Static Prediction of Worst-case Data Cache Performance in the Absence of Base Address Information

Diego Andrade, Basilio B. Fraguela and Ramón Doallo
University of A Coruña, Spain
{*dcanosa,basilio,doallo*}*@udc.es*

Abstract

While caches are essential to reduce execution time and power consumption, they complicate the estimation of the Worst-Case Execution Time (WCET), crucial for many Real-Time Systems (RTS). Most research on static worst-case cache behavior prediction has focused on hard RTS, which need complete information on the access patterns and addresses of the data to guarantee the predicted WCET is a safe upper bound of any execution time. Access patterns are available in those codes that have a steady state of access patterns after the first iteration of a loop (in the following regular codes), however, the addresses of the data are not always known at compile time for many reasons: stack variables, dynamically allocated memory, modules compiled separately, etc. Even when available, their usefulness to predict cache behavior in systems with virtual memory decreases in the presence of physically-indexed caches. In this paper we present a model that predicts a reasonable bound of the worst-case behavior of data caches during the execution of regular codes without information on the base address of the data structures. In 99.7% *of our tests the number of misses performed below the boundary predicted by the model. This turns the model into a valuable tool, particularly for non-RTS and soft RTS, which tolerate a percentage of the runs exceeding their deadlines.*

1. Introduction

Worst-Case Execution Time (WCET) estimation, needed for the design of many Real-Time Systems (RTS), is complex in the presence of caches. Thus the prediction of its Worst-Case Memory Performance (WCMP) component has been an object of research both in the presence of instruction [1], [2] and data [3]–[6] caches, the latter being more challenging, as several references may access the same line, and the data access patterns are more irregular. Hard RTS need these estimations to be completely accurate, i.e., to assure that the prediction is a safe upper bound of any potential execution time. This requires full information on data addresses and worst-case access patterns, being preferable to overestimate the WCMP when a tight accurate value cannot be found. The analysis of regular codes can provide accurate information about worst-case access pattern but this analysis may be unable to determine the exact data addresses accessed as the base addresses of the data structures are often unavailable at compile-time. There are several reasons why these base addresses may be unknown and even change in different runs: stack variables, dynamically allocated memory, modules compiled separately or by just-in-time compilers, etc. Also, physically-indexed caches decouple their behavior from the one predictable by the analyzable virtual addresses.

This paper presents the first model to our knowledge aimed at predicting a WCMP for regular codes without information about the base addresses of the data structures. It is an extension of the Probabilistic Miss Equations (PME) analytical model [7], which provides good estimations of the cache performance. PME predictions are based on average probabilities of the alignment of data with respect to the cache lines and overlapping of the footprints of the data accessed in the cache. We have modified it to consider near worst-case alignments and overlappings, but not pathological worst-case alignments, i.e, those in which references with the same access pattern fall systematically in the same cache sets. Miss rates increase sharply in those situations to values far from those observed normally, and programmers should apply padding or extra buffering to avoid large performance degradations. This strategy allows us to provide tight estimations, which are not absolute maxima in any possible situation, but which reflect realistic WCMP in practice. This way, out of 43200 simulations in our validation, only 0.03% behaved worse than our prediction. Thus our model is particularly valuable for soft RTS [8] and non-RTS designers interested in knowing a probable WCMP of their code when data addresses are unknown at compile time. The rest of this paper is organized as follows. Section 2 introduces the PME model. The modifications we make to calculate the WCMP are described in Section 3. Section 4 contains the validation, Section 5 describes some related work and Section 6 concludes the paper.

2. The PME model

The Probabilistic Miss Equations (PME) model [7] for regular codes predicts the number of misses generated by a code for any cache with a Less Recently Used (LRU) replacement policy. Its inputs are the source code to analyze and the cache configuration. The loops in the codes can be nested in any arbitrary way, and references to memory

. This work was supported by supported by the Xunta de Galicia under project INCITE08PXIB105161PR and the Ministry of Education and Science of Spain, FEDER funds of the European Union (Project TIN2007-67537-C03-02)

can be found in any nesting level. The indexing of the data structures must be done using affine functions of the loop indexes. Also, the number of iterations of each loop must be known to perform the analysis. If it cannot be inferred from the code, it can be obtained from an analysis of the input data that can be provided by the user, obtained by means of runtime profiling or provided using a compiler directive.

Although not covered in our examples, inlining, either symbolic or actual, allows the model to support applications consisting of several routines, as seen in [7]. Applications with irregular access patterns, such as those arising from the usage of pointers or indirections, can be made analyzable for the model by locking the cache before such patterns arise and unlocking it after them. This technique is commonly used to enable cache predictability, particularly for enabling a tight computation of the WCET [6]. Another popular technique equally complementary of this model is software cache partitioning [9], which divides the cache into disjoint partitions, which are assigned to different concurrent tasks. This facilitates the model of multitasking environments, since the model can analyze the behavior of the program executed by each task independently considering only its cache partition.

The PME model studies the behavior of each static reference R in a code separately. Namely, it derives an equation F_{Ri} for each reference R and each nesting level i that contains R that predicts the number of misses generated by R during the execution of that loop. The formulas are called Probabilistic Miss Equations (PMEs) because are based on estimations of probabilities that individual accesses result in misses. We will now explain how PMEs are derived and how the miss probabilities are estimated in turn.

2.1. Probabilistic miss equations

The PME F_{Ri} for static reference R at nesting level i estimates the number of misses generated by R during the execution of the loops as the sum of the number of dynamic accesses generated by R multiplied by their probability of resulting in a miss. To build this formula, the model classifies these accesses according to their reuse distance, which is measured in loop iterations and defines the code executed, and thus the accesses that have taken place, since the last previous access to the line that R is trying to reuse. The reason is that the probability an access results in a miss depends on the footprint on the cache of the data accessed during its reuse distance. For example, a reuse distance for an access of n iterations of loop i means that n iterations of loop i are executed between two consecutive accesses to the line affected by this access, and thus the accesses performed during those iterations must be analyzed to estimate the probability their footprint on the cache has replaced the line, resulting then the second access into a miss. There will be also first-time accesses in the loop, i.e, those that cannot exploit any reuse inside it.

The classification is simplified by the regularity of the access patterns analyzed by the model. First, the regularity guarantees that the access pattern of R during each iteration of a loop is the same, only displaced a constant distance in each access. We call this distance S_{Ri}, the constant stride of reference R with respect to loop i. Thus R exhibits the same number of reuses with the same reuse distances within each execution of the immediately inner loop at level $i+1$ that contains R for each iteration of loop i. Only first-time accesses to lines within loop $i+1$ may have different reuse distances in different iterations of loop i. This property enables expressing F_{Ri} in terms of $F_{R(i+1)}$, the formula for the immediately inner loop containing R. For this reason the model always begins the analysis of each reference in the innermost loop that contains it. $F_{R(i+1)}$ is a function of the reuse distance for the first-time accesses of R to lines during an execution of loop $i+1$. Those reuse distances correspond to iterations of outer or previous loops, and it is F_{Ri} responsibility to provide them. If R belongs to loop i but not to any internal loop, $F_{R(i+1)}$ simply stands for a single access of the reference in one iteration of loop i. Here the recursion of PMEs finishes and the value of $F_{R(i+1)}$ is the probability this individual access results in a miss due to the impact on the cache of the data accessed during the reuse distance.

The second advantage of the regularity is that the iterations of loop i can be classified in two categories: those that lead R to access new sets of lines (SOLs) in the immediately internal loop where it is found (or the reference itself if there are no intermediate loops), and those in which R accesses the same SOLs as in the preceding iteration, thus enabling the potential exploitation of locality. By set of lines (SOL) the PME model refers to the set of lines that R can access during one iteration of a loop. For example, if a $M \times N$ C array (stored by rows) is accessed by column by column , in the analysis of the inner loop each iteration references one line, thus the SOL consists of a single line. In the analysis of the outer loop that controls the column index of the reference, each iteration of this loop is associated to the access to the set of lines that hold the elements of a column of the matrix. Considering that the array is stored in row-major order, if $N \geq L_{\mathrm{s}}$, where L_{s} is the cache line size measured in elements, which is the most usual situation, each SOL will be made up of M different lines.

Concretely, the average number of iterations of loop i that cannot exploit either spatial or temporal locality with respect to previous iterations of that loop is

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

L_{s} being the line size, N_i the number of iterations of the loop, and S_{Ri} the stride of R with respect to loop i. When loop i index variable does not index reference R, $S_{Ri} = 0$ and $L_{Ri} = 1$, since the iterations of the loop do not lead the reference to access different data sets. In any other case, (1) is equivalent to calculating the average number of different lines accessed during N_i iterations with step S_{Ri}, each line defining a SOL.

The preceding reasoning establishes that there are $N_i - L_{Ri}$ iterations of the loop that reuse the SOL accessed in the previous iteration. As a result, in these iterations the reuse

```
for(j=0; j<M; j++)      // Level 0
  for(i=0; i<N; i++)    // Level 1
    a[j] = a[j] + b[j][i] * c[i]
```

Figure 1: Matrix-Vector product

distance for each first-time access to the SOL in the internal loop $i+1$ is one iteration of the loop at level i. But for the other L_{Ri} iterations to new SOLs nothing can be said at this point about their reuse distance. It may correspond to iterations in outer loops not yet analyzed, or to loops that precede the current one in the nesting level. For this reason, F_{Ri} is a function of the unknown RegIn, the memory regions accessed during the reuse distance for what in this loop are first accesses. Thus the number of misses generated by R at nesting level i is estimated by the PME [7]:

$$F_{Ri}(\mathrm{RegIn}) = L_{Ri} \cdot F_{R(i+1)}(\mathrm{RegIn}) + (N_i - L_{Ri}) \cdot F_{R(i+1)}(\mathrm{Reg}_{Ri}(1)) \quad (2)$$

where N_i is again the number of iterations of the loop at nesting level i, and L_{Ri} is derived from (1). $\mathrm{Reg}_{Ri}(j)$ stands for the set of memory regions accessed during j iterations of the loop in nesting level i that can interfere with the accesses of R in the cache. This equation calculates the number of misses as the sum of two values. The first one estimates the number of misses of the first access in loop i to the L_{Ri} different SOLs R accesses in the scope of this loop. In this first access in this loop the reuse can only happen with respect to outer or preceding loops. Thus the number of misses generated in these iterations is obtained evaluating $F_{R(i+1)}$, the PME for the immediately inner loop, passing as parameter for the calculation of the miss probability of its first accesses the value RegIn provided by those external loops.

The second value corresponds to the $N_i - L_{Ri}$ iterations in which there can be reuse with respect to the accesses in the previous iteration in this loop. The miss probability for the first accesses in the evaluation of $F_{R(i+1)}$ for these iterations depends on the memory regions accessed during one iteration of loop i, the reuse distance.

At the end of the recursion, in the innermost loop containing the reference, $F_{R(i+1)}(Reg)$ is substituted by $miss_prob(\mathrm{Reg})$ the miss probability associated to region Reg. Its calculation is covered in Section 2.2. In the topmost nesting level in the code the PMEs are evaluated with an RegIn whose miss probability is 1, which expresses the impossibility of reuse of any previous access.

Equation (2) captures the behavior of references that do not carry reuse with any other reference in the same loop nest. The modeling of reuses between different references in this model is explained in [7].

Example 2.1: The matrix-vector product code shown in Fig. 1 will be used as a running example throughout the paper. Let us consider the analysis of the behavior of reference `b[j][i]`. Let us assume that matrix `b` is stored in row-major order, $M = N = 4$ and the system has a direct-mapped cache that can store 8 elements of the matrix and each cache line can store 2 elements. First, we derive a formula for the innermost loop for that reference F_{R1}. Since $N_1 = 4$ and $S_{R1} = 1$, (1) yields $L_{R1} = 2$. The resulting formula is,

$$F_{R1}(\mathrm{RegIn}) = 2 \cdot F_{R2}(\mathrm{RegIn}) + (4-2) \cdot F_{R2}(\mathrm{Reg}_{R1}(1))$$

that is, in two of the iterations of the innermost loop the reference accesses new lines, since 4 elements distributed on lines of two elements require two lines, while the other two iterations reuse the line accessed in the inmediately preceding iteration. The reuse distance for the first accesses to lines is (yet) unknown, but the distance for the reuses is one iteration of this loop. Let us now derive the PME F_{R0} for the outermost loop. Since here $N_0 = 4$ and $S_{R0} = 4$, then $L_{R0} = 4$ and

$$F_{R0}(\mathrm{RegIn}) = 4 \cdot F_{R1}(\mathrm{RegIn}) + (4-4) \cdot F_{R1}(\mathrm{Reg}_{R0}(1))$$

that is, the outermost loop generates accesses to 4 different set of lines (SOLs). When the formulas are composed, the final number of misses for reference `b[j][i]` is calculated as

$$F_{R0}(\mathrm{RegIn}) = 8 \cdot miss_prob(\mathrm{RegIn}) + 8 \cdot miss_prob(\mathrm{Reg}_{R1}(1))$$

If this is the first access to `b` in the program, the first accesses to lines in this code cannot exploit reuses thanks to preceding accesses; thus $miss_prob(\mathrm{RegIn}) = 1.0$. The other 8 accesses try to reuse the line accessed in the previous iteration of loop 1, thus their miss probability depends on $\mathrm{Reg}_{R1}(1)$, the memory regions accessed during such iteration. next section ■

2.2. Miss probability calculation

The PME model follows three steps to calculate the miss probability associated to a given reuse distance: access pattern identification, cache impact estimation and area vectors union.

The first step identifies what kind of access pattern follows each reference during the reuse distance. When there are several references that access the same memory regions, they must be merged. The main types of access pattern found in regular codes are the sequential access and the access to regions of the same size separated by a constant stride.

The second step measures the impact of each access pattern on the cache using a vector V of $k+1$ probabilities called area vector (AV), k being the associativity of the cache. Its first element, V_0, is the ratio of sets that received k or more lines from the access, while $V_s, 0 < s \leq k$ is the ratio of sets that received $k-s$ lines. The AV associated with each access pattern is calculated separately. The method to derive the AV depends on the access pattern considered. For example, the sequential access to n consecutive elements, $\mathrm{Reg_s}(n)$, generates an interference area vector AV_s:

$$\begin{aligned} &AV_{\mathrm{s}_{(k-\lfloor l \rfloor)}}(n) = 1 - (l - \lfloor l \rfloor) \\ &AV_{\mathrm{s}_{(k-\lfloor l \rfloor - 1)}}(n) = l - \lfloor l \rfloor \\ &AV_{\mathrm{s}_i}(n) = 0 \qquad 0 \leq i < k - \lfloor l \rfloor - 1, k - \lfloor l \rfloor < i \leq k \end{aligned} \quad (3)$$

where, k is the associativity of the cache and l is the average number of lines the access places in each cache

set, calculated as $l = \max\{k, ((n + L_\mathrm{s} - 1)/L_\mathrm{s})/S\}$, the maximum of k and the average number of lines mapped to each set. This value is obtained dividing the average number of lines affected by the access by S, the number of cache sets, which is calculated as $C_\mathrm{s}/(L_\mathrm{s}k)$, where C_s is the cache size. The term $L_\mathrm{s} - 1$ added to n stands for the average extra elements brought to the cache in the first and last lines accessed. Details of the methods to derive the AVs corresponding to other access patterns can be found in [7].

The previous step of the calculation of the miss probability associated to a reuse distance yields one AV per each one of the memory regions accessed during that distance. The last step of the calculation of the miss probability summarizes the effects of these AVs merging them into a global one through an union operation [7] based on the addition of independent probabilities. Namely, the union ($\cup$) of two AVs V_A and V_B is calculated by:

$$
\begin{aligned}
(V_\mathrm{A} \cup V_\mathrm{B})_0 &= \sum_{j=0}^{k} \left(V_{\mathrm{A}_j} \sum_{i=0}^{k-j} V_{\mathrm{B}_i} \right) \\
(V_\mathrm{A} \cup V_\mathrm{B})_i &= \sum_{j=i}^{k} V_{\mathrm{A}_j} V_{\mathrm{B}_{(k+i-j)}} \qquad 0 < i \leq k
\end{aligned} \tag{4}
$$

The first component of the resulting AV, which is the ratio of cache sets that received k or more lines during the reuse distance, is conversely the probability a randomly chosen set received k or more lines during the reuse distance. Since a line is displaced from a LRU cache when k or more lines fall in its set during its reuse distance, this is the probability the reuse affected by that interference region results in a miss.

Example 2.2: Let us now complete the analysis of the behavior of reference `b[j][i]` in the code of Fig. 1 calculating $miss_prob(\mathrm{Reg}_{R1}(1))$, which was pending in Example 2.1. In one iteration of the innermost loop there is an access to one element of vector `a`, one element of vector `c` and one element of matrix `b`. As we are calculating the interference between the elements of `b` accessed in two consecutive iterations of the innermost loop the element of `b` accessed can not interfere with itself. However, the accesses to `a` and `c` can, so they are both identified as two sequential accesses to 1 element. Applying (3) the area vectors that represents the impact of the access to `a` and `c` on the considered cache ($C_\mathrm{s} = 8$,$L_\mathrm{s} = 2$,$k = 1$) are both $(0.25, 0.75)$. That is, 1 of 4 cache sets receive $k = 1$ or more lines from that data structure and the 3 out of 4 remaining cache sets receive 0 lines. The impact of both accesses is then summarized using the union operation of (4) on the corresponding area vectors, yielding the global area vector $(0.4375, 0.5625)$. Thus the miss probability associated to $\mathrm{Reg}_{R0}(1)$ is 0.4375. Equation (4) computes the output AV based on the independence of the probabilities that a cache set receives the line from `a` or `c`: since each one of these probabilities is 0.25, the probability any of them holds for a randomly chosen cache set is $0.25 + 0.25 - 0.25 \cdot 0.25 = 0.4375$. Also, the probability none of them hold, and thus the set is empty if $(1 - 0.25) \cdot (1 - 0.25) = 0.5625$ ■

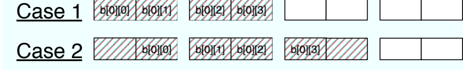

Figure 2: Lines accessed depending on the base address of matrix `b`

3. WCMP estimation

The PME model as described in Section 2 and [7] estimates the average number of misses generated by a code without any information on the addresses of the data accessed. This way, it uses the average number of SOLs accessed in its formulas, but not the worst case, which depends on the alignment with cache lines of the data structures. Similarly, it calculates the area vector representing the average impact, but not the worst one, on the cache of the access patterns. Besides, its union of area vectors is also based in independent average probabilities of overlap, rather than nearly worst-case situations. We will consider these aspects in turn to extend the PME model with the ability to derive a WCMP estimation as it will be validated in Section 4. Finally, we will discuss the theoretical accuracy of our prediction.

3.1. Worst case number of different SOLs

Equation (1) calculates the number L_{Ri} of different SOLs accessed by reference R in nesting level i. It assumes that the address referenced by the first access in the loop is aligned on cache line boundaries. When this is not the case and $0 < S_{Ri} < L_\mathrm{s}$, this formula underestimates the worst L_{Ri} by one unit. The worst-case L_{Ri} is given by

$$
L_{Ri} = \left\lceil \frac{S_{Ri}(N_i - 1) + L_\mathrm{s}}{L_\mathrm{s}} \right\rceil . \tag{5}
$$

which considers that the first element of the data structure is placed at the end of a cache line. This maximizes the number of lines affected.

Example 3.1: The analysis of reference `b[j][i]` in loop 1 in Fig. 1 in Example 2.1, yielded a PME F_{R1} where the number of different lines accessed, calculated using (1) for $N_1 = 4$ and $S_{R1} = 1$, was 2. Fig. 2 shows the mapping of a row of `b` on the considered cache ($C_\mathrm{s} = 8$, $L_\mathrm{s} = 2$, $k = 1$) considering different base addresses. The lines accessed during the first execution of the innermost loop are marked. Case 1 matches the prediction of (1) as 2 different lines are accessed. However, in case 2 the first row of matrix `b` is spread on 3 lines, which is the worst case as (5) computes ■

3.2. Worst case cache impact estimation

In Section 2.2 the area vector (AV) representing the impact on the cache of a sequential access to n elements was calculated using (3), where $l = \max\{k, ((n + L_\mathrm{s} - 1)/L_\mathrm{s})/S\}$ was the maximum of k and the average number of lines placed in each set. The term $L_\mathrm{s} - 1$ added to n stood for the average extra elements brought to the cache in the first and

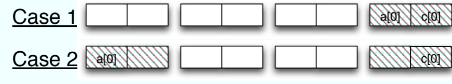

Figure 3: Impact of region $\text{Reg}_{R1}(1)$ on the cache depending on the base address of a and c

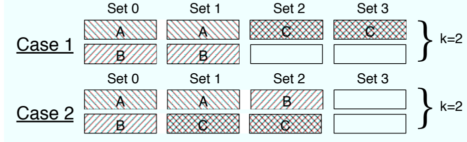

Figure 4: Considering all the AVs at a time during the union operation

last lines accessed. Nevertheless in the worst case the first element of the access is at the end of a line, and the other $n-1$ elements require bringing $\lceil(n-1)/L_s\rceil L_s$ elements to the cache to access them. Thus in the worst case the number of elements brought to the cache is $L_s + \lceil(n-1)/L_s\rceil L_s$, instead of $n+L_s-1$. Consequently we make this replacement in the computation of l.

Example 3.2: Reading a row of b, represented in Fig. 2, involves accessing $n = 4$ consecutive elements, which according to (3) yields the AV $(0.625, 0.375)$. This calculation is an average that overestimates the AV for Case 1, a best case alignment where the area vector would be $(0.5, 0.5)$, and underestimates the AV for the worst case alignment in Case 2, where the area vector is $(0.75, 0.25)$. This worst case AV is accurately computed with our modification ■

3.3. Worst case area vectors union

Equation (4) yields an average estimation of the composed impact on the cache of the input AVs, but the actual AV could have a much larger first component depending on the actual overlap of the footprints in the cache.

Example 3.3: In Example 2.2, we merged the AVs of the memory regions that may interfere with the attempt by reference `b[j][i]` to reuse the line it accesses in two consecutive iterations of the innermost loop in Fig. 1. These regions were an element of a and another one of c. The value of each one of these AVs was $(0.25, 0.75)$, since our example cache has 4 sets and the access to one element brings a single line to the cache, resulting in 25% of the cache sets receiving one line and the other 75% receiving no lines. We also explained how (4) computed the average global area vector as $(0.4375, 0.5625)$. This means that on average 43.75% of the cache sets receive lines that conflict with the reuses of `b[j][i]`, the other 56.25% remaining untouched. Fig. 3 shows two possible distributions of the lines of a and c accessed between the first two iteration of the innermost loop on the example cache, considering two different base addresses combinations. In Case 1 both lines are mapped to the same cache set, so the AV of their joint impact on the cache is $(0.25, 0.75)$, as only one of 4 sets receive lines. The figure shows both elements simultaneously in the cache to simplify the representation, but actually one of the lines would replace the other one. However, Case 2 shows the worst case for the global impact in the cache, in which the lines are mapped to different sets, resulting in an AV $(0.5, 0.5)$ ■

The exact union of the area vectors corresponding to the footprint of different regions can only be computed knowing the addresses of the data structures, but they are often not available. Besides, they can change between different executions. We have developed a `maxUnionAV` algorithm that combines the components of its input AVs in order to estimate the worst-case resulting global AV. This will be the AV with the largest possible leftmost component, the portion of cache sets that have received k or more lines, as it corresponds to the miss probability generated by the interference of the regions summarized in the vector.

A first point of interest is that while the average union of several AVs can be computed processing them two by two applying (4), the calculation of the worst-case union requires considering all the AVs simultaneously.

Example 3.4: Let us consider a 2-way cache ($k = 2$) with four sets, and a reuse distance during which the interference may come from three access patterns called A, B and C, in which each one of them accesses two lines that are mapped to different sets. Thus the three input AVs have the value $(0, 0.5, 0.5)$, meaning that two of the four sets receive one line, the other two ones remaining empty. The worst-case union of these AVs is the one that maps the lines represented by these AVs to the 2-way cache sets so that the highest possible number of sets are full, i.e., receive two or more lines.This is equivalent to finding the global AV with the highest possible value of its first component. Fig. 4 depicts two different mappings or distributions of the lines associated to the input AVs on the example cache. Case 1 shows the mapping that a worst-case union operation would obtain considering the AVs two by two: the way to fill as many sets as possible only with lines from A and B is to map the two lines of their access patterns to the same sets, so that two sets are full. Then, when C were considered, it would not make sense to map its two lines to the same sets, as they are already full. Rather they would be mapped to the other two sets. Since this mapping fills two sets and puts one line in the other two, its corresponding global AV is $(0.5, 0.5, 0)$, whose associated miss probability is 0.5. However, as Case 2 shows, the lines of the three access patterns or regions can be combined to fill 3 sets, yielding a global AV $(0.75, 0, 0.25)$ with a miss probability of 0.75. Notice that both mappigs fulfill the restriction that the AV for each one of the input access patterns is $(0, 0.5, 0.5)$: for A, B and C taken individually, no set receives two or more lines from the same region, half of the sets receive one line, and the other half are not affected by that access ■

Before describing formally our algorithm for worst-case area vector union calculation, let us discuss its basics using Fig. 5. The algorithm works on a matrix in which each row is the AV of one of the memory regions accessed during the considered reuse distance. Fig. 5 shows in Examples (a), (b) and (c) the union process of three different matrices of AVs.

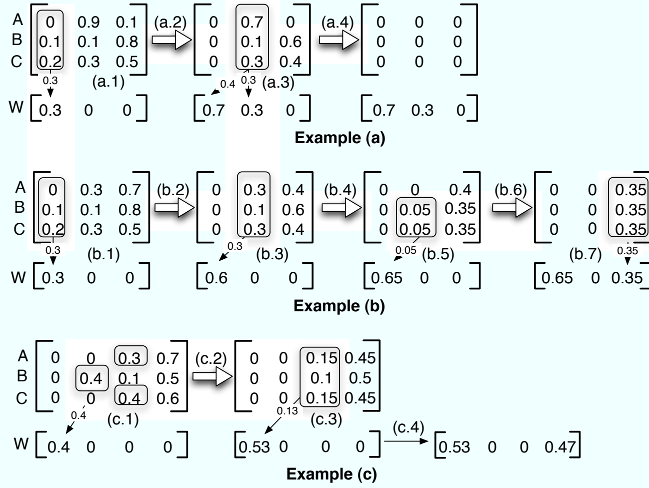

Figure 5: Examples of worst-case area vector union.

Each matrix in the figure has three AVs called A, B and C, whose worst possible joint AV W is to be computed. This AV corresponds to the placement on the cache of the lines represented in the input AVs that maximizes the ratio W_0 of cache sets that receive at least k lines, k being the associativity of the cache.

The first step adds the ratios of sets that are filled individually by each one of the input AVs, which are the first column of the matrix. This policy assumes that two AVs never fill the same set, thus maximizing the number, and thus the ratio, of sets they fill when considered together. In Fig. 5-(a) and (b), steps (a.1) and (b.1) yield an initial $W_0 = 0.3$. Example (c) obviates this step because the first column is empty.

Each AV represents the impact on the cache of a memory region by providing the ratio of sets that receive $\geq k$ (component 0), $k-1$ (component 1), and down to 0 lines (component k) from the region, thus its components cover the whole cache and their addition always yields 1. When ratios of sets are assigned to positions in W, all the AVs in the input matrix must be updated accordingly, subtracting them from their components, as those ratios are fractions of the cache that have been already processed by the algorithm. Let us see it in detail in step (a.2) in Fig. 5-(a). We just assigned 0.3 (or 30%) of the cache sets to W_0 meaning that they are filled, 0.1 from B_0 and the other 0.2 from C_0. This 0.3 is thus a portion of the cache already processed and must then be subtracted from each one of the rows in the matrix. AV A did not participate in building W_0 because $A_0 = 0$. Thus the ratio 0.3 of sets filled by B and C must correspond to/overlap with ratios of sets in A that received 1 ($A_1 = 0.9$) or 0 lines ($A_2 = 0.1$). Our algorithm overlaps/subtracts the ratios in the updates of the matrix starting at the rightmost column in each row and proceeding leftward. This maximizes the ratios of sets left with the highest number of lines, which can then be later combined with lines from other AVs to fill sets. In this example, AV A becomes $(0, 0.7, 0)$ when we subtract 0.3 starting from the right. Now, in AV B, 0.1 of the ratio 0.3 to subtract come actually from B_0, so we first zero B_0 and then subtract the remaining 0.2 starting from the right. In AV C, C_0 is zeroed and the remaining 0.1 is subtracted from the right. Step (b.2) performs the same processing in the example of Fig. 5-(b).

The next step is to combine sets with $k-1$ lines (ratios in column 1) from an AV with sets with at least one line from any other AV. Our algorithm explores in each turn the overlapping of component 1 of each row of the matrix with all the other rows. In Fig. 5-(a) step (a.3), 0.7 of the cache sets have exactly one line from A, 0.1 one from B, and 0.3 one from C. The worst-case combination achievable assigns the ratios from B_1 and C_1 to disjoint sets, which added can overlap with 0.4 of the sets from A_1. This fills those sets with two lines, which are then added to W_0. The remaining $0.7 - 0.4 = 0.3$ ratio of sets with one line from A cannot receive lines from other AVs, so they contribute to W_1. Fig. 5-(b) step (b.3) exemplifies this process with a more challenging situation where $A_1 = 0.3$, $B_1 = 0.1$ and $C_1 = 0.3$. The worst combination must overlap A_1 with portions of B_1 and C_1 so that what is left of them after the overlap with A_1 has the largest possible value for both of them. This will maximize their contribution to W_0 when we consider overlapping B with C. This is achieved in practice by producing the B_1 and C_1 with the most similar (ideally equal) value after the update. In this case this corresponds to taking 0.05 from B_1 and 0.25 from C_1 to overlap with A_1. This leaves $B_1 = C_1 = 0.05$, to be combined in step (b.5). Before that, step (b.4) subtracts from each AV the 0.3 processed in step (b.3). Just as in step (b.2), the portions used in each row to generate overlappings that contribute to W are subtracted from their column (0.3, 0.05 and 0.25 from A_1, B_1 and C_1, respectively). The remaining ratio till reaching 0.3 in each row is subtracted from the rightmost column. The first two steps in Fig. 5-(c), (c.1) and (c.2), exemplify this process for a 3-way cache (4 columns) where a ratio 0.4 of sets with 2 lines from B overlap with 0.15 with one from A and 0.25 with one from C. This leaves $A_1 = C_1 = 0.15$ after the update.

Once column 1 in the matrix is processed, the accurate study of the worst combinations of the remaining columns is extremely complex. For example a set that needs 3 lines to be full could receive them from up to three different AVs. Our algorithm calculates which is the leftmost component W_q of W that could be updated with what is left in the matrix by determining which is the leftmost nonzero in each row in the matrix, since this indicates the maximum number of lines the associated memory region can contribute to a cache set. Then the number of lines represented by the AVs in the matrix is calculated multiplying each ratio in column j by $k-j$, the number of lines per set associated to that column. Dividing this value by the number of lines $k-q$ associated to W_q gives an upper bound of their contribution to W_q. This is done in step (c.3) in Fig. 5-(c). Finally, W_k may have to be updated so that the addition of all the ratios in W is 1, as in step (c.4).

```
1function maxUnionAV(VM_[NV×(k+1)])  {
2    W_0 = ∑ VM_{*0}
3    W_j = 0,  0 < j ≤ k
4    if k = 1 {
5        W_0 = min(W_0, 1)
6    } else {
7        ovlRight2Left(VM_{i*}, W_0 − VM_{i0}),  0 ≤ i < NV
8        VM_{*0} = 0
9        for i = 0 to NV − 1 {
10           ovlComb(W, i, VM)
11        }
12        W_j = min(W_j, 1),  0 ≤ j ≤ 1
13        q = max(0, k − ∑_{i=0}^{NV−1} (k − min(j | VM_{ij} ≠ 0)))
14        W_q = W_q + min(1 − (W_0 + W_1), ∑_{j=2}^{k} VM_{*j}(k − j)/(k − q))
15    }
16    W_k = 1 − ∑_{j=0}^{k−1} W_j
17    return W
18}
```

Figure 6: Worst-case area vector union algorithm

```
1procedure ovlRight2Left(v_[l], t)  {
2    r_j = max(v_j − max((t − ∑_{z=j+1}^{l−1} v_z), 0), 0),  0 ≤ j < l
3    v = r
4}
```

Figure 7: Helper routine that zeroes the rightmost (higher index) positions of a vector v that add up t

Formal algorithm. Fig. 6 shows the pseudocode of the worst-case area vector union algorithm. Array indexing is 0-based. Notation M_{*j} means column j of matrix M; similarly M_{i*} means row i of matrix M. The vectors and arrays that are input to a function are subindexed with their size and are passed by reference. The input to the algorithm is a matrix `VM` in which row i is the i-th of the NV AVs to combine. Its output is the worst-case resulting AV W.

First the function adds component 0 of the input AVs into W_0 in line 2. As we explained in our comments on Fig. 5, this is the worst miss ratio that the sets with k or more lines from the input AVs can give place to combined. The rest of W is then initialized with zeros.

If $k = 1$, we round down in line 5 W_0 to 1 in case the computation in line 2 yielded a higher value. Then W_1 can be calculated as $1 - W_0$ in line 19.

If $k > 1$, W_0 can still grow combining for example sets that receive $k-1$ lines from one AV, i.e, from column VM_{*1}, with sets that receive 1 or more lines from a different AV. This is examined in lines 9-15. Before that, the matrix must be updated subtracting in each row or AV i the portion of W_0 that comes from the other rows ($W_0 - VM_{i0}$) to account for the fact that portion of the cache has already been processed. As we reasoned when we walked through the examples in Fig. 5, the update overlaps those sets already full from the other AVs with the sets represented in the rigthmost part of the considered AV i. This strategy leaves unprocessed the ratios of sets with the largest numbers of lines, which have better chances of generating full sets, and thus increase the miss probability, when combined with ratios of other AVs. This process takes place in line 7 using the helper function `ovlRight2Left` shown in Fig. 7. Line 8 zeroes the already processed column.

Loop 9-11 in Fig. 6 considers in its turn the overlapping of VM_{i1} for each AV in VM with all the other AVs in the matrix using routine `ovlComb`. The routine, shown in Fig. 8, first makes a temporary working copy t of VM_{i1}, and M and M' of all the rows of VM but i. These are the rows that can provide ratios of sets with lines which overlapped with VM_{i1}, which holds $k-1$ lines, will give place to full sets and thus miss probability. Loop 4-14 analyzes the overlap of what is left of VM_{i1} (t) with the components in column j of all the other rows. The loop begins in column $k-1$ since the sets in column k have 0 lines and cannot thus contribute any line. It stops in column 1 because column 0 has already been fully processed. The descending order of the loop tries to use sets with the minimum number of lines needed to fill the sets represented by VM_{i1}, and maximize the ratio of sets with as much as lines as possible available after this processing to generate miss ratio. When the addition of all the ratios in a column is smaller than t, t is updated subtracting their value to indicate that that portion of cache has already been filled and processed, and the column is zeroed. Otherwise the loop finishes zeroing t and subtracting from the current column M_{*j} an amount of ratio equal to t. We have found that the distribution of this ratio among the rows in this column that maximizes the miss probability is the one in which the resulting M_{*j} has the minimum maximum value. This achieves the best possible balance among the components of the vector, which maximizes the potential overlap among them. This computation is represented first defining in line 9 the set V of the vectors of $NV-1$ elements such that each element v_i is $0 \leq v_i \leq M_{ij}$ and whose addition equals t. Then column j is updated subtracting from it the vector from V such that the output column has the smallest maximum value, and the loop is broken.

After the loop, what is left yet in t cannot be overlapped with lines from other AVs and thus contributes to W_1. The other $VM_{i1} - t$ has been overlapped successfully and contributes to W_0. Line 17 adjusts M making the right to left overlap in each row i for the fraction of VM_{i1} that was overlapped with components from area vectors from other

```
1procedure ovlComb(W, i, VM_[NV×(k+1)])  {
2    t = VM_{i1}
3    M' = M = VM_{(0..i−1,i+1..NV−1)*}
4    for j = k − 1 to 1 step −1 {
5       if ∑ M_{*j} < t {
6         t = t − ∑ M_{*j}
7         M_{*j} = 0
8       } else {
9         let V = { v ∈ ℝ^{NV−1} | (0 ≤ v_i ≤ M_{ij}, 0 ≤ i ≤ (NV − 2)) ∧
                   (∑_{i=0}^{NV−2} v_i = t) }
10         t = 0
11         M_{*j} = M_{*j} − v, v ∈ V ∧ ∄v' ∈ V | max(M_{*j} − v') < max(M_{*j} − v)
12         break
13      }
14   }
15   W_0 = W_0 + (VM_{i1} − t)
16   W_1 = W_1 + t
17   ovlRight2Left(M_{i*}, VM_{i1} − ∑(M'_{i*} − M_{i*})),  0 ≤ i ≤ (NV − 2)
18   VM_{i1} = 0
19   VM_{(0..i−1,i+1..NV−1)*} = M
20}
```

Figure 8: Optimal worst-case overlapping of a ratio VM_{i1} with the ratios of other $NV-1$ area vectors stored in matrix VM

rows in the main loop. Finally VM_{i1} is zeroed and VM is updated from M.

Back in Fig. 6, the fraction of sets calculated for W_0 and W_1 are prevented from being greater than 1 in line 12. At this point VM can contain components that can be overlapped in columns $j > 2$. There is an explosion of combinations of components of different AVs to analyze when more than one extra line is needed to fill a set. Thus we take the conservative approach we detailed in our explanation of Fig. 5-(c) : line 13 calculates which is the leftmost component W_q of W to which the overlap of components from different rows of VM can contribute and line 14 updates it with a worst-case combination of those ratios.

Finally, although all the sets with at least one line have already been combined, it is possible that empty sets are available to contibute to W_k, thus it is adjusted in line 16 so that the addition of all the elements in W is 1.

3.4. Model accuracy

A totally safe WCMP prediction in the absence of data address information requires taking a totally pessimistic approach. For example, when several references exhibit the same access pattern in a loop, their accesses fall systematically in the same cache sets if their base addresses are aligned with respect to the cache. If the number of conflicting references is $> k$ (the associativity of the cache), the miss probability for the attempts to reuse their lines, even in the innermost loop, is necessarily 1. We can quantify this error statistically. The probability of underpredicting the WCMP in a code where n data structures have the same access pattern in the innermost loop is equal to the probability that more than k of them are aligned with respect to the cache. The probability an address belongs to a particular cache set is $1/S$, S being the number of cache sets. So the probability that a base address combination fulfills this condition is $S \cdot P(x > k)$, where x is the number of data structures aligned in the same cache set, which belongs to a binomial of n elements with probability $1/S$ each.

4. Validation results

We have validated the WCMP predictions of our model comparing them with the results of trace-driven simulations. The programs used were: the product of two dense matrices (DMXDM); 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), and the calculation of the first `N` fibonacci numbers (FIBONACCI). These programs are used in similar works in the bibliography [3], [6].

Tests were performed for each program considering different data sizes and cache configurations. The cache configurations were the 80 possible combinations of the cache sizes $16KB-32KB-64KB-128KB-256KB$, the line sizes $16B-32B-64B-128B$ and the associativities $1-2-4-8$. The data structure sizes were all the combinations considering in each dimension sizes of $500-1000-1500$. For each combination of data sizes and cache configuration, 10 simulations were performed using different random base addresses for each data structure. The maximum miss rate obtained was compared with the one predicted by the model, which was applied automatically. The model prediction was typically obtained in less than one second.

Table 1 summarizes the results obtained. The first column identifies the code; the second column is the number of configurations tested for each code. The third column contains the statistics about the WCMP prediction. ErrorConf$\%$ is the percentage of configurations where the model mispredicted the WCMP, as the miss rate of at least one of their ten simulations was larger than the predicted one. ErrorSim$\%$ is the percentage of individual simulations that behaved worse than the prediction. The remaining statistics focus on the results obtained in the configurations where the worst-case prediction was successful. Avg($\Delta MR\%$) is the average difference between the worst-case miss rate predicted and the one simulated. The miss rate is always expressed as a percentage. Avg($MR\%$) contains the average worst-case miss rate obtained by the simulator. Min($\Delta MR\%$) and Max($\Delta MR\%$) reflect the minimum and maximum value respectively of the discrepancy between the worst-case miss rate predicted and the one simulated. Finally, ($\Delta MR > 5\%$) and ($\Delta MR > 10\%$) are the percentage of configurations where this discrepancy was larger than 5% and 10%, respectively. The fourth column shows the minimum, maximum and average time (Min(M), Max(M), Avg(M)) needed to apply the model on each code, in milliseconds. This time does not include the time required by the compiler to extract the input information of the model from the source code to analyze, since this time is highly dependable on the compiler infrastructure where the model is integrated. The maximum time necessary for the whole analysis, including the compiler time, was 108 milliseconds. These times were measured in a system based on an AMD Athlon at 1.2 Ghz.

The prediction is valid in an extremely high percentage of the cases, 99.7%, for which the worst-case miss rate observed is on average only 0.38% below the predicted one. Also, actually only 0.03% of the simulations yields a worse miss rate than the prediction.

Section 3.4 explained that the probability our model fails in its WCMP estimation was quantified as $S \cdot P(x > k), x \in B(n, 1/S)$, S being the number of cache sets. That is, S times the probability that the base addresses of more than k of the n data structures are mapped to the same cache set. We validated this expression with the ST program because it is the one with the largest ErrorConf$\%$ and ErrorSim$\%$ errors in Table 1. The reason is that it is the program with the largest number, five, of data structures whose accesses follow the same pattern. In order to test the accuracy of our estimation of the theoretical percentage of errors, we run 10000 simulations changing randomly the base addresses of the data structures per each one of the 240 configurations considered. The average difference between the theoretical percentage of errors and the ErrorSim$\%$ measured per configuration was just 0.02%.

Fig. 9 compares the worst miss rate obtained by the

Table 1: Validation results

Code	# of confs.	Validation statistic	Modeling time (in ms)
DMXDM	2160	ErrorConf% $= 0.04\%$; ErrorSim% $= 0.004\%$; Avg($\Delta MR\%$) $= 0.23\%$; Avg($MR\%$) $= 6.05\%$ Min($\Delta MR\%$) $= 0.0\%$; Max($\Delta MR\%$) $= 6.63\%$; ($\Delta MR > 5\%$) $= 0.41\%$; ($\Delta MR > 10\%$) $= 0.0\%$	Min(M) =1; Max(M) $= 9$ Avg(M) $= 5$
ST	240	ErrorConf% $= 3.33\%$; ErrorSim% $= 0.33\%$; Avg($\Delta MR\%$) $= 1.78\%$; Avg($MR\%$) $= 12.99\%$ Min($\Delta MR\%$) $= 0.06\%$; Max($\Delta MR\%$) $= 38.44\%$; ($\Delta MR > 5\%$) $= 3.33\%$; ($\Delta MR > 10\%$) $= 3.33\%$	Min(M) $= 0$; Max(M) $= 5$ Avg(M) $= 8$
STENCIL	240	ErrorConf% $= 1.25\%$; ErrorSim% $= 0.125\%$; Avg($\Delta MR\%$) $= 0.48\%$; Avg($MR\%$) $= 6.32\%$ Min($\Delta MR\%$) $= 0.0\%$; Max($\Delta MR\%$) $= 48.96\%$; ($\Delta MR > 5\%$) $= 0.83\%$; ($\Delta MR > 10\%$) $= 0.83\%$	Min(M) $= 0$; Max(M) $= 3$ Avg(M) $= 1$
CNT	720	ErrorConf% $= 0.0\%$; ErrorSim% $= 0\%$; Avg($\Delta MR\%$) $= 0.19\%$; Avg($MR\%$) $= 11.71\%$ Min($\Delta MR\%$) $= 0.06\%$; Max($\Delta MR\%$) $= 1.02\%$; ($\Delta MR > 5\%$) $= 0.0\%$; ($\Delta MR > 10\%$) $= 0.0\%$	Min(M) $= 0$; Max(M) $= 4$ Avg(M) $= 1$
TRANS	720	ErrorConf% $= 0.64\%$; ErrorSim% $= 0.14\%$; Avg($\Delta MR\%$) $= 0.63\%$; Avg($MR\%$) $= 31.15\%$ Min($\Delta MR\%$) $= 0.0\%$; Max($\Delta MR\%$) $= 10.11\%$; ($\Delta MR > 5\%$) $= 3.20\%$; ($\Delta MR > 10\%$) $= 0.16\%$	Min(M) $= 0$; Max(M) $= 68$ Avg(M) $= 13$
FIBONACCI	240	ErrorConf% $= 0.0\%$; ErrorSim% $= 0\%$; Avg($\Delta MR\%$) $= 0.15\%$; Avg($MR\%$) $= 11.76\%$ Min($\Delta MR\%$) $= 0.06\%$; Max($\Delta MR\%$) $= 0.93\%$; ($\Delta MR > 5\%$) $= 0.0\%$; ($\Delta MR > 10\%$) $= 0.0\%$	Min(M) $= 0$; Max(M) $= 3$ Avg(M) $= 1$

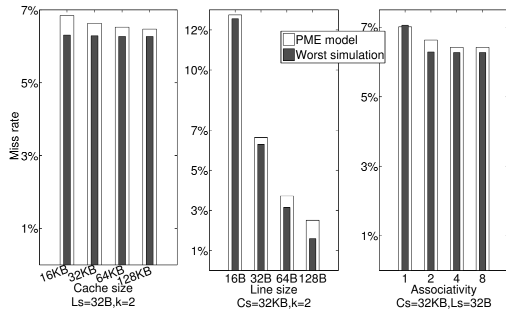

Figure 9: Worst-case miss rate prediction accuracy considering different cache configurations for the DMXDM program

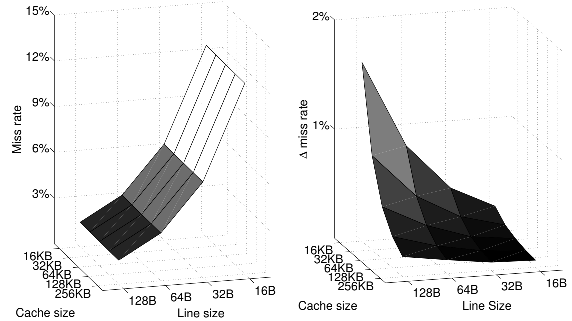

Figure 10: Evolution of the worst miss rate measured and the difference with respect to the worst miss rate predicted by the model as the cache and line size change in the DMXDM program

simulation and the one predicted by the model for the product of two 500×500 matrices. Considering the bar diagrams from left to right, in the first one the cache size is varied while the line size and associativity are fixed to 32 bytes and 2, respectively. The second diagram considers a 2-way cache of 32 KBytes, changing its line size. The third one, considers a cache of 32 KBytes and a line size of 32 bytes, with a varying associativity. The diagrams show that the model provides a tight estimation of the worst-case miss rate. The first experiment in the third graph caused a small error in the worst-case prediction. In this program two matrices follow the same access pattern in the innermost loop so an alignment of their base addresses with respect to the cache can cause an error in the prediction.

Fig. 10 shows the evolution of the worst miss rate obtained in the simulations (left graph) and the difference between that value and the one predicted by the model (right graph) for the product of two 1000×1000 matrices. Considering associativity 2, the cache and line sizes were varied. The model calculated always a tight and correct value for the worst-case miss rate. The biggest discrepancies between the simulation and the prediction happened in small caches with large lines, which is actually an uncommon situation in practice. In these cases the cache has fewer sets, and consequently the alignment of the data structures has a bigger influence in the total number of misses. The model considers a large overlapping in these caches between the data structures, but this is not matched by any of the 10 simulations performed for each configuration. However the smallest errors are obtained with big caches with small lines, in which the large number of cache sets makes miss rate less dependent of the alignment between data structures, and the prediction is tighter.

5. Related work

Several previous works have used analytical methods to calculate the WCMP in the presence of caches. The modeling of instruction-caches [1], [10] has had a lot of success, even recently in multicore systems with shared L2 instruction caches [2]. There are also many works devoted to the study of data caches. White et al. [3] 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. In most programs, the prediction of the worst-case predicted is equal to the value observed. However, in some programs similar to ours like the dense matrix product (DMXDM) it gets an overestimation of 10% for big data sizes. A version of our ST program presents also an overestimation of 17%.

Lundqvist and Stenström [4] 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). They dot not present any result.

Ramaprasad and Mueller [11] use the cache miss equations (CMEs) [12], 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. [6] 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.

6. Conclusions

This paper extends an existing model [7] to estimate the WCMP of regular codes in the presence of data caches at compile time. The main novelty of this work with respect to previous ones in this area is that our approach is the only one that requires no information about the base addresses of the data structures; only the cache configuration and the source code are needed. This ability is very interesting, since base addresses are sometimes unavailable at compile time, and they can change between different executions. This extends the scope of applicability to many situations other models cannot consider such as the usage of dynamically allocated memory, physically-indexed caches, etc. The model yields not only the number of misses generated, but also a formula for each reference and each nesting level which includes an estimation of the worst-case miss probabilities.

An extensive validation using trace driven simulations for a large number of different data sizes and cache configurations shows that this approach yields accurate and tight values of the WCMP. The main source of error of the WCMP calculation using this model is the lack of information about the base addresses of the data structures. Rarely, only in 0.03% of our 43200 simulations, concentrated in less than 0.3% of the problem-cache configurations, this led to underestimating the WCMP. As for tightness, the average difference between the worst-case miss rate predicted and measured was just 0.38%.

In the future, we plan to integrate our model prediction of the WCMP in an existing WCET tool such as [13]. We also intend to extend the WCMP model to cover irregular computations. 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 a safer WCMP calculation.

References

[1] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon, "Bounding pipeline and instruction cache performance," *IEEE Trans. Computers*, vol. 48, no. 1, pp. 53–70, 1999.

[2] J. Yan and W. Zhang, "Wcet analysis for multi-core processors with shared l2 instruction caches," in *IEEE Real Time Technology and Applications Symposium*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 80–89.

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

[4] T. Lundqvist and P. Stenström, "A method to improve the estimated worst-case performance of data caching," in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 1999, pp. 255–262.

[5] H. Ramaprasad and F. Mueller, "Bounding preemption delay within data cache reference patterns for real-time tasks," in *IEEE Real Time Technology and Applications Symposium*, 2006, pp. 71–80.

[6] X. Vera, B. Lisper, and J. Xue, "Data cache locking for tight timing calculations," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 1, 2007.

[7] B. B. Fraguela, R. Doallo, and E. L. Zapata, "Probabilistic Miss Equations: Evaluating Memory Hierarchy Performance," *IEEE Transactions on Computers*, vol. 52, no. 3, pp. 321–336, March 2003.

[8] S. Manolache, P. Eles, and Z. Peng, "Optimization of soft real-time systems with deadline miss ratio constraints," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2004, pp. 562–570.

[9] F. Mueller, "Compiler support for software-based cache partitioning," in *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*. New York, NY, USA: ACM, 1995, pp. 125–133.

[10] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm, "Cache behavior prediction by abstract interpretation," in *SAS '96: Proc. Third Intl. Symp. on Static Analysis*. Springer-Verlag, 1996, pp. 52–66.

[11] H. Ramaprasad and F. Mueller, "Bounding worst-case data cache behavior by analytically deriving cache reference patterns," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2005, pp. 148–157.

[12] S. Ghosh, M. Martonosi, and S. Malik, "Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior," *ACM Trans. on Programming Languages and Systems*, vol. 21, no. 4, pp. 702–745, July 1999.

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