Static Analysis of the Worst-Case Memory Performance for Irregular Codes with Indirections

DIEGO ANDRADE, BASILIO B. FRAGUELA, and RAMÓN DOALLO,
Universidade da Coruña, Spain

Real-time systems are subject to timing constraints, whose upper bound is given by the Worst-Case Execution Time (WCET). Cache memory behavior is difficult to predict analytically and estimating a safe and precise worst-case value is even more challenging. The worst-case memory performance (WCMP) component of the WCET can only be estimated with the precise knowledge of the stream of data addresses accessed by the code, which is determined by the access patterns and the base addresses of the data structures accessed. The regularity of strided access patterns simplifies their analysis, as they are characterized by relatively few parameters, which are often available at compile time. Unfortunately codes may exhibit irregular access patterns, which are much more difficult to statically analyze. As for the base addresses of the data structures, they are not always available at compile-time for many reasons: stack variables, dynamically allocated memory, modules compiled separately, etc. This article addresses these problems by presenting a model that predicts an upper bound of the data cache performance for codes both with regular and irregular access patterns, which is valid for any possible base addresses of the data structures. The model analyzes irregular access patterns due to the presence of indirections in the code and it can provide two kinds of predictions: a safe hard boundary that is suitable for hard real-time systems and a soft boundary whose safeness is not guaranteed but which is valid most of the times. In fact, in all our experiments the number of misses was below the soft boundary predicted by the model. This turns this soft boundary prediction into a valuable tool, particularly for non and soft real-time systems, which tolerate a percentage of the runs exceeding their deadlines.

Categories and Subject Descriptors: B8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms: Performance

Additional Key Words and Phrases: WCET, Cache memories

ACM Reference Format:
Andrade, D., Fraguela, B. B., and Doallo, R. 2012. Static Analysis of the worst-case memory performance for irregular codes with indirections. ACM Trans. Architec. Code Optim. 9, 3, Article 20 (September 2012), 32 pages.
DOI = 10.1145/2355585.2355593 http://doi.acm.org/10.1145/2355585.2355593

1. INTRODUCTION

Worst-Case Execution Time (WCET) must be calculated for Real-Time Systems (RTS) in the context of a schedulability analysis. The presence of cache memories complicates the compile-time estimation of a safe and tight upper bound of its Worst-Case Memory

This work has been supported by the Xunta de Galicia under projects INCITE08PXIB105161PR and UDC/GI-000265; and the Ministry of Education and Science of Spain, FEDER funds of the European Union (Project TIN2010-16735).
Authors' address: D. Andrade, B. B. Fraguela, and R. Doallo, Electronics and Systems Department, University of A Coruña, Spain; email: diego.andrade@ude.es.

DOI 10.1145/2355585.2355593 http://doi.acm.org/10.1145/2355585.2355593

Performance (WCMP) component. Good results have been obtained in the prediction of the WCMP in the presence of instruction caches [Healy et al. 1999; Yan and Zhang 2008]. The analysis of data caches [White et al. 1997; Lundqvist and Stenström 1999; Ramaprasad and Mueller 2006; Vera et al. 2007] is more challenging, as several references can access the same line simultaneously and the access patterns can be more irregular. The number of data cache misses varies largely depending on the access patterns and the data addresses accessed. The access patterns cannot be statically determined in irregular codes, and the data addresses depend on the base addresses of the data structures, which may not be available at compile-time and even change in different runs. There are several reasons for this: stack variables, dynamically allocated memory, modules compiled separately or by just-in-time compilers, etc.

The worst-case Probabilistic Miss Equations (wcPME) model introduced in [Fraguela et al. 2010] modifies the Probabilistic Miss Equations (PME) analytical model [Fraguela et al. 2003; Andrade et al. 2007b], to predict the WCMP for regular codes. This wcPME model inherits the PME model ability to make a prediction without information of the base addresses of the data structures involved in the code, which is a property not present in previous works in the bibliography. This characteristic is very interesting because the cache performance depends largely on the base addresses, as they modify the alignment of data with respect to the cache and thus the overlapping of the footprints of the data accessed on the cache.

The main contribution of this article to the wcPME model with respect to Fraguela et al. [2010] is the extension of the model to cope with irregular codes where the irregularity is due to indirections. This ability to predict the upper bound of the memory performance of this kind of irregular codes is a characteristic not present in the bibliography. The prediction of this bound must be tight, as an estimation far from the real behavior would impact negatively the system performance. In hard RTS the prediction must also be completely safe, as it should not be exceeded by any execution. The cache footprints of references with irregular access patterns due to indirections depend on the base addresses as well as on the particular values contained in the index arrays used in the indirections. The extension presented in this article can provide two kinds of predictions: a soft one that is not guaranteed to be absolute maxima, but which reflect realistic WCMP in practice, and a hard one that is absolutely safe.

Regarding the soft prediction, our extension considers both the worst-case alignments and reasonable worst-case contents of the index arrays. The memory performance observed in our experiments in Section 6 never exceeded the soft WCMP predicted by the model. Thus, while this soft prediction for irregular codes is not safe, it makes reasonable worst-case assumptions, which makes it a very valuable tool for soft RTS.

In the case of the hard prediction, there is a guarantee that this boundary will not be exceeded by any run. This kind of prediction is suitable for hard RTS. However, the predicted hard boundary is much higher than the soft one. In some cases, this boundary predicts a 100% miss rate (which means that all the accesses turns into misses). These high values of predicted hard boundaries do not necessarily indicate lack of tightness as in most cases this will be the worst-case behavior observed for some specific contents of the index arrays and base addresses. This happens mainly in direct mapped caches, however for set-associative caches lower miss rates are obtained. Both kinds of predictions, the soft and the hard one, can be used as an input to a WCET analyzer that needs some knowledge of the worst-case number of misses for each array reference.

The rest of this article is organized as follows. Section 2 introduces the worst-case PME model. Then, the method used to generate the soft prediction is presented. Thus, Section 3 describes the equations that the model generates to provide this kind of

prediction, which are based on miss rates associated to reuse distances. Then, Section 4 is devoted to the worst-case miss rate estimation process used to provide this soft prediction. Section 5 describes the method used to generate the hard safe prediction. This method is presented by explaining the differences with the soft version. A discussion on its safeness is also included in this section. Section 6 contains the experimental results, Section 7 is devoted to related work and Section 8 concludes the article.

2. THE WORST CASE PROBABILISTIC MISS EQUATIONS (WCPME) MODEL

The worst-case Probabilistic Miss Equations (wcPME) model [Fraguela et al. 2010] is a modification of the original PME model [Fraguela et al. 2003] to predict an upper bound of the number of misses generated by regular codes for any cache with a Less Recently Used (LRU) replacement policy. The model requires as inputs a representation of the code to analyze and the cache configuration. 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, which reflects the final structure of the executable. 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 so as not to endanger the precision of the prediction. The scope of application of the wcPME model is extended in this article to irregular codes where the irregularity is due to the existence of indirections. The proposed extension is able to provide a soft unsafe prediction, suitable for non-RTS, or a hard safe one, which is suitable for hard RTS. Section 2.1 describes in detail, the scope of application of our model and Section 2.2 introduces the basics of the wcPME model.

2.1. Scope of Application

The scope of application of the extension proposed in this article consists of codes with a set of normalized perfectly or nonperfectly nested loops. The number of iterations of each loop, or at least a bound, must be known at compile time. All the code parameters required for the analysis are not always available explicitly in the code. They can be derived through either program analysis [David and Puaut 2004], user annotation, or profiling. It is assumed that the required statistical information about the input data is provided by any of these mechanisms and its generation is not covered in this article.

The code may contain any number of references, which can be located in any nesting level. The existence of several references to the same data structure is also supported. The data accessed can be stored in any memory area (stack, heap, etc.) The reference indexes must be affine functions f_i either of the loops control variables $\mathtt{I}_i$ or of values read from arrays. An array whose values index another array is known as an index array and the array being indexed, the base array.

While the PME model has been extended to model data-dependent flows [Andrade et al. 2006], that is, codes with conditional statements and multiple paths, this article does not consider this extension, and its modification to compute the WCET is left as future work. The only conditionals allowed inside the portions of code analyzed in this article are those that only guard accesses to registers or to the latest data item accessed before the branch, so that the data-dependent flow cannot modify the cache behavior. These restrictions are common in the compile-time analytical models of the cache behavior [Ghosh et al. 1999; Fraguela et al. 2003; Xue and Vera 2004; Vera et al. 2007]. Inlining, either symbolic or actual, allowed PMEs to model inter-routine cache effects in Fraguela et al. [2003] and can be applied in the same way to the extensions proposed here. Besides, the applications that exhibit irregular access patterns, such as those arising from the usage of pointers or more complex conditional statements, can be made analyzable for the model by locking the cache before such patterns arise

and unlocking it after them. This latter technique is commonly used to enable cache predictability, particularly for enabling a tight computation of the WCET [Vera et al. 2007].

A clear proof of the large applicability of the model proposed in this article in real-world situations is that its scope of application is larger than [Fraguela et al. 2003], and the scope of that model sufficed to analyze complete SPECfp95 and Perfect Benchmarks applications, or at least their more significative and time-consuming routines. The support of codes with irregular access patterns due to indirections by our extension enables the modeling of an even wider range of codes.

2.2. Introduction to the wcPME Model

The wcPME model generates a formula F_{Ri}, called Probabilistic Miss Equation (PME), that analyzes the behavior of each static reference R during the execution of each loop at nesting level i that encloses R. This formula classifies the accesses produced by R during the execution of the loop as either potential first-time (or cold) misses or potential interference misses (which also include capacity misses). The first ones are produced when a line is accessed for the first time during the execution of the loop. These accesses cannot exploit reuse with respect to previous accesses in that loop. The second ones are the nonfirst accesses to a line in the loop, which may result in successful cache reuses within the loop. The success of a reuse attempt of a line depends on the number of other lines mapped to the cache set where this line resides, since the immediately previous access to the line. If that number exceeds or equals the cache associativity K, the reuse attempt will result in a miss. The reason is that the LRU replacement policy replaces a line before its reuse if and only if K or more other lines are mapped to its cache set before it is reused.

The probability that a given access results in a miss, called miss probability, depends on the footprint on the cache of the data accessed during the reuse distance. Probabilities are suitable to estimate the average performance but not the WCMP, where they must be replaced by worst-case miss rates. In the following, the term *miss probability* used in the original PME model [Fraguela et al. 2003] is replaced by *miss rate*. The reasons for this replacement are explained in detail in Section 4.

The reuse distance (RD) is defined as the piece of code executed since the last access to the line whose reuse may take place. Normally, a line can be reused with different reuse distances. In the case of references found in loop nests, which is the scope of the PME model, each loop enclosing a reference gives place to a different RD, which can be measured in terms of loop iterations, that (possibly) characterizes some of the reuses not captured by the inner loops. This way, the model estimates the number of misses generated by a reference by exploring the loops that enclose it from the innermost one to the outermost one. In each loop, the model builds a partial PME that adds information about the reuses whose RD is associated with that loop. Specifically, each partial PME estimates the number of accesses generated by the reference that cannot exploit reuse in the considered loop, the number of accesses whose RD is associated with this loop, and the associated worst-case miss rate for such reuses. The PME for each loop and static reference is expressed recursively in terms of the PME for the same reference in the immediately inner loop, so that it contains all the information for the behavior of the reference within the loop. The recursion finishes in the innermost loop, where the worst-case miss rate associated to the RD for each individual access is calculated. Thus, the PME associated with the outermost loop in a nest takes into account all the reuses, and its evaluation yields the number of misses generated by the reference during the execution of the loop nest.

In order to provide an upper bound of the number of misses produced by each reference, rather than the average value estimated in Andrade et al. [2007b], the

original PME model must be modified to: (1) Associate the longest possible RD to each reuse attempt and (2) Maximize the miss rate associated to each RD. This way, the calculation of this upper bound is a two steps process:

—In the first step, the PME F_{Ri} that models the behavior of the reference R in the loop at nesting level i computes the minimum number of reuse attempts within the loop and discovers the longest possible RD for each one of them. The construction of the PME is described in Sections 3 and 5.1 for the soft and the hard versions of the model, respectively. There are two different PMEs depending on whether the analyzed reference follows a regular access pattern with respect to the considered loop (Section 3.1) or an irregular one (Section 3.2). The hard prediction of the model is provided using the PME for the regular case described in Section 3.1 and using a new one for the irregular case, which is presented in Section 5.1.
—The model requires the calculation of the maximum miss rate associated to a given RD RegIn, called $MissR$(RegIn). Sections 4 and 5.2 explain how this miss rate is calculated for the soft and the hard version of the model, respectively.

3. PROBABILISTIC MISS EQUATIONS CONSTRUCTION: SOFT VERSION

This section explains the method used in the construction of the PMEs to generate the soft (unsafe) prediction of the model. Along this section, by simplicity we will use the terms worst-case and upper bound to refer to the predictions provided by this version of the model although they are not safe worst-case of upper bound values but reasonable pessimistic predictions in practice. As explained in Section 2.2, a partial PME F_{Ri} is built for each static reference R in the code and loop at nesting level i that encloses such reference. This PME estimates the number of misses that R generates during a complete execution of this loop. It is a sum of the number of accesses that enjoy each possible reuse distance (RD) associated with this loop multiplied by the miss rate that the memory regions accessed during that reuse distance generate. Of course every access that is the first one to a line in this execution of the loop, cannot result in reuses of lines already accessed, thus their miss rate cannot be associated to RDs within the loop. The miss rates for those accesses correspond either to (a) RDs that are associated with outer loops; or (b) RDs with respect to accesses to the same data in previous loops in the same nesting level, when we consider nonperfectly nested loops; or (c) when the loop is the outermost one ($i = 0$) and there are no preceding loops that could give place to reuses, the miss rate is simply one, since every first access to a line in this loop is indeed a first access to the line, unable to exploit any reuse, which results in a compulsory miss. A PME is always a function of the input parameter RegIn, which are the memory regions accessed during the reuse distance for what in this level of the nest happen to be first accesses. The reason is that PMEs are built beginning in the innermost loop and proceeding outwards, and their evaluation depends on memory regions associated with reuses that are calculated in outer or previous loops. The exception are the PMEs for outermost loops F_{R0}, in which no reuse from previous accesses is possible. This is modeled by using as RegIn, a memory region whose associated miss rate is one, so that the first-time accesses to a line in the nest are predicted as misses.

Each PME F_{Ri} is expressed in terms of the PME for the immediately inner loop that encloses R. The recursion finishes in the innermost loop containing the reference, where $F_{R(i+1)}$ simply stands for the miss rate for each access of R, $MissR$(RegIn), which is a function of the region accessed during its RD. This region is the RegIn input to $F_{R(i+1)}$.

The construction of F_{Ri} depends on whether the control variable for loop i, $\mathtt{I}_i$, is used in the indexes of index arrays found in the reference, or not. If $\mathtt{I}_i$ does not play any role in the indexing of index arrays in R, the reference has a steady access pattern after the

Table I. Notation Used

C_s	Cache size
L_s	Line size
K	Associativity of the cache
S	Number of cache sets
D_{Aj}	size of the j-th dimension of array A. Dimensions are numbered from left to right starting at 0. In a 2D matrix, the row index is dimension 0 and the column index dimension 1.
d_{Aj}	cumulative size of the j-th dimension of array A, $d_{Aj} = \prod_{i=j+1}^{N} D_{Ai}$
N_i	# of iterations of loop at nesting level i, whose index is I_i
S_{Ri}	stride of reference R with respect to the loop at nesting level i, $S_{Ri} = \alpha_{Rj} \times d_{aj}$, where j is the dimension of array a referenced by R indexed by I_i
L_{Ri}	# of different sets of lines (SOLs) accessed by reference R during the execution of the loop at nesting level i
D_{Ri}	# of different sets of lines (SOLs) that reference R can potentially access during the execution of the loop at nesting level i

first iteration of loop i and so it is regular with respect to this loop. The PME for this loop is built as a *Worst-Case Regular Access PME*, explained in Section 3.1. If, on the contrary, I_i participates directly or indirectly in the indexing of an index array in R, thus giving place to an indirection, the access pattern of R is irregular with respect to loop i. This gives place to a very different behavior of the reference that is modeled with a *Worst-Case Irregular Access PME* in Section 3.2. Both types of PME maximize the RD associated to each attempt of reuse and the number of lines accessed by the reference. This will maximize the number of misses generated for a reference if, in addition, the worst-case miss rate for each RD is used, which will be addressed in Section 4.

3.1. Worst-Case Regular Access PME

When the access pattern of a reference R is regular (strided) with respect to a loop, its worst-case behavior in that loop is modeled by the PME introduced in Fraguela et al. [2010]. The regularity guarantees that R follows the same access pattern in each iteration of the loop, and that the region it accesses in each iteration is the same as the one accessed in the previous one but displaced a constant stride. In this context, we call set of lines (SOL) this region that R accesses during the execution of one iteration of a loop. The SOL consists of a single line in the innermost loop that encloses R. In the loops at outer nesting levels it will usually consist of several lines. For example, if we consider a $M \times N$ C array (stored by rows) that is accessed by columns, the analysis of the inner loop discovers that each iteration references one line, thus the SOL consists of a single line. The analysis of the outer loop that controls the column index of the reference, discovers that each iteration of this loop is associated to the access to the set of lines that holds the elements of a column of the matrix. If the array is stored by rows, if $N \geq L_s$, where L_s is the cache line size measured in array elements, which is by far the most usual situation, each SOL will be made up of M different lines.

Regarding the notation of our model, dimensions are numbered from left to right starting at zero. In the case of a bidimensional array, the row index is dimension 0, while the column index is dimension 1. D_{Aj} is the size of the j-th dimension of array A. The model assumes, without loss of generality, that multidimensional arrays are stored by rows. Thus, d_{Aj} the cumulative size of the j-th dimension of array A is calculated as $d_{Aj} = \prod_{k=j+1}^{n} D_{Ak}$. Table I depicts these parameters and others that will be referenced during the explanation of the model.

The regular access PME classifies the iterations of the corresponding loop in two groups: those in which R accesses a SOL for the first time, and those in which R

reuses an already accessed SOL. The WCMP is achieved when the number of SOLs is maximized. This is because the iterations that access a new SOL for the first time in the execution of the loop, have longer reuse distances than those that reuse a SOL within the loop. Thus we need to estimate the maximum number of different SOLs that R can access during the execution of loop i, given by

$$L_{Ri} = min\left\{N_i, \left\lceil \frac{S_{Ri}(N_i - 1) + L_s}{L_s} \right\rceil \right\}, \quad (1)$$

where N_i is the number of iterations of the loop, and S_{Ri} the stride of R with respect to loop i. The rationale of this formula is that during the execution of loop i, reference R performs N_i accesses separated by a stride S_{Ri} to SOLs. Therefore, there are $S_{Ri} \cdot (N_i - 1)$ elements between the first element of the SOL accessed by R in the first iteration of the loop, and the one accessed in the last iteration. Additionally, up to $2 \cdot L_s - 2$ more elements can be brought to the cache in the first and the last line of this memory region accessed by R. The second term inside *min* in Equation (1) takes into account both facts to estimate the maximum (worst-case) number of lines that can be brought to the cache if the whole region between both elements is accessed, each line defining a SOL. Now, if $S_{Ri} > L_s$ not all the lines will be accessed, as the stride leads R to skip some of them. Besides it is impossible to access more than N_i different SOLs in N_i iterations. Therefore Equation (1) adjusts L_{Ri} to be the minimum between N_i and the expression just discussed, yielding a safe and tight estimation of the maximum number of SOLs accessed. The stride S_{Ri} is a constant, since either the loop i index variable I_i does not index reference R, or the index we are considering is an affine function of I_i. In the former case, trivially $S_{Ri} = 0$ and $L_{Ri} = 1$, since the iterations of the loop do not lead the reference to access different data sets. In the latter case $S_{Ri} = \alpha_{Rj} \times d_{Aj}$, where j is the dimension whose index depends on I_i; α_{Rj} is the scalar that multiplies I_i in the affine function. For simplicity, in all the terms and formulas, sizes and strides are expressed in elements of the array whose access is being analyzed rather than in bytes. Equation (1) computes the maximum number of different lines accessed during N_i iterations with step S_{Ri}, each line defining a SOL. The equation assumes that the first element accessed is placed at the end of a cache line, which maximizes the number of SOLs affected.

Since PME $F_{R(i+1)}$ provides the number of misses of R during an execution of the loop at level $i + 1$, it also provides its number of misses during one iteration of the loop at level i, that is, during the access to the SOL associated to that iteration. Regarding the SOLs reused in different iterations of loop i, when a reference follows a strided access with respect to a loop, it can only reuse the same line in consecutive iterations of the loop. Thus the reuse distance for the reuses of SOLs in a loop in which R follows a regular pattern, the case we are considering, is always one iteration of the loop. As a result of both facts, the maximum number of misses generated by R at nesting level i is estimated by the PME:

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

where N_i is again the number of iterations of the loop at nesting level i, and L_{Ri} is derived from Equation (1). $\text{Reg}_{Ri}(j)$ 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 term is the number of misses generated by the first accesses in loop i to the L_{Ri} different SOLs that R accesses in the scope of this loop. The reuse for these accesses can only happen with respect to outer or preceding loops, thus number of misses is obtained evaluating $F_{R(i+1)}$ passing as parameter the value RegIn provided by those

```
for(i=0;i<m;i++) { // Level 0
  reg = 0;
  for(j=r[i];j<r[i+1];j++) { // Level 1
    reg = reg + a[j] * x[c[j]];
  }
  d[i] = reg;
}
```

Fig. 1. Sparse Matrix-Vector Product.

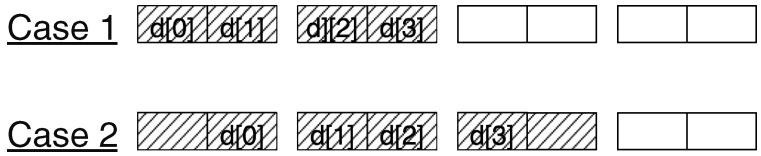


Fig. 2. Lines accessed depending on the base address of array `d`.

external loops. The second term 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, so the argument to $F_{R(i+1)}$ for these iterations is the set of memory regions accessed during one iteration of loop i. In the topmost nesting level of code the PMEs are evaluated with a RegIn whose miss rate is 1, which means no reuse due to any previous access is feasible.

Example 3.1. We will use as ongoing example to illustrate the construction of the different kinds of PMEs the sparse matrix-vector product code in Figure 1, where the matrix is stored in CRS[1] (Compressed Row Storage) format [Barret et al. 1994]. The code multiplies an sparse `m` $\times$ `n` matrix (stored in arrays `a`, `c` and `r`) that contains Nnz nonzero values and a vector `x` of size `n`. Let us consider that `m` is 4 and a cache with line size $L_s = 2$. If we analyze the reference R=`d(i)` in the context of the loop at nesting level 0, we see that the variable that controls the loop, indexes this reference by means of the affine function $1\times$`i`$+0$. Thus, the regular access PME of Equation (2) models the behavior of this reference in this loop. In order to apply it, we must first calculate the number L_{R0} of different sets of lines this reference accesses during the execution of this loop by means of Equation (1). Since the number of iterations of this loop is 4 and the stride S_{R0} is 1, we get that at most $L_{R0} = min\{4, \lceil \frac{1(4-1)+2}{2} \rceil\} = 3$ different SOLs are accessed. Figure 2 shows the influence of the base address of `d` in the number of different lines affected by the access. The figure represents two mappings of the 4 elements of vector `d` on the example cache considering different base addresses. The lines accessed during the first execution of the innermost loop are marked. We can see that in Case 1 the 4 elements of `d` are spread on 2 lines while in Case 2 they are spread on 3 lines, which matches our worst-case L_{R0} prediction. Replacing L_{R0} in Equation (2) we get the PME that calculates an upper bound of the number of misses produced by that reference,

$$F_{R0}(\text{RegIn}) = 3 \times F_{R1}(\text{RegIn}) + (4-3) \times F_{R1}(\text{Reg}_{R0}(1)) \,. \tag{3}$$

[1]The CRS (Compressed Row Storage) format stores sparse matrices by rows in a compressed way using three vectors. One vector stores the nonzeros of the sparse matrix ordered by rows, another vector stores the column indices of the corresponding nonzeros, and finally another vector stores the position in the other two vectors where the nonzeros of each row begin. In the example code these vectors are called `a`, `c` and `r`, respectively.

Since loop 0 is the innermost loop containing the reference R=d(i), F_{R1}(RegIn) must be substituted by $MissR$(RegIn) and $F_{R1}(\text{Reg}_{R0}(1))$ by $MissR(\text{Reg}_{R0}(1))$.

3.2. Worst-Case Irregular Access PME: Soft Version

When the control variable for loop i, $\texttt{I}_i$, indexes directly or indirectly an index array in an indirection, the access pattern of the base array of our reference R is irregular with respect to loop i. The reason is that the address accessed by R no longer depends directly on $\texttt{I}_i$, but on the value read from the index array that $\texttt{I}_i$ indexes either directly or through more levels of indirection. In that scenario, both the number of sets of lines (SOLs) accessed in each execution of the loop, and the number of iterations of the loop between different accesses to a given SOL, which is their reuse distance (RD), are variable.

Since the wcPME model tries to provide an upper bound for the number of misses, it assumes the situation that maximizes both the number of SOLs and the RD among all the possible ones. This way, the largest number of misses will be achieved when L_{Ri}, the number of different SOLs that R can access during the execution of loop i, reaches its maximum value. When loop i governs an indirection in reference R,

$$L_{Ri} = \min\{N_i, D_{Ri}\}, \tag{4}$$

where N_i is the number of iterations of loop i and D_{Ri} is the number of different SOLs that R can potentially access during the execution of loop i. The rationale of this formula is that a maximum of D_{Ri} SOLs are accessed during the execution of loop i, unless this value is larger than the actual number of iterations N_i.

The maximum number D_{Ri} of different SOLs that R can potentially access during the execution of the loop i is given by

$$D_{Ri} = \begin{cases} \left\lceil \dfrac{D_{\text{A}j} d_{\text{A}j} - 1 + L_s}{\max\{S_{Ri}, L_s\}} \right\rceil & \text{if } \nexists v / i < v \wedge \text{DimInd}_R(v) = \text{DimInd}_R(i) \\ \lceil D_{Rt}/L_{Rt} \rceil & t = \min\{v / i < v \wedge \text{DimInd}_R(v) = \text{DimInd}_R(i)\}, \end{cases} \tag{5}$$

where $S_{Ri} = \alpha_{Rj} \times d_{\text{A}j}$, and $d_{\text{A}j}$ and $D_{\text{A}j}$ are defined as in the preceding section. Let us remember that j is the dimension that is indexed, in this case indirectly, by the loop index. This also means that in this case the constant α_{Rj} is multiplying the indirection indexed by the loop index rather than the variable of the loop index itself. The first case of this equation computes D_{Ri} when there are no loops v nested inside i ($i < v$) such that their index variable $\texttt{I}_v$ indexes (directly or indirectly) the same dimension of the reference R as the variable $\texttt{I}_i$ of the considered loop i ($\text{DimInd}_R(v) = \text{DimInd}_R(i)$). The rationale of the expression is that the total size of the dimension indexed by the indirection is divided by the stride of the indirection on this dimension, or the line size (whichever is larger) to calculate the number of SOLs on which the indirection is defined. The numerator adds $L_\text{s} - 1$ and the result is rounded up to take into account the worst-case alignment of the access with respect to the cache lines. When such loops exist, the second case of the equation calculates D_{Ri} from D_{Rt} and L_{Rt}, where t is the outermost loop nested inside i that indexes this dimension. If in the immediately inner loop t that controls the indirection there are D_{Rt} SOLs of which L_{Rt} are accessed, this leaves $\lceil D_{Rt}/L_{Rt} \rceil$ SOLs for this loop level.

This definition of D_{Ri} allows to handle correctly those cases in which, for example, the indirection for a given dimension in R depends on several loop index variables, for instance, in a(b(i,j)) both i and j index indirectly the only dimension of vector a. Another example for this situation is often found in the codes where indirections are generated by sparse matrices because of the formats used to store them.

Example 3.2. The modeling of the behavior of reference `x(c(j))` in the code of Figure 1 will be used as a running example throughout this section. This code multiplies a $m \times n$ sparse matrix in CRS format by a vector `x` of length n. We will assume a cache configuration ($C_s = 8$, $L_s = 2$, $K = 1$), $m = 4$, $n = 8$, and that the number of nonzeroes Nnz of the sparse matrix is 8. The modeling of this reference requires an upper bound for Nnz because from it we can estimate the number of accesses across the indirection. This information is provided externally to our model and its obtention is not covered in this work. If that value is not available the worst-case situation is that the matrix does not have null values, thus, Nnz is equal to the matrix size. However, the number of nonzero values is usually small, so, that assumption would distance the model prediction from the real behavior.

L_{R1}, the number of different SOLs that R can access during the execution of loop 1 (the innermost one), can be calculated using using Equation (4) in nesting level 1. For this purpose, D_{R1}, the number of different SOLs that R can potentially access during the execution of that loop, must be calculated in advance. The first case of Equation (5) is used because loop 1 is the innermost loop that governs the indirection. It calculates D_{R1} knowing that (a) the indirection takes place in the first dimension of the base array `x` ($j = 1$), (b) the cumulative size for the first dimension of any array is always one ($d_{x1} = 1$), (c) the stride S_{R1} of the reference with respect to its indirection is one ($S_{R1} = \alpha_{R1} \times d_{x1} = 1 \times 1$), and (d) the size of the first (and only) dimension of `x` is $D_{x1} = 8$. With these data Equation (5) yields $D_{R1} = 5$, that is, during each iteration of loop `j`, `x(c(j))` can potentially access any of the 5 lines that constitute `x` in the worst-case. The average number of iterations of the loop $N_1 = Nnz/m = 8/4 = 2$, since the innermost loop sweeps along the elements of a row of the sparse matrix, Nnz being the number of nonzeroes in the sparse matrix and m its number of rows. This way Equation (4) yields $L_{R1} = min\{N_1, D_{R1}\} = min\{2, 5\} = 2$.

L_{R0}, the number of different SOLs that R can access during the execution of loop 0 (the outermost one), is computed also applying Equation (4). The reason is that the outermost loop also indexes the reference across the indirection, in this case indirectly, because the index `j` of the innermost loop depends on `r(i)`. In this loop, $N_0 = 4$ and Equation (5) calculates D_{R0}, which is $\lceil 5/2 \rceil = 3$, using the second case of Equation (5). The reason is that loop $t = 1$ also indexes the array across and indirection and $L_{R1} = 2$, while $D_{R1} = 2$. Replacing these values in Equation (4) yields $L_{R0} = min\{N_0, D_{R0}\} = min\{4, 3\} = 3$.

If L_{Ri} is smaller than N_i, the number of iterations of loop i, there will be some reuse attempts. Namely, L_{Ri} iterations generate first-time accesses to SOLs, while the remaining $N_i - L_{Ri}$ iterations can reuse a previous access to any of those SOLs. The RD between two reuse attempts is not necessarily one iteration of the loop, as in the regular case. Instead, it depends on the particular sequence of values contained in the index array. The worst-case cache performance takes place when this sequence maximizes the RD. This happens when the indirection accesses the SOLs cyclically each L_{Ri} iterations, resulting in a RD of L_{Ri} iterations for all the reuse attempts. That situation is modeled with the following PME,

$$F_{Ri}(\text{RegIn}) = L_{Ri} \times F_{R(i+1)}(\text{RegIn}) + (N_i - L_{Ri}) \times F_{R(i+1)}(\text{Reg}_{Ri}(L_{Ri})) \,. \tag{6}$$

Example 3.3. The modeling of the behavior of reference `x(c(j))` in the code of Figure 1 starts in the innermost loop containing the reference. The behavior of `x(c(j))` in this loop is modeled by the irregular access PME in Equation (6) because the loop index `j` controls the reference across an indirection. Since $N_1 = 2$, as it was seen in Example 3.2, the PME for this loop is

$$F_{R1}(\text{RegIn}) = L_{R1} \times MissR(\text{RegIn}) + (2 - L_{R1}) \times MissR(\text{Reg}_{R1}(L_{R1})),$$

where PME for the inner loop, F_{R2}, has been replaced by the miss rate calculation, as this is the innermost loop containing the reference.

The outermost loop also controls the indirection in `x(c(j))`, so it is also modeled with the PME of Equation (6). Since $N_0 = m = 4$, we have

$$F_{R0}(\text{RegIn}) = L_{R0} \times F_{R1}(\text{RegIn}) + (4 - L_{R0}) \times F_{R1}(\text{Reg}_{R0}(L_{R0}))$$

Equation (4) yielded the maximum number L_{Ri} of iterations of loop i that cannot exploit either spatial or temporal locality with respect to any previous iteration of that loop. This value of L_{Ri} always provides the worst-case cache performance when there are not outer loops that index the same dimension. The reason is that this strategy maximizes L_{Ri}, and consequently the first term of the PME (6), while it minimizes the second one. The miss rate for the accesses in the first term is always larger, as it is associated to reuses with respect to previous loops or iterations of outer loops, which include then one or more full executions of the considered loop i. On the other hand, the second term is associated to a given number of iterations of the current loop, during which a smaller interference region is accessed, and thus, whose corresponding miss rate is smaller.

However, if there is an outer loop m that indexes the same dimension j across an indirection and Equation (4) is used in the modeling of both loops, the maximization of L_{Ri} results in a minimization of L_{Rm}. When Equation (4) is applied at level m, N_m is a constant (it does not depend on the method used to calculate L_{Ri}) and D_{Rm} adopts is maximum value as the second case of Equation (5) is used in its calculation, which depends on L_{Ri}. The opposite strategy would be to minimize L_{Ri}, so that L_{Rm} and, consequently, the RD in loop m is maximized, which may increase the miss rate associated to the reuses in that loop. The minimum L_{Ri} is trivially 1; this would happen if all the elements of the index array accessed during one complete execution of loop i contained the same value. If it is assumed that all the values must be different, which is a reasonable worst-case assumption, L_{Ri} is minimized when the N_i values in the index array are consecutive and the addresses they map to occupy the minimum number of lines. This is modeled by the equation

$$L_{Ri} = 1 + \left\lfloor \frac{N_i - 1}{\max\{L_s/S_{Ri}, 1\}} \right\rfloor. \tag{7}$$

Both strategies, maximizing L_{Ri} in either the inner or the outer loop, must be evaluated to find out which one maximizes the number of misses. Although we do not have a proof of its safeness, our approach of evaluating these two opposite situations and choosing the worst one, makes reasonable worst-case assumptions. In fact, in our validation in Section 6 the WCMP observed never exceeded the prediction.

Example 3.4. L_{R0} and L_{R1}, the number of different SOLs that reference `x(c(j))` can access during the execution of loop 0 and 1 respectively in the code of Figure 1 were calculated in Example 3.2 using using Equation (4), obtaining that $L_{R0} = 2$ and $L_{R1} = 3$. In we expand the PMEs derived in Example 3.3 and replace it by these values, we get

$$F_{R0}(\text{RegIn}) = 6 \times MissR(\text{RegIn}) + 2 \times MissR(\text{Reg}_{R0}(3)),$$

which means that 6 accesses cannot exploit reuse within the loop nest and 2 have a RD of 3 iterations of the outermost loop. Figure 3, case (a), shows the sparse matrix that produces this kind of worst-case access pattern in the accesses to vector `x`. Nonzero values in the matrix, and the elements accessed in `x` in each iteration of the loops are colored grey. The thick vertical black bar between some elements of `x` represents the limits of each cache line. Using this matrix, 5 accesses cannot exploit reuse within

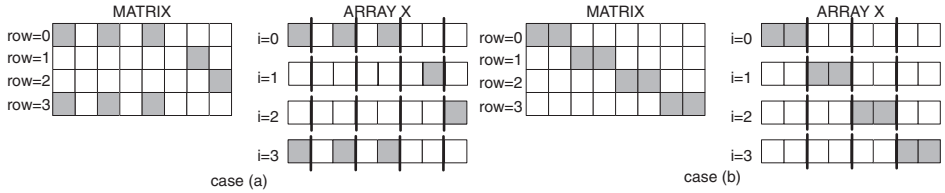


Fig. 3. Worst-case matrix modeled and elements of array x accessed in each iteration of the outermost loop of Figure 1.

the loop, while 3 have a RD of 3 iterations of the outermost loop, which produces a better performance than the WCMP estimation of the model. The worst-case miss rate associated to $\text{Reg}_{R0}(3)$, computed following the method that will be described in Section 4, is 1.0, and we assume no reuses from previous pieces of code, that is, $MissR(\text{RegIn}) = 1$. Thus, in this case $F_{R0}(\text{RegIn}) = 8$.

If L_{R1} is calculated using Equation (7) and L_{R0} using Equation (4) we get $L_{R1} = 1 + \lfloor \frac{2-1}{\max\{2/1,1\}} \rfloor = 1$, which would lead L_{R0} to adopt the value $L_{R0} = min\{N_0, D_{R0}\} = min\{4, 5\} = 4$, where D_{R0} is calculated using the second part of Equation (5), where $t = 1$, $D_{R1} = 5$ and $L_{R1} = 1$. Substituting these values and composing the PMEs, we get

$$F_{R0}(\text{RegIn}) = 4 \times MissR(\text{RegIn}) + 4 \times MissR(\text{Reg}_{R1}(1)),$$

that is, there are four accesses that cannot exploit reuse within the loop nest and another four that enjoy a RD of one iteration of the innermost loop. The sparse matrix associated to this possibility and the corresponding accesses on vector x are depicted in Figure 3, case (b). As we can see, this situation also requires a different mapping of x on lines. Following the procedure that will be described in Section 4 to calculate miss rates, we get $MissR(\text{Reg}_{R1}(1)) = 0.5$. Also, since no reuses from previous pieces of code are modeled in this example, $MissR(\text{RegIn}) = 1$. Thus in this case $F_{R0}(\text{RegIn}) = 6$.

As we have seen, in this example the use of Equation (4) to calculate L_{R1} produced the largest number of misses for this code, matrix and cache characteristics. That is, the worst-case memory performance is achieved spanning as much as possible in different lines the elements accessed in the innermost loop across the indirection. This distribution maximizes the RD of the reuse attempts in the innermost loop. However, if $C_s = 64$, $L_s = 2$ and $K = 1$, the number of iterations of the outermost loop, m, is 35, array x has size 8, and the number of nonzeroes Nnz of the sparse matrix is 128, the use of Equation (4) in the calculation of L_{R1} produces 35 misses, while the use of Equation (7) produces 49.34 misses. So, for this other matrix and cache configuration the best strategy to estimate a reasonably safe WCMP is to assume that the positions accessed across the indirection are consecutive in the innermost loop in order to maximize the RD in the outermost loop.

4. WORST CASE MISS RATE CALCULATION: SOFT VERSION

This section explains the method used in the calculation of the worst-case miss rate to generate the soft (unsafe) prediction of the model. As we have seen in Section 3, in the PME equation for the innermost loop containing a reference R, the PME for the immediately inner level $F_{R(i+1)}$ evaluated for a given input RD RegIn, is substituted by $MissR(\text{RegIn})$, the worst-case miss rate associated to that RD. Let us remember that a RD is the portion of code executed between two consecutive accesses to a line. The worst-case miss rate associated to a RD is an upper bound of the rate of accesses that can exploit that RD, which can result in misses. Thus, multiplying it by the number

of accesses that experience that RD yields an upper bound of the number of misses among those accesses.

The model described in this article uses worst-case miss rates instead of miss probabilities (average miss rates), which is what the original PME model in Fraguela et al. [2003] estimated. Now the worst-case overlapping of the lines to be reused and the interfering lines is computed. This is the mapping that places the largest possible number of lines to reuse in the sets that receive the largest number of lines from the considered interfering access pattern. Then, the model calculates the ratio of lines of the reuse region, the region whose reuse we are studying, that have to compete with a given number of interfering lines in their set in this worst case overlapping. The changes in the model to compute worst-case miss rates instead of probabilities were explained in detail in Fraguela et al. [2010]. This article also includes an explanation of why the results generated following this approach are indeed worst-case miss rates.

This section explains the computation of the worst-case miss rate associated to a RD in four stages. Sections 4.1 through 4.4 develop these four stages: access pattern identification, cache impact estimation, worst-case overlapping adjustment, and area vectors union.

4.1. Worst Case Access Pattern Identification

This first step identifies the kind of access pattern followed by each reference during the RD. The main access patterns are the sequential access to M elements, the access to M regions of size N separated by a constant stride S, and their irregular counterparts: the sequential access with uniform probability P of access per element and the access to groups of elements separated by a constant stride with uniform probability of access P per element. Access pattern identification is described in detail in Andrade et al. [2007b] and needs no changes for WCMP prediction, thus it is not detailed here due to space limitations.

4.2. Worst Case Cache Impact Estimation

This second step of the worst-case miss rate calculation measures the impact on the cache of each access pattern identified by the previous step. The impact is quantified with a vector V of $K+1$ elements called area vector (AV), K being the associativity of the cache. Each component of an AV is a ratio or probability of interference, that is, it is the probability a line to be reused conflicts in its cache set with a given number of lines from the access pattern that the AV characterizes. Its first element, V_0, is the ratio of sets that receives K or more lines from the access, while $V_i, 0 < i \leq K$ is the ratio of sets that receive $K-i$ lines. Since a K-way cache with LRU replacement expels a line if it has placed K or more different lines in its set before it is reused, component 0 of an AV is the miss rate generated by the access pattern. The other ratios are used in the union operation described in Sec. 4.4, since lines from different access patterns can be mapped to the same set and thus combined to fill it and increase the miss rate. We now describe how to compute the AV for each access pattern.

4.2.1. Worst Case Cache Impact Estimation for Regular Accesses. The sequential access to n consecutive elements generates an area vector $\mathrm{AV}_s(n)$:

$$\begin{aligned} &AV_{s_{(K-\lfloor l \rfloor)}}(n) = 1 - (l - \lfloor l \rfloor) \\ &AV_{s_{max\{0,(K-\lfloor l \rfloor-1)\}}}(n) = l - \lfloor l \rfloor \\ &AV_{s_i}(n) = 0 \quad 0 \leq i < K - \lfloor l \rfloor - 1, K - \lfloor l \rfloor < i \leq K, \end{aligned} \tag{8}$$

l being the average number of lines a cache set can hold simultaneously from the access, given by

$$l = \min\left\{K, Lines(n)/S\right\}, \tag{9}$$

where $Lines(n)$ is the the maximum number of lines that n consecutive elements from the referenced array can occupy and it is calculated as follows

$$Lines(n) = 1 + \lceil (n-1)/L_s \rceil. \tag{10}$$

It must be divided by S, the number of cache sets, to obtain the average number of lines that this region places in each cache set. 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 $\lceil (n-1)/L_s \rceil$ lines to the cache to access them. Thus in the worst case the number of lines brought to the cache is $1 + \lceil (n-1)/L_s \rceil$. As a consequence, a $l - \lfloor l \rfloor$ ratio of the cache sets receives $\lceil l \rceil$ lines while the remaining ratio $1 - (l - \lfloor l \rfloor)$ receives $\lfloor l \rfloor$.

Example 4.1. Reading the four elements of `d` in Figure 2, involves accessing 4 consecutive elements. The figure represents two possible placement of those elements on the cache. In Case 2 the four elements of `d` are spread on 3 lines, which is the worst-case alignment. According to Equation(8) the model predicts the AV $(0.75, 0.25)$ that represents the impact of that worst-case alignment, that is, that 3 out of the 4 cache sets $(3/4 = 0.75)$ have received $K = 1$ or more lines, while the other one $(1/4 = 0.25)$ did not receive any line

An upper bound of the impact on the cache of an access to M regions of size N separated by a constant stride S, can be estimated by the method described in [Fraguela et al. 2010], which is not included here due to space reasons.

4.2.2. Worst Case Cache Impact Estimation for Irregular Accesses. Let us consider the calculation of the worst-case area vector associated to an access to E elements that belong to a group of M consecutive elements, $\mathrm{AV}_{\mathrm{irreg}}(E, M)$.

The placement of the E elements accessed that maximizes their footprint on the cache must be considered. The region of the cache occupied by these E elements can not exceed the maximum area occupied by the M elements potentially accessible. In the worst case each one of the E elements brings a different line to the cache. Consequently, the maximum number of elements from the data structure brought to the cache is the minimum of $E \times L_s$ and M, as it cannot be larger than M. Since the access is irregular, the specific lines brought to the cache are unknown. Their worst-case mapping on the cache is the one that gives place to a larger miss rate. This happens when they are placed in the cache filling the maximum number of cache sets. That kind of placement for a group of n consecutive elements distributed in the area occupied by m consecutive elements generates an interference area vectors $AV_{\mathrm{fill}}(n, m)$, where all the components $AV_{\mathrm{fill}_i}(n, m)$ are zeroed except:

$$\begin{aligned}
&AV_{\mathrm{fill}_{(K-\lfloor l \rfloor - 1)}}(n,m) = min\left\{ l - \lfloor l \rfloor, \frac{\left\lfloor \frac{\mathrm{Lines(n)}}{\lceil l \rceil} \right\rfloor}{S} \right\} \text{ when } l < K \\
&AV_{\mathrm{fill}_{(K-\lfloor l \rfloor)}}(n,m) = \frac{\left\lfloor \frac{rem}{\lceil l \rceil} \right\rfloor}{S} \text{ when } \lfloor l \rfloor > 0 \\
&AV_{\mathrm{fill}_{K-(rem \bmod \lfloor l \rfloor)}}(n,m) = \frac{1}{S} \text{ when } \lfloor l \rfloor > 0 \\
&AV_{\mathrm{fill}_K}(n,m) = 1 - \textstyle\sum_{i=0}^{K-1} AV_{\mathrm{fill}_i}(n,m),
\end{aligned} \tag{11}$$

where S is the number of cache sets, $Lines(n)$ is the maximum number of lines that can be occupied by n consecutive elements, and l is the average number of lines from the access to m consecutive elements that a cache set can hold simultaneously. The calculation of $Lines(n)$ and l was detailed previously for the regular counterpart of

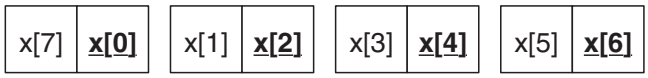


Fig. 4. Irregular access to several consecutive elements of array x.

Fig. 5. Irregular access to 2 groups of 3 elements separated by a distance 4 with a probability 0.57 of array x.

this access pattern. Finally, $rem = Lines(n) - AV_{\text{fill}_{(K-\lfloor l \rfloor -1)}}(n, m) \times \lceil l \rceil$, is the amount of lines from Lines(n) not assigned to $AV_{\text{fill}_{(K-\lfloor l \rfloor -1)}}(n, m)$. The rationale is that, as in the regular case, a $l - \lfloor l \rfloor$ ratio of cache sets would receive $\lceil l \rceil$ lines from the access if the m elements were accessed. Since only n elements are brought to the cache by the irregular access, the number of lines (Lines(n)) that they occupy are mapped on S cache sets in groups of $\lceil l \rceil$ lines. The unprocessed rem lines are mapped on S cache sets distributed in groups of $\lfloor l \rfloor$ lines. As rem may not be a multiple of $\lfloor l \rfloor$, the remaining $rem \ mod \ \lfloor l \rfloor$ are concentrated in one cache set. The unprocessed cache sets remain empty. As a result, the worst-case cache impact can be estimated as $\text{AV}_{\text{sp}}(E, M) = \text{AV}_{\text{fill}}(\min\{M, E \times L_{\text{s}}\}, M)$.

Example 4.2. Let us consider the access to 4 elements of an array x of 8 elements. The area vector that quantifies the impact of this access on the cache is $\text{AV}_{\text{irreg}}(4, 8)$. Figure 4 shows a mapping of the elements of array x in an example cache ($C_{\text{s}} = 8$, $L_{\text{s}} = 2$, $K = 1$). Let us note that the line containing the last element of array x (element 7) is mapped to the same cache set as the line containing the first element of the array (element 0). In the worst-case four cache lines are affected by the access and one element of each line is accessed. The impact of that access on this cache is represented by the AV (1, 0). The area vector $\text{AV}_{\text{irreg}}(8, 0.5)$ is calculated as $\text{AV}_{\text{fill}}(8, \min\{8, 8 \times 0.5 \times 2\} = \text{AV}_{\text{fill}}(8, 8)$ and, this way, the right AV (1, 0) is calculated.

The impact of an access to E elements that belong to an access pattern of M groups of N consecutive elements separated by a constant stride D, is calculated as $\text{AV}_{\text{irreg}}(E, M, N, D)$. As the access is strided, the total accessible region consists of $(M - 1) \times D + N$ elements. This way, its impact on the cache is calculated as $\text{AV}_{fill}(M \times L_{\text{s}} \times Lines(min\{N, E \times L_{\text{s}}\}), (M - 1) \times D + N)$, as it is also an irregular access, but in this case only E elements of each one of the M groups are accessed. These elements are spanned covering the maximum number of lines, thus, $min\{N, E \times L_{\text{s}}\}$ elements are affected by the access. As there are M groups, and each group can access a maximum of $Lines(min\{N, E \times L_{\text{s}}\})$ lines, the maximum number of elements brought to the cache by that access is $M \times L_{\text{s}} \times Lines(min\{N, E \times L_{\text{s}}\})$.

Example 4.3. The area vector associated to an irregular access to four elements, out of two groups of three elements separated by a distance of 4 elements, of an array x is $\text{AV}_{\text{irreg}}(4, 2, 3, 4)$. Figure 5 shows a possible mapping of the elements of array x in an example cache ($C_{\text{s}} = 16$, $L_{\text{s}} = 2$, $K = 1$). The four elements accessed across the indirections are underlined and the lines affected are marked. Each element belongs to a different cache line, which maximizes the number of lines affected. The AV that represents the impact of that access is (0.5, 0.5). The method described previously calculates that AV, as $\text{Reg}_{\text{rp}}(4, 2, 3, 4)$, which is modeled as $\text{Reg}_{\text{fill}}(2 \times 2 \times Lines(min\{3, 4\}), 1 \times 4 + 3) = \text{Reg}_{\text{fill}}(4 \times Lines(3), 7) = \text{Reg}_{\text{fill}}(4 \times 2, 7) = \text{Reg}_{\text{fill}}(8, 7)$, which yields the right AV (0.5, 0.5)

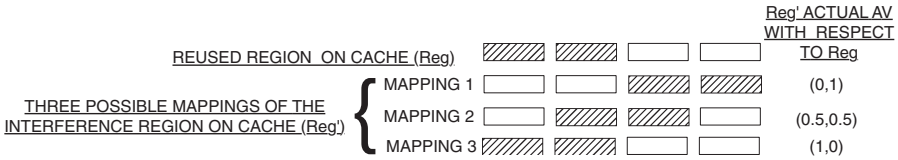


Fig. 6. Miss rate depending on the relative positions of the reused and the interfering memory regions.

The strategy to estimate the impact of the two types of irregular regions (AV_{fill}) is a potential source of unsafeness of our WCMP estimation. This strategy maximizes the number of cache sets filled by each individual region, thus maximizing the component 0 of the resulting AV. However, that strategy does not guarantee that when more regions are accessed during the reuse distance, the combined impact of all those regions provides the AV with the largest component 0 (miss rate).

Example 4.4. Let us consider a 2-way cache with $L_s = 2$ elements and 4 cache sets. During a given reuse distance four consecutive elements of an array are accessed, whose AV is $AV_s(4)$ =(0,0.75,0.25); in addition 2 elements out of 4 consecutive ones of another array are accessed, which yields an AV $AV_{irreg}(2, 4)$ =(0.25,0,0.75). The method to calculate the worst-case union of several AVs will be introduced in 4.4. The worst-case combined effect of both regions according to this algorithm is the AV (0.25,0.5,0.25), which maximizes the 0 component of the resulting AV. However, if the lines brought to the cache by the access to 2 of the 4 elements of an array were distributed in two different cache sets, this would yield the AV (0,0.5,0.5). This AV combined with the AV (0,0.75,0.25) belonging to the access to 4 consecutive elements yields a worts-case area vector (0.5,0.25,0.25), which has a largest component 0 (miss rate) than the AV previously calculated.

The previous example shows that maximizing the component 0 of the AV associated to an isolated region does not always guarantee that when it is combined with other AVs it maximizes the result.

4.3. Worst-Case Overlapping

The overlapping of the reuse and the interference regions on the cache can produce a larger interference between both regions than the one predicted by the component 0 of the AV of the interference region. The reason is that the AV obtained in the cache impact estimation process reflects the distribution of the lines of a memory region on the whole cache. Each element of the AV represents the ratio of cache sets that receives a given number of lines from that region. This is also the probability a line in a randomly chosen cache set has to compete for the set with a given number of lines from the region. This is what is needed to estimate average miss probabilities when the relative positions in the cache of the region whose reuse we are studying, which we will call reuse region, `Reg` and the interfering region, `Reg'`, are unknown. Now, when worst-case miss rates are to be predicted, a worst-case alignment of `Reg` and `Reg'` must be considered, not an average one.

Example 4.5. 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'`. The AV calculated for `Reg'` in the cache impact estimation process is (0.5, 0.5) for the three mappings, which indicates an average 50% probability of conflict, since `Reg'` fills two of the four cache sets. However, the actual interference with the cache sets of `Reg` is represented by the AV placed on the right side of each mapping in the figure. 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.

The average-case AV does not correspond to a worst-case alignment of the reusable and the interfering regions, but to an average-case one. The worst-case alignment will be the one in which the largest possible number of lines from region Reg have to compete with the largest 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 $AV_{Reg'_0} \times S$ sets that receive K or more lines from Reg', where $AV_{Reg'}$ is the AV for Reg' calculated in the standard way and S is the number of sets in the cache. Component 0 of $AVwc_{Reg'}$, the AV for Reg' considering this worst-case alignment, will be then the ratio of lines of Reg mapped to these full sets, as it is conversely the probability a randomly chosen line from Reg has to compete with K or more lines from Reg'. Once those full sets are exhausted, then the largest possible number of lines from Reg are mapped to the $AV_{Reg'_1} \times S$ sets that receive $K-1$ lines from Reg', and their ratio on the total number of lines of Reg will be $AVwc_{Reg'_1}$, and so on. This algorithm requires $AV_{Reg'}$, the standard AV for Reg', but it also needs 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 purpose 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. If for example in Figure 6 region Reg had occupied three sets with a single line each, its AV would have been $(0.75, 0.25)$, and since two of them would have collided in the worst case with Reg', $AVwc_{Reg'}$ would have been $(0.66, 0.33)$. 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 4 out of its 5 lines could collide with Reg' in the worst case, $AVwc_{Reg'}$ would have been $(0.8, 0.2)$. The wcPME model presented an algorithm to systematically calculate this worst-case overlapping in Fraguela et al. [2010].

4.3.1. Treatment of Full Alignments. This method estimates safely and tightly the maximum overlapping between the reuse and the interference region. However, its predictions can be very far from the average observed in those codes where there are references whose accesses may collide systematically in the same cache sets, a situation we call full alignment. Two references R and R' are potentially fully aligned when $S_{Ri} \mod C_{\mathrm{sk}} = S_{R'i} \mod C_{\mathrm{sk}}, \forall\, 0 \leq i \leq Z$, where S_{Ri} is the stride of reference R with respect to loop i as defined for Equation (2), $C_{\mathrm{sk}} = C_{\mathrm{s}}/k$ and Z is the innermost loop containing both references.

The full alignment actually takes place when the addresses accessed by the references are aligned with respect to the cache, that is, mapped to the same cache sets. Fully aligned references collide cyclically in the same cache sets. In the worst case they will be mapped to the same cache set in every iteration, in the best case in one out of L_{s} iterations. The corresponding accesses will result in misses when the number of lines involved per set is larger than the associativity.

The modular nature of our model allows to disable selectively the modeling of full alignments when desired. This is achieved by disabling the worst-case overlapping adjustments described in Section 4.3 only for the potentially fully aligned references in the innermost loop containing them. The user may wish to do this because if techniques such as padding or buffering have been applied to avoid the full alignments, the WCMP predicted (and observed) will be much smaller. Even if full alignments are not avoided explicitly, the model would provide a highly probable WCMP prediction, as the percentage of base address combinations that lead to full alignments is fortunately very small.

4.4. Worst Case Area Vectors Union

The previous step of the calculation of the miss rate 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 rate summarizes the effects of these AVs merging them into a global one through an union operation [Andrade et al. 2009] that calculates the maximum combined impact on the cache of all the AVs.

After the worst-case overlapping adjustment process described in Section 4.3 the meaning of the ratios of the AVs change. They are not ratios of cache sets that receive a given number of lines from a memory region anymore, as in the original PME model. They become ratios of lines to be reused that conflict with a given number of lines from the interfering region when the very worst-case overlapping between the reuse and the interfering regions is considered. This change of interpretation has no influence on the worst-case AV union algorithm. Without worst-case overlapping adjustment, this algorithm should combine ratios of cache sets with a given number of lines in order to maximize the final ratio of sets that receive K or more interfering lines. Now, after the adjustment, it should combine ratios of lines to reuse that conflict with a given number of lines in order to maximize the final ratio of lines of the reuse region whose reuse is hampered by K or more interfering lines. As we see the mechanics of the algorithm should be the same for both interpretations. The algorithm ensures that the resulting AV has the largest leftmost component that can be obtained by combining the ratios in the input AVs, and thereby it is a safe upper bound of the miss rate provided by the union operation. The details of this algorithm can be found in Fraguela et al. [2010].

5. HARD PREDICTION OF THE WORST-CASE MEMORY PERFORMANCE

Hard RTS require a safe estimation of the maximum number of misses. This safe estimation can be provided by our model with some modifications to the soft version described in Sections 3 and 4, namely, a safe version of the PME for the irregular case, and an adapted method to calculate the worst-case miss rate. These modifications are explained separately in Sections 5.1 and 5.2, respectively. Finally, Section 5.3 contains a discussion on the safeness of the proposed method.

5.1. Probabilistic Miss Equations Construction: Hard Version

The construction of the PME for references following regular access patterns does not change because the one presented in Section 3.1 is safe, as proved in Fraguela et al. [2010]. The safe worst-case PME for references following irregular access patterns (due to indirections) considers that none of the irregular accesses can reuse cache lines within any loop nest. Thus, the new form of this PME is as follows,

$$F_{Ri}(\text{RegIn}) = N_i \times F_{R(i+1)}(\text{RegIn}). \tag{12}$$

This prediction may not be very tight in some situations. A tighter safe prediction would be obtained analytically by searching exhaustively among all the possible sequences of accesses generated by the contents of the index array. For example, when D_{Ri} the number of different SOLs that R could potentially access during the execution of loop i is 4, then in the first iteration of loop i any of these 4 SOLs may be accessed through the indirection. So, these four possibilities turn out into the first four leaves of the tree, which reflects all the possible sequence of accesses. In the second iteration any of these 4 SOLs may be accessed, thus, each leaf of our tree is expanded to take it into account. At this point, the tree already has 16 leaves, which correspond to 16 possible sequence of accesses for the first two iterations of the loop that should be evaluated separately. The tree is expanded in each new iteration of loop i and the number of possible sequence is multiplied by D_{Ri}. At the end, if the loop has n iterations, there are 4^n possible sequence of accesses that must be evaluated separately. In

real-life applications, there are hundreds or thousands of SOLs accessible through the indirections (D_{Ri}) and loops have thousands of iterations (N_i). So in general, the number of possible sequences whose performance must be evaluated would be $(D_{Ri})^{N_i}$. The time and resources required to evaluate all these sequences would be impractical or even impossible in realistic situations, as the model should be applied independently to each possible sequence to calculate its associated memory performance.

When the contents of the index array are known to fulfill certain properties, a tighter estimation can be provided in a reasonable time. For example, if the analyzed loop i is the only one whose control variable indexes the reference across an indirection and the values contained in the index arrays are ordered, the PME for the regular case, Equation (2), can be used as a safe upper bound for references following irregular access patterns. The reason is that in each iteration of loop i a SOL is accessed for the first time in this loop or, it is a reuse attempt whose associated reuse distance (RD) is necessarily one iteration of that loop. The component L_{Ri} of Equation (2) would be calculated in this situation using Equation (4), which calculates its maximum value in the irregular case instead of Equation (1), which applies only to regular access patterns.

5.2. Worst-Case Miss Rate Calculation: Hard Version

As we explained in Section 4.2, the strategy to estimate the impact of the irregular regions (AV_{fill}) is a potential source of unsafeness of our WCMP estimation. The reason is that this strategy maximizes the number of cache sets filled by each individual region, thus maximizing the component 0 of the resulting AV. However, that strategy does not guarantee that when more regions are accessed during the reuse distance (RD) the combined impact of all those regions provides the AV with the largest component 0 (miss rate), as Example 4.4 showed.

A safe prediction of the worst-case miss rate associated to a given RD can be provided by modifying the method to calculate the worst-case miss rate. The safe method begins following the process described in Section 4, but only for regions associated to regular access patterns, and with the change that the worst-case overlapping process described in Section 4.3 is skipped. At the end of this process an AV, AV_{REG}, which summarizes the impact of all the regions associated to regular access patterns is obtained. This AV represents the worst-case distribution of the lines associated to the regions following regular access pattern among all the cache sets. This distribution is the one that fills more cache sets with lines from those interfering regions. This AV must be processed still in three stages. In the first one, the AV is mapped to a representation of its impact on the cache called $nlinesInterf$. This representation facilitates the calculation of the worst-case overlapping of the regular interfering regions represented by AV_{REG} and the irregular interfering regions as well as the reuse region. In the second stage, the interference generated by the irregular interfering regions is added to $nlinesInterf$. Finally, the worst-case overlapping between all the interference regions represented by $nlinesInterf$ and the reuse region is calculated. This gives place to the worst-case interference AV, and therefore to the maximum possible miss rate generated by the interferences. In this safe version of the algorithm, the adjustment to take into account this overlapping is different and it is performed after the area vectors union.

Figure 7 shows the algorithm used to change the representation of the impact of the interfering regions from an area vector to a vector of number of lines per set. The result of this algorithm is a vector $nlinesInterf$ of S components, where each component is associated to a different cache set and it represents the number of lines that this set holds due to the interfering regions represented by AV_{REG}. The loop between lines 2 and 6 is used to make a correspondence between the cache sets represented in AV_{REG} and the cache sets in vector $nlinesInterf$. The loop iterates across the $K+1$ components of AV_{REG}. First, the number of cache sets $nsets$ associated to the jth components of

```
function regProcess(AV_REG[K+1]) {
1   c = 0
2   for j = 0 to K {
3       nsets = AV_REG_j × S
4       nlinesInterf_i = K − j,  c ≤ i < (c + nsets)
5       c = c + nsets
6    }
7   return nlinesInterf
8 }
```

Fig. 7. Algorithm to change the representation of an area vector to a vector of number of lines per set.

```
function irregProcess(nlinesInterf_[S], nlines, l) {
1   i = 0
2   while (i < S) and (nlinesInterf_i = K) {
3       i = i + 1
4    }
5   while (i < S) and (nlines ≠ 0) {
6       tmp = min{nlines, K − nlinesInterf_i}
7       tmp = min{tmp, ⌈l⌉}
8       nlinesInterf_i = nlinesInterf_i + tmp
9       nlines = nlines − tmp
9       i = i + 1
10    }
11 }
```

Fig. 8. Worst-case overlap for irregular access patterns.

AV_{REG} is calculated (line 3). Then, line 4 assigns $K - j$ to $nlinesInterf_i$ for the *nsets* sets associated to this component of the AV. This represents that $K - j$ lines from the interference region represented by AV_{REG} are placed in *nsets* cache sets, which is the meaning of AV_{REG_j}. Notice that $nlinesInterf$ is filled in so that it is monotonically decreasing, that is, $nlinesInterf_i \geq nlinesInterf_j$ for any $i < j$.

Then, the worst overlapping of interfering regions associated to irregular access patterns with the regular ones already represented in $nlinesInterf$ is calculated. These regions are processed one by one using the algorithm *irregProcess* in Figure 8. In this algorithm, the processed interfering region is not represented by an AV. The reason is that since the access is irregular, the lines it brings to cache can be distributed in arbitrary ways on the cache sets. Therefore our strategy is to estimate the maximum number of lines *nlines* that the irregular access can bring to the cache, and calculate the worst distribution of these lines on the cache, that is, the one that will give place to the largest interference. The only restriction we will take into account for this distribution is that at most $\lceil l \rceil$ lines from the access can go to the same cache set, $\lceil l \rceil$ being the maximum number of lines a cache set can hold simultaneously from the access. This maximum value is calculated using l, the average number of lines from the access a cache set holds simultaneously. The method to calculate both values is different depending on whether the irregular access pattern accesses random positions on a region of a number of consecutive positions, or on a number of regions of consecutive positions separated by a constant stride. Both methods are explained in turn.

The maximum number of lines brought to the cache during the access to E elements that belong to a group of M consecutive elements, $\text{AV}_{\text{irreg}}(E, M)$, is $nlines = min\{E, Lines(M)\}$, where $Lines(n)$ is calculated using Equation (10) in Section 4.2.1. The rationale of the formula used to calculate *nlines* is that $Lines(M)$ is the maximum number of lines occupied by M consecutive elements, but no more than E different lines can be accessed.

The average number of lines from an access a cache set holds simultaneously, l, is calculated using Equation (9) in Section 4.2.1 where n is replaced by M. Thus, a maximum of $\lceil l \rceil$ lines from an access can be held simultaneously by a cache set.

```
function simToAV(sim_[S], nlinesInterf_i) {
1   lines = Σ_{i=0}^{S-1} sim_i
2   AV_(K−nlinesInterf_i) = AV_(K−nlinesInterf_i) + sim_i/lines ∀ i = 0...(S − 1)
3   return AV
4}
```

Fig. 9. Simulation to AV conversion.

The total accessible region by the access to E elements that belong to a memory region made up of M groups of N consecutive elements separated by a constant stride D, $\text{AV}_{\text{irreg}}(E, M, N, D)$, consists of $(M-1) \times D + N$ elements. This way, $nlines = min\{E, Lines((M-1) \times D + N)\}$ and l is calculated using Equation (9) in Section 4.2.1 where n is replaced by $Lines((M-1) \times D + N)$.

The algorithm *irregProcess* in Figure 8 receives the values of *nlines* and l representing the irregular interfering region, and the vector $nlinesInterf$, which stores the number of lines of interference per set brought by those regular and irregular interfering regions that have been already processed. The *nlines* lines associated to the irregular access to process are placed in those cache sets where they can help generate the largest possible interference, respecting the limitation of a maximum of $\lceil l \rceil$ lines per cache set. Let us remember that in a K-way cache with LRU replacement there is a miss in an attempt of reuse in a set if K or more lines have been placed in the set during the reuse distance. Therefore the algorithm tries to fill as many positions as possible of $nlinesInterf$ with a value K. The algorithm exploits the fact that the values in $nlinesInterf$ are monotonically decreasing. It first skips all the sets that already have K lines in the first loop. In the second loop it adds one line to those with $K-1$ lines, two to those with $K-2$ lines, etc. until in runs out of lines. This strategy maximizes the number of sets in which reuse attempts will result in misses, and thus models a worst-case overlap between the interference regions.

At the end of this processing of all the regions that can interfere with the reuse, the data structure $nlinesInterf$ contains the number of lines each cache set receives from the interfering regions following the worst-case overlapping policy. This data structure must be converted into an AV, in which each component will represent the ratio of lines from the reuse region that compete in a cache set with a given number of lines from the interfering regions. This process requires a simulation of the distribution of the lines of the reuse region *Reg* on the cache whose output is a vector *sim* of S elements, S being the number of cache sets. Each component of this vector is associated to a different cache set and it contains the number of lines that this set receives from the reuse region. The elements of the vector *sim* are sorted in decreasing order of their contents. Algorithm $simToAV$ in Figure 9 receives *sim* and $nlinesInterf$ as parameters. Line 1 in the algorithm computes the total number of lines in the reuse region. Then, line 2 computes the makes a correspondence between the cache sets represented in *sim* with the cache sets in $nlinesInterf$. This correspondence is reflected in an AV where each components contains the ratio of lines from the reuse region that conflict with a certain number of lines from the interference region in their set. As both vectors are monotonically decreasing and they are processed from left to right, this guarantees that the sets containing most lines from the reuse region are filled also with the maximum number of lines from the interfering region. This guarantees that the maximum miss rate is calculated.

5.3. Safeness of the Method

The safeness of the wcPME model to predict the worst-case memory performance has already been discussed for codes with regular access patterns in Fraguela et al. [2010]. A discussion on the safeness of the components of the model associated to the

processing of irregular access patterns must be added to assure the safeness of this extension. The discussion in Fraguela et al. [2010] is based on the premises that the worst-case number of misses can be obtained by the PME model provided that the number of different SOLs that a reference R can access during the execution of a loop and the RD associated to each access are maximized. These two premises affect the construction of the PMEs. Section 5.1 removes any uncertainty about the fulfillment of these two premises for references following irregular access patterns, as Equation (12) models that every one of the N_i iterations of the loop will give place to the access to a different SOL, which also forces the RDs for the reuses of those SOLs to be maximal.

A third premise of the discussion in Fraguela et al. [2010] was that the proposed method maximizes the miss rate associated to the regions accessed during a given RD. The method proposed in 5.2 is safe because, in the first place, the AV representing the worst-case placement of the regular access pattern on the cache is calculated. This procedure follows a safe method already introduced in Fraguela et al. [2010]. The only modifications are that the worst-case overlapping is moved after the union operation, and that the results of the overlapping procedure are represented using the format of the $nlinesInterf$ data structure instead of an AV. These two modifications do not affect the safeness of the prediction.

This way, the only possible source of unsafeness is the treatment of the irregular access patterns in the worst-case miss rate calculation process. The placement in the cache of the lines brought by irregular accesses is totally unpredictable due to the lack of any pattern. It can only be assured that, given a region on which an irregular access is performed, at most $nlines$ are accessed and, at most $\lceil l \rceil$ lines may be placed in the same cache set. The method used to calculate these two values is safe. The algorithm in Figure 8 maps the $nlines$ lines of each irregular access pattern to the sets represented in $nlinesInterf$ in such a way that the maximum number of cache sets receives enough lines to generate a miss when there is a reuse attempt in those sets. Then, the algorithm in Figure 9, executed after the algorithms in Figures 7 and 8 have filled $nlinesInterf$ with the appropriate values, matches those sets with the largest numbers of lines of the reuse region with those sets with the largest number of interfering lines. This gives place to a maximization of the number of lines of the reuse region that cannot be reused in cache, and whose access results therefore in a miss. As a consequence, the PME model presented in this section is safe because the safeness of all its components have been discussed.

6. EXPERIMENTAL RESULTS

The model, which is integrated in a compiler framework [Andrade et al. 2007], has been applied automatically to ten regular codes and six irregular codes and its predictions have been validated against a trace-driven simulator. The code of the applications is both modeled and simulated at the source code level. The only optimization assumed (both in the modeling and the simulation) is that the compiler will permanently store the scalars used in processor registers; therefore only array references are modeled and simulated. This is not a limitation of the model, but an assumption on a minimal optimization typically applied by the compiler that is made for the experiments in the article. Anyway, all the nonarray references could be easily modeled as references to arrays of a single element that have stride zero with respect to any enclosing loop.

The regular codes used in the experiments have been: 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 [Zivojnović et al. 1994]: convolution, fir, lms, matrix1 and n_real_updates. Pointer-based memory accesses have been replaced with equivalent array accesses,

Table II.
Characteristics of the caches used in the experiments. C_s is the cache size, L_s is the line size, K is the associativity, and Hit and Miss are the hit and miss time in cycles.

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

and functions were inlined. These codes have been gathered from similar works in the bibliography [Ramaprasad and Mueller 2005; Vera et al. 2007; White et al. 1997]. The six irregular codes used in the experiments have been: the sparse-matrix vector product (SPMXV) displayed in Figure 1, which uses the CRS format for the storage of the sparse matrix; the three possible orderings for the sparse matrix-dense matrix product (SPMXDM); the reordering of the elements of an array according to the indexes contained in an index array (REORDER), and the addition of two arrays, where the positions of both arrays to be added are selected from an index array (IRADD). Regarding the SPMXDM codes, the three capital letters after the name of each one of them (SPMXDMIKJ, SPMXDMIJK and SPMXDMJIK), indicate the ordering of the loops from the outermost one to the innermost one. The sparse matrix is stored in CRS format (see Example 3.1) and is thus accessed by rows. Loop `I` selects the different rows of the sparse matrix, loop `K` iterates on the elements inside each row, and loop `J` selects the different columns of the dense matrices.

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. In the case of SPMXV and SPMXDM codes, a 200×200 sparse matrix with 4000 non-zero values was used. The REORDER and IRADD codes used base arrays of 25000 elements, and an index array of 2500 elements. Each code was tested using the cache configurations of a MicroSPARC II-ep [Microelectronics 1997], a PowerPC 604e [Inc 1996], a MIPS R4000 [MIPS 2001] and a IDT79RC64574 [IDT 2001], which have been used in [Vera et al. 2007] too. Table II summarizes the main characteristics of these caches, including the hit and cache miss cycles.

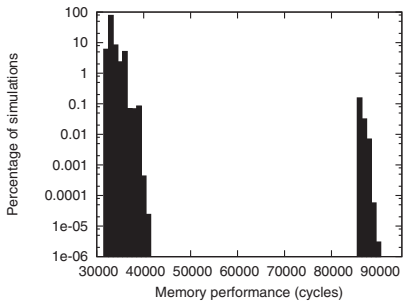
For the STENCIL, CNT, TRANS, FIBONACCI, convolution, fir, lms, matrix1 and REORDER codes, and each cache configuration, all the possible relative positions in the cache of the data structures that appear in the code were simulated systematically. The simulations were conducted using a highly optimized simulator extensively validated against the well-known trace-driven simulator dineroIII [Hill 1985]. It was not possible to simulate all the relative positions for ST, n_real_updates, SPMXV, SPMXDMIKJ, SPMXDMIJK, SPMXDMJIK and IRADD because the number of combinations increases exponentially with the number of the data structures in the code, and these ones are the codes with more than three data structures. Thus, for these codes a subset of random base address combinations was simulated for 3200 hours in a 1.6 Ghz Itanium Montvale processor.

The index arrays of the irregular codes were filled in with the worst-case values. In the IRADD and REORDER codes, the values of the indirection were spanned as much as possible along the dimension indexed (Case (a) of Figure 3). For the SPMXV and the three SPMXDM orderings, we tested the usage of matrices with layouts similar to the ones shown in Cases (a) and (b) of Figure 3. The irregular access takes place on a vector in the SPMXV code and on a dense matrix in the SPMXDM codes. Case (b) matrix minimizes the number of lines accessed across the indirection in each whole

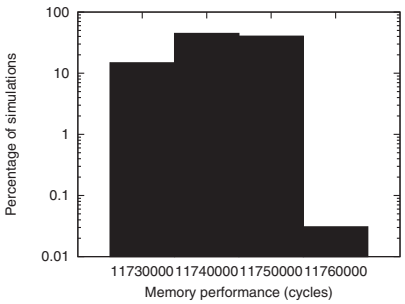
execution of loop K (loop J in SPMXV), which iterates on the elements inside each row. This way, the RD of loop I, which selects selects the different rows of the sparse matrix, is maximized, which avoids many reuses between different iterations of this loop. However, in SPMXDMIJK and SPMXDMIKJ, one iteration of loop I includes a complete execution of the loop J (that selects the different columns of the dense matrix), which does not happen in SPMXV and SPMXDMJIK. This inclusion increases the interference produced in each iteration of loop I, thus avoiding a large number of reuses with no need to minimize the number of lines accessed in each execution of loop K. This causes that Case (a) matrix gives place to the WCMP for the SPMXDMIKJ and SPMXDMIJK codes, while Case (b) matrix produces it in SPMXV and SPMXDMJIK.

Figure 10 shows the results of the simulations run using random varying base addresses for each one of the data structures for the codes with irregular access patterns and the MicroSPARC II-ep [Microelectronics 1997] cache (one-way 8KB cache with lines of 16B). Each graph represents, for each code, the percentage of the simulations run for that code for which the memory performance obtained, expressed in cycles, was within a given range. For example, a total of 20333308 simulations were run for the SPMXDMIKJ code, considering each simulation a different combination of the base addresses of the data structures used in this code. The first bar in Figure 10(b) shows the percentage of these simulations for which the memory performance obtained was between 11730000 and 11740000 cycles. Although these simulations did not cover all the base addresses combinations, as it was noted previously, they considered a wide set of them. Thus, the results shown in these figures are representative of the influence of the base addresses in the behavior of the different codes. The memory performance in each simulation 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 time and miss time of the studied cache, extracted from Table II. The range of cycles associated to each bar is adjusted in each code for a better display of the graphic. As Figure 10 shows, the number of cache misses varies largely along the simulations. Despite this large variability, it is often the case that many (sometimes most) simulations attain a similar memory performance. These simulations are thus concentrated in a bucket of the graph. The graphs use a logarithmic scale for the percentage of simulations so that it is easier to see the outliers. We can see that these outliers can present a much worse memory performance than the average. These results show that the base addresses of the data structures largely condition the memory performance, as the simulations only differ in their base addresses. The worst-case contents of the index arrays have been used in all the simulations. If these contents had been varied for each simulation, the variability of the cache performance would have even been larger. In all the codes but SPMXDMIKJ there are large differences between the best-case and the worst-case memory performance. The reason is the modification of the relative alignments of the data structures on the cache when the base addresses are changed. An extreme case is the presence of a number of fully aligned references. Since the cache is direct-mapped, this would lead to systematic interferences (and thus misses) whenever at least two of them are actually aligned with respect to the cache. In fact, the bars in all these codes are concentrated in a number of groups equal to the number of references that may be fully aligned. In SPMXDMIKJ, there are 2 potentially fully aligned references but the number of accesses produced by those references is a small percentage of the total number of accesses of the code, thus the impact of their full alignment is not very important. These results confirm the interest of a base-address independent prediction of the WCMP.

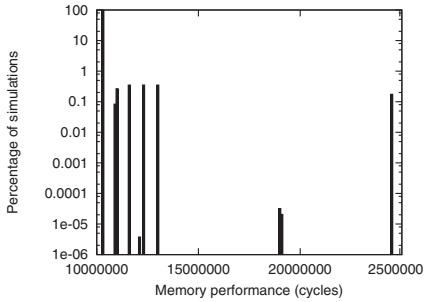
Figure 10 gave an idea of the influence of the base addresses on the cache performance. Now, Table III shows the influence of the contents of the index arrays. This table contains, for each irregular code, the number of accesses and the number of misses



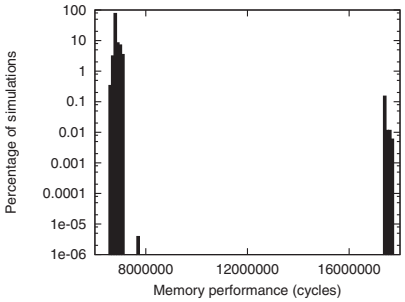
(a) SPMXV (each bar presents a percentage of simulations whose memory performance falls within a range of x=1000 cycles)



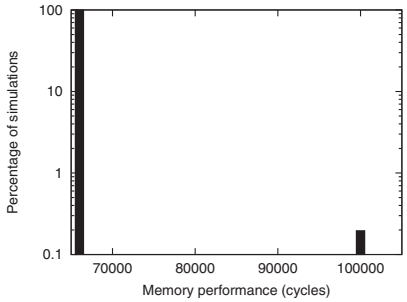
(b) SPMXDMIKJ (each bar presents a percentage of simulations whose memory performance falls within a range of x=10000 cycles)



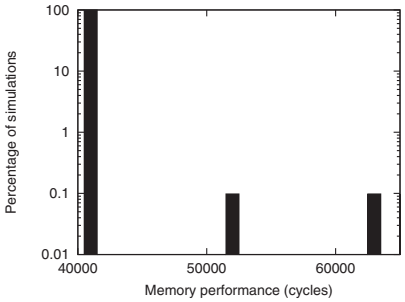
(c) SPMXDMIJK (each bar presents a percentage of simulations whose memory performance falls within a range of x=100000 cycles)



(d) SPMXDMJIK (each bar presents a percentage of simulations whose memory performance falls within a range of x=100000 cycles)



(e) IRADD (each bar presents a percentage of simulations whose memory performance falls within a range of x=1000 cycles)



(f) REORDER (each bar presents a percentage of simulations whose memory performance falls within a range of x=1000 cycles)

Fig. 10. Percentage of simulations where the memory performance is within a given range for the in the MicroSPARC II-ep cache (logarithmic scale in y axes).

it achieves in the caches described in Table II both considering the best-case and the worst-case contents of the index arrays. Regarding the base addresses, each data structure in the code is stored after the previous one, which is a very common situation. In the case of the IRADD and REORDER codes, the best-case contents of the index array is an ordered list of consecutive positions. It is assumed that one element cannot be accessed more than once across the indirection. In the case of SPMXV and the three SPMXDM, all the nonnulls are assumed to be stored consecutively in the first columns of each row. The results show the large influence the contents of the indirections have on the cache

Table III. Accesses and number of misses considering the best-case and worst-case contents of the index arrays for four different cache configurations.

		MicroSPARC II-ep		PowerPC 604e		MIPS R4000		IDT79RC64574	
Code	Accesses	BC_{misses}	WC_{misses}	BC_{misses}	WC_{misses}	BC_{misses}	WC_{misses}	BC_{misses}	WC_{misses}
SPMXV	12400	2123	2358	1032	1059	2131	2206	1055	1077
SPMXDMIKJ	1608202	745779	1122961	147659	886652	459223	965407	98217	854281
SPMXDMIJK	2560001	238728	861402	161026	841026	207746	851794	104097	841155
SPMXDMJIK	2560200	431801	469690	210800	215200	417585	441922	33226	55776
REORDER	7500	1875	3757	939	3027	1620	3407	939	2927
IRADD	10000	1875	3757	653	3126	1539	3703	673	3126

performance. Only the results for SPMXV show a modest variability. The reason is that in this code the index array is used to select different positions of the same vector across different iterations of its outermost loop (see Figure 1). Thus, the possibilities of reuse are always high and the influence of the contents of the index array are attenuated.

The rest of the section is organized as follows. Sections 6.1 and 6.2 contain the results obtained using the hard and the soft version of the model, respectively. Finally, Section 6.3 discusses the complexity of the model.

6.1. Results for the Soft Prediction

Table IV contains for each code and cache configuration, the average memory performance observed along the simulations expressed in cycles, $\overline{\text{MP}}$. $\Delta_{\text{WCMP}\%}$ is the difference between the soft WCMP predicted by the model and the actual WCMP observed along the simulations, expressed as a percentage of this latter value. The codes marked with a $*$ are those for which 3200 hours in a 1.6 Ghz Itanium Montvale processor did not suffice to simulate all the possible base address combinations. The nonnegativity of the $\Delta_{\text{WCMP}\%}$ column shows the validty of the soft prediction, while its small value shows its tightness for most codes. $\Delta_{\text{W}\overline{\text{MP}}\%}$ is the difference between the WCMP observed in the simulations and the average memory performance observed along the simulations, $\overline{\text{MP}}$, expressed as a percentage of this latter value. The large value of $\Delta_{\text{W}\overline{\text{MP}}\%}$ for some codes indicates that for these codes the WCMP observed is far from the average value observed in the simulations, sometimes up to 7 times larger. The factor that explains the large difference between the average and the worst-case memory performance for these codes is the existence of full alignments of more than K (K being the associativity of the cache) references, which causes systematic cache misses. In the TRANS code the soft predictions of the WCMP are not very tight. The reason for this lack of tightness is that the overlapping adjustments described in Section 4.3 consider worst-case overlappings that do not take place for the data size and cache configurations used in our experiments.

Programmers can avoid explicitly full alignments by using buffering or extra padding. In that case the model can provide a soft WCMP prediction closer to the average behavior by disabling the modeling of the wort-case overlapping. The results obtained using this new soft prediction of the WCMP are summarized in Table V, which only contains thus the codes where full alignments may appear, and whose statistics are referred only to the simulated cases where full alignments did not occur finally. This way, $\overline{MP}$ is the average memory performance (expressed in cycles) observed in these simulations, and $\Delta_{\text{WCMP}\%}$ and $\Delta_{\text{W}\overline{\text{MP}}\%}$ are calculated considering only simulations without full alignments. The values of $\Delta_{\text{WCMP}\%}$ are similar to those in Table IV and the values of $\Delta_{\text{W}\overline{\text{MP}}\%}$ are fairly smaller, which indicates that the WCMP is now closer to the average observed. Table V also includes a column A_s, which is the percentage of

Table IV. $\overline{\text{MP}}$, $\Delta_{\text{WCMP\%}}$, and $\Delta_{\text{W}\overline{\text{MP}}\%}$ for Four Different Cache Configurations.

	MicroSPARC II-ep			PowerPC 604e		
Code	$\overline{\text{MP}}$	$\Delta_{\text{WCMP\%}}$	$\Delta_{\text{W}\overline{\text{MP}}\%}$	$\overline{\text{MP}}$	$\Delta_{\text{WCMP\%}}$	$\Delta_{\text{W}\overline{\text{MP}}\%}$
ST$_*$	8256	0.00%	202.80%	14155	0.00%	571.14%
STENCIL	4286	0.00%	183.11%	6699	0.00%	0.55%
CNT	812500	0.96%	0.96%	1406250	1.97%	1.97%
TRANS	2045300	0.12%	0.57%	6151565	17.64%	17.74%
FIBONACCI	1625	0.55%	0.55%	2794	2.65%	2.65%
convolution	3276	0.00%	205.19%	5689	0.00%	0.81%
fir	4276	0.00%	183.52%	6689	0.00%	0.69%
lms	8244	0.00%	203.22%	14386	0.76%	1.74%
matrix1	44290990	2.11%	4.07%	61416512	2.42%	2.42%
n_real_updates$_*$	6618	0.00%	202.20%	11365	4.53%	5.17%
SPMXV$_*$	33820	3.66%	176.67%	52286	0.00%	0.85%
SPMXDMIKJ$_*$	11746972	12.32%	12.54%	31294745	3.86%	4.61%
SPMXDMIJK$_*$	10362317	0.00%	136.83%	33677963	0.02%	0.02%
SPMXDMJIK$_*$	6906561	4.46%	168.48%	10522600	12.04%	12.06%
REORDER	41394	0.00%	81.19%	123162	2.76%	2.79%
IRADD$_*$	66466	0.00%	50.45%	218162	0.03%	0.05%
	MIPS R4000			IDT79RC64574		
Code	$\overline{\text{MP}}$	$\Delta_{\text{WCMP\%}}$	$\Delta_{\text{W}\overline{\text{MP}}\%}$	$\overline{\text{MP}}$	$\Delta_{\text{WCMP\%}}$	$\Delta_{\text{W}\overline{\text{MP}}\%}$
ST$_*$	27159	0.00%	268.19%	7225	0.00%	453.58%
STENCIL	11867	0.00%	286.90%	3905	0.38%	0.77%
CNT	2687500	0.99%	0.99%	718750	1.57%	1.57%
TRANS	6292982	0.03%	0.91%	1762070	17.37%	17.83%
FIBONACCI	5375	0.73%	0.73%	1430	2.10%	2.10%
convolution	10836	0.00%	269.11%	2901	0.51%	1.16%
fir	11836	0.00%	287.56%	3901	0.38%	0.87%
lms	27280	0.00%	266.56%	7318	0.00%	446.54%
matrix1	107316817	2.83%	7.03%	39195460	2.69%	5.64%
n_real_updates$_*$	22045	0.00%	262.88%	5799	0.00%	451.77%
SPMXV$_*$	100891	4.38%	254.86%	28915	0.00%	17.59%
SPMXDMIKJ$_*$	39420691	8.78%	9.32%	12494302	12.53%	13.93%
SPMXDMIJK$_*$	35895329	0.00%	172.39%	15131531	0.24%	1.63%
SPMXDMJIK$_*$	20073189	4.52%	253.62%	3342135	0.00%	8.16%
REORDER	152658	0.00%	96.52%	54270	0.00%	0.50%
IRADD$_*$	254218	0.00%	57.35%	94390	0.00%	0.22%

all the simulations where full alignments appeared. The values of this column are low, which denotes that the existence of full alignments is unprobable.

6.2. Results for the Hard Prediction

The method used to calculate a hard prediction of the WCMP presented in Section 5 cannot be evaluated against a simulation in a reasonable time. Section 5.1 established that when a reference R is indexed through an indirection controlled by the variable of loop i, there are $(D_{Ri})^{N_i}$ possible sequence of accesses, and conversely, the same number of possible contents of the index array. In this expression D_{Ri} is the number of different SOLs that R could potentially access during the execution of loop i and N_i is the number of iterations of that loop. For each one of the $(D_{Ri})^{N_i}$ possible contents of the index array, a simulation using each possible relative position of the data structures

Table V.
$\overline{MP}$, $\Delta_{WCMP\%}$, $\Delta_{W\overline{MP}\%}$ and A_s(A_t) for four different cache configurations when excluding full alignments in the codes where they may appear.

	MicroSPARC II-ep				PowerPC 604e			
Code	$\overline{MP}$	$\Delta_{WCMP\%}$	$\Delta_{W\overline{MP}\%}$	A_s	$\overline{MP}$	$\Delta_{WCMP\%}$	$\Delta_{W\overline{MP}\%}$	A_s
ST$_*$	8125	0.98%	2.55%	1.93	14155	1.49%	6.80%	0.0
STENCIL	4268	0.21%	0.42%	0.29	6691	0.00%	0.66%	0.0
convolution	3262	0.55%	0.72%	0.29	5680	0.00%	0.98%	0.0
fir	4264	0.42%	0.51%	0.29	6691	0.00%	0.66%	0.0
lms	8190	0.99%	0.74%	0.48	14432	0.76%	1.41%	0.0
n_real_updates$_*$	6500	2.08%	2.08%	1.16	11324	4.53%	5.55%	0.0
SPMXV$_*$	100657	0.00%	23.17%	0.10	28915	0.00%	17.59%	0.0
SPMXDMIKJ$_*$	11746972	12.31%	12.53%	0.0	31294745	3.86%	4.61%	0.0
SPMXDMIJK$_*$	10337504	5.26%	33.15%	0.17	33677963	0.02%	0.02%	0.0
SPMXDMJIK$_*$	6887283	0.35%	13.03%	0.18	10522600	12.04%	12.06%	0.0
REORDER	41322	0.35%	0.39%	0.29	123162	2.76%	2.79%	0.0
IRADD$_*$	66385	0.08%	0.12%	0.18	218162	0.03%	0.05%	0.0
	MIPS R4000				IDT79RC64574			
Code	$\overline{MP}$	$\Delta_{WCMP\%}$	$\Delta_{W\overline{MP}\%}$	A_s	$\overline{MP}$	$\Delta_{WCMP\%}$	$\Delta_{W\overline{MP}\%}$	A_s
ST$_*$	26875	0.14%	2.03%	0.97	7225	1.63%	3.74%	0.0030
STENCIL	11828	0.00%	0.33%	0.19	3905	0.38%	0.77%	0.0
convolution	10804	0.36%	0.58%	0.14	2900	0.51%	1.19%	0.0
fir	11812	0.33%	0.46%	0.14	3908	0.38%	0.69%	0.0
lms	27148	0.58%	0.29%	0.24	7225	1.64%	2.70%	0.0
n_real_updates$_*$	21500	1.63%	1.63%	0.58	5780	3.37%	3.37%	0.0015
SPMXV$_*$	100657	0.00%	23.17%	0.10	28915	0.00%	17.59%	0.0
SPMXDMIKJ$_*$	39420691	8.48%	9.01%	0.0	12494302	12.53%	13.93%	0.0
SPMXDMIJK$_*$	35837063	6.32%	42.36%	0.0910	15131531	0.24%	1.63%	0.0
SPMXDMJIK$_*$	20029431	12.65%	21.19%	0.09	3342135	0.00%	8.16%	0.0
REORDER	152541	0.08%	1.02%	0.14	54270	0.00%	0.50%	0.0
IRADD$_*$	254023	0.05%	0.08%	0.09	94390	0.00%	0.22%	0.0

with respect to the cache should be executed to obtain the actual maximum of the number of misses valid for any base addresses of these structures. Since in a K-way cache of size C_s there are C_s/K relative positions with respect to the cache, this would involve a total of $(D_{Ri})^{N_i} \times (C_s/K)^X$ simulations, where X is the number of different data structures in the code.

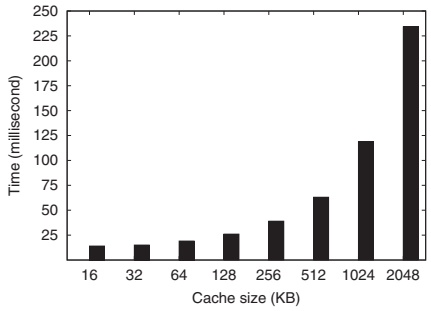
Table VI contains simply the worst-case memory performance predicted by the hard (safe) version of the model (WC_{mp}) and the miss rate associated (WC_{mr}), both when full alignments are avoided by the programmer (wo fa) or not (w fa). Only the codes with irregular access patterns are included in this table, as the predictions shown in Tables IV and V for regular codes are already covered by the safe wcPME model presented for them in Fraguela et al. [2010]. For direct-mapped caches (MicroSPARC II-ep and MIPS R4000) or if full alignments are possible, the worst-case miss rate predicted is close to 100%, thus, the effectivity of the cache during the execution of irregular codes for hard RTS in these circumstances is almost null. However, the usage of set associative caches (PowerPC 604e and IDT79RC64574) would be more useful for irregular codes and hard RTS, as safe lower miss rates are obtained.

6.3. Complexity of the Model

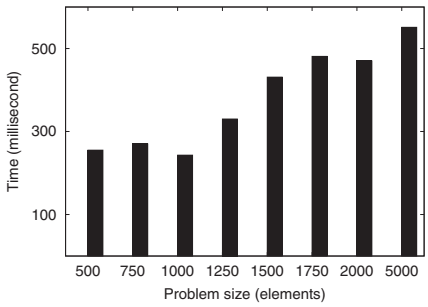
In all our experiments in the previous sections and the additional ones we present here to measure the computing demands of our model, the predictions of the model were

Table VI. WC_{mp} and WC_{mr} Predictions of the Safe Version of the PME Model

	MicroSPARC II-ep				PowerPC 604e			
	w fa		wo fa		w fa		wo fa	
Code	WC_{mp}	WC_{mr}	WC_{mp}	WC_{mr}	WC_{mp}	WC_{mr}	WC_{mp}	WC_{mr}
SPMXV	124000	100%	124000	100%	471200	100%	234400	48%
SPMXDMIKJ	16082020	100%	16082020	100%	45567902	74%	45567902	74%
SPMXDMIJK	24880001	97%	24880001	97%	94320001	97%	46960001	47%
SPMXDMJIK	24880200	97%	24880200	97%	92199730	95%	46960200	47%
REORDER	75000	100%	75000	100%	285000	100%	123236	42%
IRADD	100000	100%	100000	100%	380000	100%	218236	56%
	MIPS R4000				IDT79RC64574			
	w fa		wo fa		w fa		wo fa	
	WC_{mp}	WC_{mr}	WC_{mp}	WC_{mr}	WC_{mp}	WC_{mr}	WC_{mp}	WC_{mr}
SPMXV	496000	100%	496000	100%	198400	100%	102400	48%
SPMXDMIKJ	64328080	100%	64328080	100%	20117152	77%	20117152	77%
SPMXDMIJK	99280001	97%	99280001	97%	39760001	97%	20560001	47%
SPMXDMJIK	99280200	97%	99280200	97%	38776350	94%	19552500	44%
REORDER	300000	100%	300000	100%	120000	100%	54420	42%
IRADD	400000	100%	400000	100%	160000	100%	160000	100%



(a) SPMXDMIKJ considering different cache sizes



(b) SPMXDMIKJ considering different problem sizes

Fig. 11. Evolution of the modeling time

always generated in less than one second. Since these experiments were performed using a set of very diverse codes, problem sizes and cache configurations, the model times measured cover many situations and are thus very representative. The main factor that can increase the execution time of the model is the complexity of the code, namely, the number of references and the number of loops where they are embedded. This factor influences the number of PMEs that must be generated and the complexity and the number of times that the worst-case miss rate calculation process must be executed. The complexity of the model is secondarily influenced by the cache size, the associativity, the number of iterations of the loop and the sizes of the data structures. These factors increase the complexity of some of the algorithms in the miss rate calculation process such as the calculation of the AV associated to strided access patterns.

Figures 11(a) and 11(b) show the evolution of the modeling times of the soft version of the model for one of the most complicated codes, SPMXDMIKJ, for different cache and problem sizes, respectively. The model was executed in a machine with an Intel Core 2 Duo at 2.4 Ghz and 2 GB of RAM. Figure 11(a) models the behavior of the product (SPMXDMIKJ) of one dense 1000×1000 matrix and a 1000×1000 sparse

matrix with 40000 nonzeros. The cache has a line size of 32 bytes and associativity 2, while cache size varies from 16KB to 2048KB. The modeling time (expressed in milliseconds) is always below one second. Figure 11(b) models the behavior of the product (SPMXDMIKJ) of a $n \times n$ dense matrix and a $n \times n$ sparse matrix with $10 \times n$ nonzeroes. The problem size n takes values between 250 and 5000 and the cache size is the biggest one used in Figure 11(a). The modeling time is also always below one second. These results confirm that, although the modeling time depends on the cache size or the size of the data structures, the model is very fast and it provides its results in very short times.

7. RELATED WORK

None of the works in the bibliography models the behavior of codes with indirections or can provide a base-address independent predictions, thus, it has not been possible to compare the predictions of the wcPME model with other models.

Several researchers have used analytical methods to calculate the WCMP considering the caches of a memory hierarchy. The modeling of instruction caches [Alt et al. 1996; Healy et al. 1999] has had a lot of success, even in multicore systems with shared L2 caches [Yan and Zhang 2008]. There are also many works devoted to the study of data caches. White et al. [1997] bound-using a static analysis, the worst-case performance of set-associative instruction caches and direct-mapped data-caches. The analysis of data caches required 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 classified the accesses in one of four categories: always miss, always hit, first miss and first hit. The validation was performed considering only one cache configuration. In most programs, the prediction of the worst-case predicted was equal to the value observed. However, in some programs similar to ours like the dense matrix product (DMXDM) an overestimation of 10% for big data sizes occurred. A version of our ST program presented also an overestimation of 17% in their paper.

Lundqvist and Stenström [1999] 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, were considered to be predictable. When the cache is bypassed, the behavior is equivalent to a miss rate of 100% 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). In the validation, they showed statistics about the predictability of the references but they did not present any result of WCMP calculations.

Ramaprasad and Mueller [2005] used the cache miss equations (CMEs) [Ghosh et al. 1999], which need the data addresses for their predictions, as a basis for the WCMP estimation. Nonperfectly nested and nonrectangular loops are covered using loop transformations like the forced loop fusion that involves the insertion of loop index-dependent conditionals in the code. Loop index-dependent conditionals are modeled using an extra analysis stage. The validation showed almost perfect predictions of the WCMP but only two (direct-mapped) cache configurations were considered.

Vera et al. [2007] used 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, thus becoming predictable the cache behavior of those pieces of code outside the scope of application of the CMEs. Good predictions

of the WCMP were achieved for codes that use the cache locking in order to improve the WCMP predictability. The problem with the cache locking technique arises when the data involved in the irregular access exceeds largely the cache capacity, as most accesses result in cache misses, or when some portions of data structures involved in that access, which are preloaded in the cache, are never accessed. Besides, cache locking incurs the overhead of preloading the cache, even with portions of data structures that will not be actually used. Thus, this technique is only suitable for computations that deal with a small amount of data that fit in the cache.

8. CONCLUSIONS

This article presents an analytical model capable of predicting very fast the WCMP for codes with irregular access patterns due to the existence of indirections. No previous work has tackled the prediction of the WCMP in the presence of these patterns, resorting instead to locking or bypassing the cache, which can degrade substantially the performance of the memory hierarchy. The model is very general and flexible, as it does not need the concrete values of the indirections, which are in fact usually unavailable for the analysis; rather it considers the worst possible distribution of the indirections. A second unique feature of this model is that it does not require the base addresses of the data structures. Instead, it computes the overlapping of the data structures with respect to the cache that gives place to more misses. This allows to use the model in situations in which the data addresses are not available at compile time, and can in fact vary in different executions. This is the case for example in systems with physically-indexed caches or codes that use dynamically allocated memory. Another advantage of our model is that it can predict the miss rate per individual reference. This enables its usage to choose whether the cache is bypassed or not in the presence of irregular access patterns, and, in the latter case, whether preloading the data and locking it in the cache would be beneficial. Finally, the model is able to provide two kinds of estimations: a soft one, suitable for soft or non-RTS, and a hard safe one, suitable for hard RTS, and whose safeness has been for-mally proved.

An extensive validation using trace-driven simulations for 16 codes and 4 real cache configurations has been performed. The results show that the cache performance can vary largely depending on the base addresses of the data structures, and that our model provides tight predictions of the worst-case cache performance even in the presence of irregular access patterns. The safeness of the soft prediction is not ensured, as it has been commented throughout the article. Still, the soft predictions of our model have not been exceeded by any of the simulations in our experiments, even when reasonable worst-case contents of the index arrays have been used. The hard model predicts miss rates close to 100% for the irregular codes when direct-mapped caches are considered and when full alignments are possible. However, reasonable lower miss rates are obtained for set-associative caches and when full alignments are explicitly avoided by the programmer. These experiments were conducted only for the irregular codes as the soft model is already safe for the regular ones.

As future work, we will consider the possibility of using as optional input the base addresses of the data structures involved in the code whenever they are available, in order to provide a tighter WCMP calculation.

ACKNOWLEDGMENTS

We would like to thank the editor and the reviewers for their valuable comments that greatly helped to improve this article. We also want to acknowledge the Centro de Supercomputación de Galicia (CESGA) for the use of its computers.

REFERENCES

ALT, M., FERDINAND, C., MARTIN, F., AND WILHELM, R. 1996. Cache behavior prediction by abstract interpretation. In *Proceedings of the 3rd International Symposium on Static Analysis (SAS '96)*. Springer, 52–66.

ANDRADE, D., ARENAZ, M., FRAGUELA, B. B., TOURIÑO, J., AND DOALLO, R. 2007a. Automated and accurate cache behavior analysis for codes with irregular access patterns. *Concur. Comput. Pract. Exper. 19*, 18, 2407–2423.

ANDRADE, D., FRAGUELA, B. B., AND DOALLO, R. 2006. Analytical modeling of codes with arbitrary data-dependent conditional structures. *J. Syst. Archit. 52*, 394–410.

ANDRADE, D., FRAGUELA, B. B., AND DOALLO, R. 2007b. Precise automatable analytical modeling of the cache behavior of codes with indirections. *ACM Trans. Archit. Code Optim. 4*, 3, 16.

ANDRADE, D., FRAGUELA, B. B., AND DOALLO, R. 2009. Static prediction of worst-case data cache performance in the absence of base address information. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. 45–54.

BARRET, R., BERRY, M., CHAN, T., DEMMEL, J., DONATO, J., DONGARRA, J., EIJKHOUT, V., POZO, R., ROMINE, C., AND VAN DER VORST, H. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM Press.

DAVID, L. AND PUAUT, I. 2004. Static determination of probabilistic execution times. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS '04)*. 223–230.

FRAGUELA, B. B., ANDRADE, D., AND DOALLO, R. 2010. Address-independent estimation of the worst-case memory performance. *IEEE Trans. Ind. Inf. 6*, 4, 664–677.

FRAGUELA, B. B., DOALLO, R., AND ZAPATA, E. L. 2003. Probabilistic miss equations: Evaluating memory hierarchy performance. *IEEE Trans. Comput. 52*, 3, 321–336.

GHOSH, S., MARTONOSI, M., AND MALIK, S. 1999. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst. 21*, 4, 703–746.

HEALY, C. A., ARNOLD , R. D., MUELLER , F., WHALLEY, D. B., AND HARMON, M. G. 1999. Bounding pipeline and instruction cache performance. *IEEE Trans. Comput.48*, 1, 53–70.

HILL, M. 1985. Dineroiii documentation, unpublished unix-style. Tech. rep.

IBM. 1996. PowerPC 604e RISC Microprocessor Technical Summary. Tech. rep.

IDT. 2001. 79RC64574/RC64575 Data Sheet. Tech. rep. http://smartdata.usbid.com/datasheets/usbid/2000/2000-q1/rc64574_ma_75096.pdf

LUNDQVIST, T. AND STENSTRÖM, P. 1999. A method to improve the estimatedworst-case performance of data caching. In *Proceedings of the IEEE International Conference on Real-Time and Embedded Computing Systems and Applications*. 255–262.

MIPS. 2001. MIPS32 4Kp- Embedded, MIPS Processor Core. Tech. rep. www.mips.com.

RAMAPRASAD, H. AND MUELLER, F. 2005. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. 148–157.

RAMAPRASAD, H. AND MUELLER, F. 2006. Bounding preemption delay within data cache reference patterns for real-time tasks. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. 71–80.

SUN MICROELECTRONICS. 1997. MicroSPARC-IIep User's Manual. Tech. rep.

VERA, X., LISPER, B., AND XUE, J. 2007. Data cache locking for tight timing calculations. *ACM Trans. Embed. Comput. Syst*. 7, 1, 1–38.

WHITE, R., HEALY, C., WHALLEY, D., MUELLER, F., AND HARMON, M. 1997. Timing analysis for data caches and set-associative caches. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. 192–202.

XUE, J. AND VERA, X. 2004. Efficient and accurate analytical modeling of whole-program data cache behavior. *IEEE Trans. Comput. 53*, 5, 547–566.

YAN, J. AND ZHANG, W. 2008. WCET analysis for multi-core processors with shared l2 instruction caches. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. 80–89.

ZIVOJNOVIĆ, MARTINEZ, J., SCHLÄGER, C., AND MEYR, H. 1994. DSPSTONE: A DSP-oriented benchmarking methodology. In *Proceedings of theInternational Conference on Signal Processing Applications and Technology*.

Received June 2011; revised January 2012; accepted June 2012