689	5736	5736
fir	4276	12125	12125	6689	6736	6736
lms	8244	25000	25000	14386	14636	14525
matrix1	44290990	46095742	45142507	61416512	62904800	61417474
$n\_real\_updates_*$	6618	20000	20000	11365	11953	11435
Code	MIPS R4000			IDT79RC64574		
	$\overline{MP}$	$WCMP_{mod}$	$WCMP_{sim}$	$\overline{MP}$	$WCMP_{mod}$	$WCMP_{sim}$
ST_*	27159	100000	100000	7225	40000	40000
STENCIL	11867	45914	45914	3905	3935	3920
CNT	2687500	2714098	2687500	718750	730000	718750
TRANS	6292982	6350000	6347972	1762070	2076305	1768985
FIBONACCI	5375	5414	5375	1430	1460	1430
convolution	10836	40000	40000	2901	2935	2920
fir	11836	45875	45875	3901	3935	3920
lms	27280	100000	100000	7318	40000	40000
matrix1	107316817	114865150	111699325	39195460	41404540	40318000
$n\_real\_updates_*$	22045	80000	80000	5799	32000	32000

Table 4: $\overline{MP}$, $WCMP_{mod}$ and $WCMP_{sim}$ for four different cache configurations. Results already published in [3]

Table 5: $BCMP_{sim}$, $WCMP_{sim}$ and $\overline{MP}$ of the TRANS and the matrix1 benchmarks for 20×20 matrices in the MicroSPARC II-ep cache

Code	$BCMP_{sim}$	$WCMP_{sim}$	$\overline{MP}$
TRANS	2600	3158	2663
matrix1	27100	41554	28162

validation working on 20×20 matrices in the MicroSPARC II-ep cache. The memory performance is 21% worse in the WCMP than in the BCMP for TRANS code and a 53% for matrix1. A single execution of our model, which takes less tan one second, can provide a base-address independent prediction of the WCMP and the BCMP. However, such a prediction could only be provided by the other models by making the individual predictions for all the possible base-address combinations.

7. Related work

The simultaneous calculation of the WCET and the BCET in the presence of data caches has been studied by several authors. Wegener and Mueller [21] presented a code-based static analysis that uses the control-flow information, the calling structure and the cache configuration to produce instruction and data categorizations which describe the cache behavior of each instruction an data reference. The data addresses ac-

cessed are calculated using an address calculator. Petters [22] proposed a measurement-based method that uses information about the memory hierarchy configuration, the used access modes and the access cycles to calculate the BCET/WCET in an Intel P6 by subtracting/adding a safety margin to the measured values.

However, most works have focused only in the calculation of the WCET in the presence of data caches. Several of them have used analytical methods to calculate the WCMP in the presence of caches. The modeling of instruction-caches [23, 24] has had a lot of success, even recently in multicore systems with shared L2 instruction caches [25]. There are also many works devoted to the study of data caches. White et al. [16] bounds, using an static analysis, the worst-case performance of set-associative instruction caches and direct-mapped data-caches. The analysis of data caches needs to determine the base addresses of the involved data structures. Relative address information is used in conjunction with control-flow information by an address calculator to obtain this information. The analysis classifies the accesses in one of four categories: always miss, always hit, first miss and first hit. The validation is performed considering only one cache configuration.

Lundqvist and Stenström [26] distinguish between data structures that exhibit a predictable cache behavior, which is automatically and accurately determined, and those with an unpredictable cache behavior, which bypass the cache. Only memory references, whose address reference can be determined statically, are considered to be predictable. The predictability of a reference is determined considering the storage type (global, stack or heap) and the access type (scalar, regular, irregular or input data dependent). Nevertheless, they do not present an experimental results section.

Ramaprasad and Mueller [15] use the cache miss equations (CMEs) [6], which need the data addresses for their predictions, as a basis for the WCMP estimation. Non-perfectly nested and non-rectangular loops are covered using loop transformations like the forced loop fusion which involves the insertion of loop index-dependent conditionals in the code. Loop index-dependent conditionals are modeled using an extra analysis stage. The validation shows almost perfect predictions of the WCMP but only two (direct-mapped) cache configurations are considered.

Vera et al. [7] use also the data address-dependent cache miss equations (CMEs) to predict the WCMP in a multitasking environment. This work combines the static analysis, provided by the CMEs, with cache partitioning, for eliminating intertask interferences, and cache locking, to make predictable the cache behavior of those pieces of code outside the scope of application of the CMEs. Good predictions of the WCMP are achieved for codes that use the cache locking in order to improve the WCMP predictability.

More recently, Hahn and Grund [27] introduce the concept of symbolic names to predict the cache behavior in the absence of the base addresses information by means of a relational cache analysis. They identify three classes of programs whose worst-case execution time analysis can take advantage of this approach and they successfully validate their technique against the Ferdinand's analysis using one representative benchmark per class for small data and cache sizes.

8. Conclusions

This paper presents a model to predict a tight lower bound of the cache performance whose main novelty is that it is the only one that can be applied without the base addresses of the data structures; only the cache configuration and the source code are needed. This model may be used in conjunction with the WCMP model in [3] to delimit the boundaries of the memory performance in the absence of base address information. This property is very interesting, since base addresses are sometimes unavailable at compile time, and they can even change between different executions. The BCMP predictions have been extensively validated using trace driven simulations for ten codes and four cache configurations that required more than 30000 hours of CPU time showing that this approach yields tight and safe estimations of the BCMP. The experimental results show also the estimations of the WCMP which are validated using trace driven simulations and the same test set.

In the future, we plan to integrate our model in an existing BCET/WCET tool such as [28]. Finally, we will consider the possibility of using as an optional additional input the base addresses of the data structures involved in the code, whenever they are available, in order to provide tighter predictions.

Acknowledgments

This work has been supported by the Galician Government (Xunta de Galicia) under the Consolidation Program of Competitive Reference Groups, cofunded by FEDER funds of the EU (Ref. GRC2013/055) and project INCITE08PXIB105161PR and the Ministry of Education and Science of Spain, FEDER funds of the European Union (Project project TIN2010- 16735). We also want to acknowledge the Centro de Supercomputacion de Galicia (CESGA) for the usage of its computers.

References

[1] R. Wilhelm, et al., The worst-case execution-time problem - overview of methods and survey of tools, ACM Trans. Embedded Comput. Syst. 7 (3).

[2] K. D. Bosschere, et al., High-performance embedded architecture and compilation roadmap, Trans. on HIPEAC 1 (3) (2006) 5–29.

[3] B. B. Fraguela, D. Andrade, R. Doallo, Address-independent estimation of the worst-case memory performance, IEEE Transactions on Industrial Informatics 6 (4) (2010) 664 –677.

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

[5] J. L. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach, 4th Edition, Morgan Kaufmann Publishers, 2006.

[6] S. Ghosh, M. Martonosi, S. Malik, Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior, ACM Trans. on Programming Lang. and Sys. 21 (4) (1999) 703–746.

[7] X. Vera, B. Lisper, J. Xue, Data cache locking for tight timing calculations, ACM Trans. Embedded Comput. Syst. 7 (1).

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

[9] F. Mueller, Compiler support for software-based cache partitioning, in: Proc. of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems, ACM, New York, NY, USA, 1995, pp. 125–133.

[10] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, C. Ferdinand, Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 28 (7) (2009) 966–978.

[11] S. Mohan, F. Mueller, Merging state and preserving timing anomalies in pipelines of high-end processors, in: Proceedings of the 2008 Real-Time Systems Symposium, RTSS '08, IEEE Computer Society, pp. 467–477.

[12] D. Andrade, B. B. Fraguela, R. Doallo, Precise automatable analytical modeling of the cache behavior of codes with indirections, ACM Trans. Archit. Code Optim. 4 (3) (2007) 16.

[13] D. Andrade, B. B. Fraguela, R. Doallo, Static prediction of worst-case data cache performance in the absence of base address information, in: IEEE Real-Time and Embedded Technology and Applications Symposium, 2009, pp. 45–54.

[14] Zivojnović, et al., DSPSTONE: A DSP-oriented benchmarking methodology, in: ICSPAT, 1994.

[15] H. Ramaprasad, F. Mueller, Bounding worst-case data cache behavior by analytically deriving cache reference patterns, in: IEEE Real-Time & Embedded Tech. and Applications Symp., 2005, pp. 148–157.

[16] R. White, C. Healy, D. Whalley, F. Mueller, M. Harmon, Timing analysis for data caches and set-associative caches, in: IEEE Real-Time and Embedded Technology and Applications Symposium, 1997, pp. 192–202.

[17] S. Microelectronics, microSPARC-IIep User's Manual, Tech. rep. (1997).

[18] M. Inc, PowerPC 604e RISC Microprocessor Technical Summary, Tech. rep. (1996).

[19] M. Technologies, MIPS32 4Kp- Embedded, MIPS Processor Core, Tech. rep. (2001).

[20] I. D. Technologies, 79RC64574/RC64575 Data Sheet, Tech. rep. (2001).

[21] J. Wegener, F. Mueller, A comparison of static analysis and evolutionary testing for the verification of timing constraints, Real-Time Systems 21 (3) (2001) 241–268.

[22] S. M. Petters, Bounding the execution time of real-time tasks on modern processors, in: Int. Workshop on Real-Time Comp. and App., 2000, pp. 498–502.

[23] M. Alt, C. Ferdinand, F. Martin, R. Wilhelm, Cache behavior prediction by abstract interpretation, in: SAS '96: Proceedings of the Third International Symposium on Static Analysis, Springer-Verlag, London, UK, 1996, pp. 52–66.

[24] C. Healy, et al., Bounding pipeline and instruction cache performance, IEEE Trans. Computers 48 (1) (1999) 53–70.

[25] J. Yan, W. Zhang, Wcet analysis for multi-core processors with shared l2 instruction caches, in: IEEE Real-Time & Embedded Technology and Applications Symp., Vol. 0, 2008, pp. 80–89.

[26] T. Lundqvist, P. Stenström, A method to improve the estimated worst-case performance of data caching, in: Proc. of the 6th International Workshop on Real-Time Computing and Applications Symposium, 1999, pp. 255–262.

[27] S. Hahn, D. Grund, Relational cache analysis for static timing analysis, in: Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on, 2012, pp. 102–111.

[28] J. Engblom, A. Ermedahl, Modeling complex flows for worst-case execution time analysis, in: IEEE Real-Time Systems Symposium, 2000, pp. 163–174.

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.