CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2006; **00**:1–15 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Automated and accurate cache behavior analysis for codes with irregular access patterns

Diego Andrade*, Manuel Arenaz,
Basilio B. Fraguela, Juan Touriño and
Ramón Doallo

Computer Architecture Group, Department of Electronics and Systems
University of A Coruña, A Coruña, Spain
{*dcanosa,arenaz,basilio,juan,doallo*}*@udc.es*

SUMMARY

The memory hierarchy plays an essential role in the performance of current computers, thus good analysis tools that help predict and understand its behavior are required. Analytical modeling is the ideal base for such tools if its traditional limitations in accuracy and scope of application are overcome. While there has been extensive research on the modeling of codes with regular access patterns, less attention has been paid to codes with irregular patterns due to the increased difficulty to analyze them. Nevertheless, many important applications exhibit this kind of patterns, and their lack of locality make them more cache-demanding, which makes their study more relevant. The focus of this paper is the automation of the Probabilistic Miss Equations (PME) model, an analytical model of the cache behavior that provides fast and accurate predictions for codes with irregular access patterns. The paper defines the information requirements of the PME model and describes its integration in the XARK compiler, a research compiler oriented to automatic kernel recognition in scientific codes. We show how to exploit the powerful information-gathering capabilities provided by this compiler to allow automated modeling of loop-oriented scientific codes. Experimental results that validate the correctness of the automated PME model are also presented.

KEY WORDS: Memory hierarchy; cache behavior; performance prediction; irregular access patterns; chains of recurrences

*Correspondence to: Diego Andrade, Facultad de Informática, Campus de Elviña, s/n. 15071 A Coruña, Spain

Contract/grant sponsor: Ministry of Education and Science of Spain and FEDER funds of the European Union; contract/grant number: TIN2004-07797-C02
Contract/grant sponsor: Galician Government; contract/grant number: PGIDIT03TIC10502PR and PGIDT05PXIC10504PN

1. INTRODUCTION

The ever-growing gap between the memory speed and the processor speed is cushioned by means of memory hierarchies that combine different types of memory technologies. The performance of a program is highly dependent on the way it uses the cache memory. For this reason, different strategies that study the cache behavior have been proposed in the literature. Hardware counters [1] can measure the events related to cache behavior during the execution of the code. Their main limitations are the high computational cost and the lack of explanations about the observed behavior. Besides, they are not available or conveniently accessible in all architectures. Another well known approach is trace-driven simulation [15], where the code is executed to generate a trace that is used by a simulator to study the cache behavior.

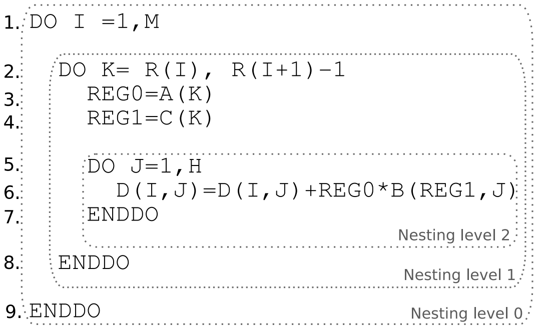
Analytical modeling [9, 10, 16] is an alternative method that avoids the execution of the program by building a model of the cache behavior from its source code. Analytical models enjoy limited accuracy due to the difficulty of finding a precise mathematical representation of the cache behavior whose calculation requires less time than the execution of the code. Due to the complexity of this modeling, most existing analytical models restrict themselves to the analysis of codes with regular access patterns and simple control flows. Our work is nevertheless capable of analyzing codes with irregular access patterns accurately and automatically, which is an important step forward to widen the applicability of analytical models. From now on, the existing model for regular access patterns will be referred as the original PME model.

The main contribution of this paper is the automation of the PME model for codes with irregular access patterns due to indirections [3] using the XARK compiler [4], an extensible framework for automatic kernel recognition that can be used as a powerful and efficient information-gathering tool [5]. The well-known formalism of chains of recurrences is used for the representation of the access patterns followed by the references in the code.

The paper is organized as follows. Section 2 presents a motivation example that will be used throughout the paper. Section 3 introduces chains of recurrences for the characterization of the access patterns. Section 4 describes the algorithm to build the PME model from the point of view of the information to be retrieved by the XARK compiler. Section 5 presents an extension of XARK that retrieves the information required by the model. Section 6 shows validation results. Finally, Section 7 discusses related work, and Section 8 concludes the paper.

2. MOTIVATING EXAMPLE

The original Probabilistic Miss Equations (PME) analytical model [9] was aimed to analyze codes with regular access patterns, for which it provides very accurate estimations of the cache behavior. Later extensions enabled handling codes with irregular access patterns due to the existence of conditional statements [2] and indirections [3]. The development of this model has been driven by a set of well-known codes that contain regular and irregular access patterns. A manual analysis of such codes revealed that the automation of the model from scratch is a difficult task, specially in the scope of irregular applications, as advanced symbolic analysis is needed to retrieve the necessary information.



```
DO I =1,M

  DO K= R(I), R(I+1)-1
    REG0=A(K)
    REG1=C(K)

    DO J=1,H
      D(I,J)=D(I,J)+REG0*B(REG1,J)
    ENDDO

  ENDDO

ENDDO
```

Figure 1: Sparse Matrix - Dense Matrix Product (IKJ)

For illustrative purposes, consider the computation of the product of a MxN sparse matrix in CRS format [6] [†] and a NxH dense matrix B shown in the code of Figure 1. The outermost loop $\mathtt{do}_\mathtt{I}$ presents array references with regular access patterns that can be rewritten as affine functions of the enclosing loop indices. For instance, the subscript of $\mathtt{R}(\mathtt{I}+\mathtt{1})$ takes increasing values in the interval $[2,\mathtt{M}+\mathtt{1}]$. Current commercial and research compilers can gather this information. However, irregular access patterns due to indirections require advanced symbolic analysis techniques. For example, reference $\mathtt{B}(\mathtt{REG1},\mathtt{J})$ follows an irregular access pattern because the values of REG1 are determined by $\mathtt{C}(\mathtt{K})$, whose values are not known at compile-time. Note that K introduces a higher level of indirection because it takes values in the interval $[\mathtt{R}(\mathtt{I}),\mathtt{R}(\mathtt{I}+\mathtt{1})-\mathtt{1}]$ in each $\mathtt{do}_\mathtt{I}$ iteration. Further analysis of the headers of $\mathtt{do}_\mathtt{I}$ and $\mathtt{do}_\mathtt{K}$ reveals that the code traverses the whole array of row indices of the sparse CRS matrix. The recognition of this programming construct, usually referred in the literature as *offset and length* [13], leads to conclude that K takes a strictly monotonically increasing set of values during the execution of $\mathtt{do}_\mathtt{I}$ and, thus, different elements of array C are referenced at run-time. The accuracy of the model would increase if the compiler could retrieve this information. The XARK compiler represents access patterns by means of the chains of recurrences formalism, which will be introduced in Section 3. From these chains of recurrences the PME model will build the equations that characterize the cache behavior for such access patterns. The corresponding algorithm will be described at high level in Section 4. The details about the recognition of programming constructs such as offset and length will be presented in Section 5.

[†] 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.

3. CHAINS OF RECURRENCES

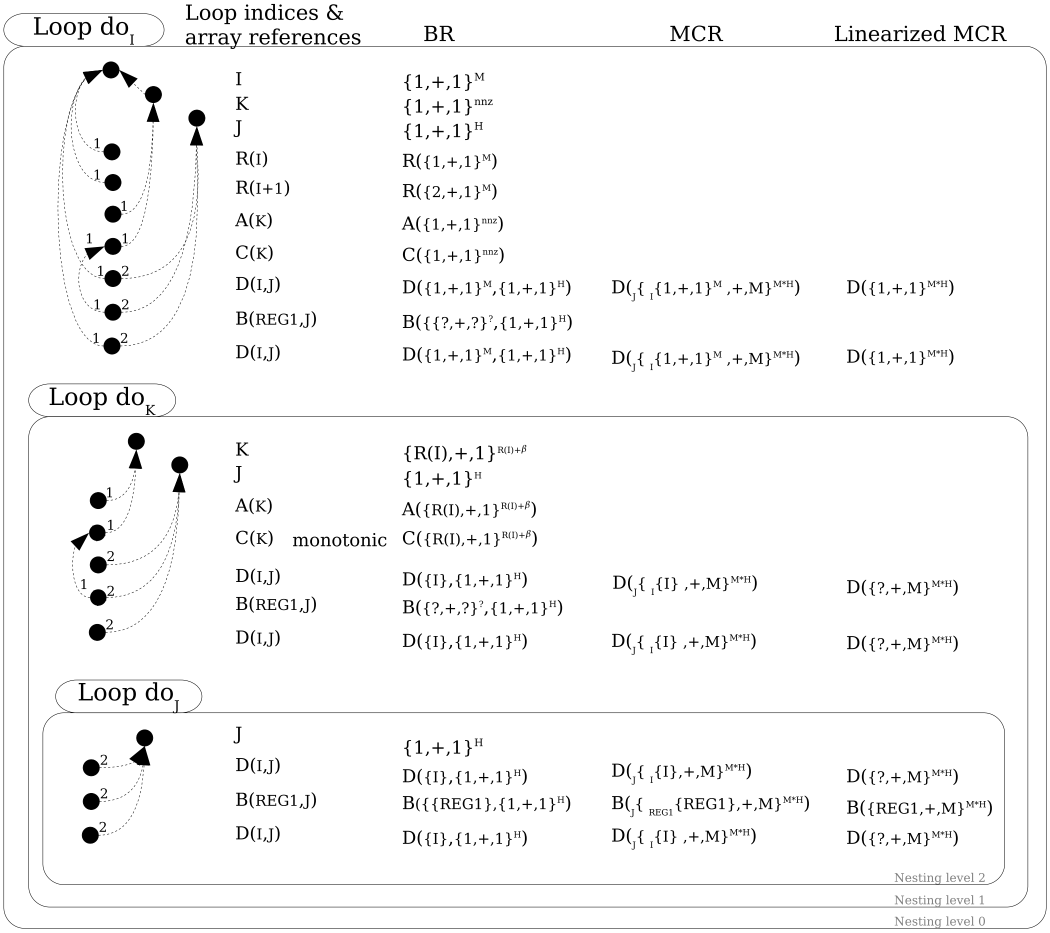
Chains of recurrences (CR) is a formalism to represent closed-form functions [17] that is used in different computer algebra systems, optimizing compilers and stand-alone C and Java libraries. Chains of recurrences have been successfully used to expedite function evaluation at a number of points in a regular interval. Given a constant ϕ_0, a function g defined over the natural numbers and zero, $\mathbb{N} \cup \{0\}$, and the operator $+$, a *Basic Recurrence (BR)* f, represented by the tuple $f = \{\phi_0, +, g\}$, is defined as a function over $\mathbb{N} \cup \{0\}$ by

$$\{\phi_0, +, g\}(i) \quad = \quad \phi_0 + \sum_{j=0}^{i-1} g(j) \;\text{ with }\; i \in \mathbb{N} \cup \{0\} \tag{1}$$

For example, the loop index of $\mathtt{do}_{\mathtt{I}}$ in Figure 1 takes integer values in the regular interval [1,M]. The BR $f = \{1, +, 1\}$ provides a closed-form function to compute the value of I at each $\mathtt{do}_{\mathtt{I}}$ iteration and thus to determine the affine memory access pattern $1 + I$ of array reference R(I). The algebraic properties of BRs provide rules for carrying out arithmetic operations with them [17]. For instance, the addition of a BR and a constant c is given by $\{\phi_0, +, g\} + c = \{\phi_0 + c, +, g\}$. This rule enables the representation of the access pattern of $\mathtt{R}(\mathtt{I} + \mathtt{1})$ as $\{1, +, 1\} + 1 = \{2, +, 1\}$.

Multidimensional Chains of Recurrences (MCR) [11] provide a formalism to describe memory access patterns of multidimensional arrays. In the following, an intuitive description of MCRs based on their interpretation is presented. Consider the bi-dimensional array reference $\mathtt{D}(\mathtt{I}, \mathtt{J})$ of Figure 1. In the scope of $\mathtt{do}_{\mathtt{I}}$, a row-major traversal of matrix D is performed, M and H being the number of rows and columns, respectively. As both rows and columns are accessed sequentially one after another, the BR $\{1, +, 1\}$ captures the access pattern defined by the subscript expressions I and J. However, from the point of view of the cache behavior, the description of the access pattern of the multidimensional array mapped onto a linear memory model is required. Assuming column-major storage, the MCR ${}_{\mathtt{J}}\{{}_{\mathtt{I}}\{1, +, 1\}, +, \mathtt{M}\}$, composed of two nested BRs, provides such information as follows. First, the inner BR ${}_{\mathtt{I}}\{1, +, 1\}$ is evaluated according to equation (1) in order to locate the beginning of row number I. Next, the outermost BR ${}_{\mathtt{J}}\{\mathtt{I}, +, \mathtt{M}\}$ is evaluated to access the row elements stored in memory locations with stride M. Within MCRs, the subscript on the left of each BR indicates the source code variable used to evaluate the BR. In this work only BRs and MCRs with constant g function are used as they enable the representation of the access patterns handled by the PME model. Note that CRs provide a powerful representation that will capture more complex cases that are expected to appear in full-scale real applications, like triangular access patterns. Besides, chains of recurrences are a well-known and widely used formalism that has an extensive research associated to it which can be used in future extensions of our work.

Figure 2 summarizes the information requirements of the PME model for the code of Figure 1. For each loop, a graph of dependence relations (represented as use-def chains) between array references and loop indices is depicted. Use-def chains starting from array references are labeled with the array dimension where the target reference appears. BRs that capture loop index values and access patterns for each dimension of each array reference are shown. When enough information is available, multidimensional arrays are also annotated with MCRs and



Loop indices & array references	BR	MCR	Linearized MCR
Loop do_I			
I	$\{1,+,1\}^{M}$		
K	$\{1,+,1\}^{nnz}$		
J	$\{1,+,1\}^{H}$		
R(I)	$R(\{1,+,1\}^{M})$		
R(I+1)	$R(\{2,+,1\}^{M})$		
A(K)	$A(\{1,+,1\}^{nnz})$		
C(K)	$C(\{1,+,1\}^{nnz})$		
D(I,J)	$D(\{1,+,1\}^{M},\{1,+,1\}^{H})$	$D(_J\{\,_I\{1,+,1\}^{M},+,M\}^{M*H})$	$D(\{1,+,1\}^{M*H})$
B(REG1,J)	$B(\{\{?,+,?\}^{?},\{1,+,1\}^{H})$		
D(I,J)	$D(\{1,+,1\}^{M},\{1,+,1\}^{H})$	$D(_J\{\,_I\{1,+,1\}^{M},+,M\}^{M*H})$	$D(\{1,+,1\}^{M*H})$
Loop do_K			
K	$\{R(I),+,1\}^{R(I)+\beta}$		
J	$\{1,+,1\}^{H}$		
A(K)	$A(\{R(I),+,1\}^{R(I)+\beta})$		
C(K) monotonic	$C(\{R(I),+,1\}^{R(I)+\beta})$		
D(I,J)	$D(\{I\},\{1,+,1\}^{H})$	$D(_J\{\,_I\{I\},+,M\}^{M*H})$	$D(\{?,+,M\}^{M*H})$
B(REG1,J)	$B(\{?,+,?\}^{?},\{1,+,1\}^{H})$		
D(I,J)	$D(\{I\},\{1,+,1\}^{H})$	$D(_J\{\,_I\{I\},+,M\}^{M*H})$	$D(\{?,+,M\}^{M*H})$
Loop do_J			
J	$\{1,+,1\}^{H}$		
D(I,J)	$D(\{I\},\{1,+,1\}^{H})$	$D(_J\{\,_I\{I\},+,M\}^{M*H})$	$D(\{?,+,M\}^{M*H})$
B(REG1,J)	$B(\{\{REG1\},\{1,+,1\}^{H})$	$B(_J\{\,_{REG1}\{REG1\},+,M\}^{M*H})$	$B(\{REG1,+,M\}^{M*H})$
D(I,J)	$D(\{I\},\{1,+,1\}^{H})$	$D(_J\{\,_I\{I\},+,M\}^{M*H})$	$D(\{?,+,M\}^{M*H})$

Figure 2: Information requirements of the PME model for the code of Figure 1. The symbol *nnz* stands for the number of nonzeros of the sparse matrix, and β is the average number of iterations of $\mathtt{do}_K$

linearized MCRs. The superscript on the right of the BRs represents an average of the number of times that the chain of recurrences is evaluated. The notation ? within BRs reflects that the corresponding information cannot be determined to be a constant expression at compile-time.

4. INFORMATION REQUIREMENTS OF THE EXTENDED PME MODEL

The PME model [3] uses its probabilistic nature to build formulae that estimate accurately the number of misses from statistical information about access patterns. This section describes a high-level algorithm of the PME model as well as the information requirements of its implementation in a compiler. Section 4.1 focuses on the construction of the formulae of the model and Section 4.2 on the computation of the interference regions, that is, the memory regions accessed by each given reference during a period of the execution of the code.

4.1. CONSTRUCTING THE FORMULAE

The pseudo-code of Figure 3 gives an overview of the PME model. As shown in the top-level procedure *analyze_code*, the references that appear in each loop nest of the source code are studied one by one. Each reference R is analyzed in several scopes. At each nesting level, the procedure *number_of_misses* computes a formula that calculates the number of misses produced by that reference in that nesting level. This formula is expressed in terms of the formula of the immediately inner loop. A reference may exhibit different access patterns with respect to different loops. These access patterns are modeled by the following formulae: the *regular access PME* for regular patterns, the *monotonic irregular access PME* for irregular patterns that access a monotonic sequence of memory positions, and the *non-monotonic irregular access PME* for irregular patterns that cannot be predicted at compile-time. Procedure *number_of_misses* selects the appropriate formula by analyzing the BRs associated with each dimension of `R` as follows:

- The regular access PME is applied if the BR matches $\{\phi_0, +, g\}$ with constant function g.
- The monotonic irregular access PME is applied if (1) a BR characterizing one of the dimensions has a non-constant g, and (2) there is a path of use-def chains between `R` and the loop index of the current loop that contains at least another different array reference. The first step of this path must be a use-def chain with a target array reference whose values can be determined to be monotonic.
- Otherwise, the non-monotonic irregular access PME is selected.

As an example, consider the array reference `B(REG1, J)` in Figure 1. In the analysis of the innermost loop $\texttt{do}_\texttt{J}$, the BRs that describe every dimension of the reference are explored. As shown in Figure 2, the BR $\{REG1\}$, simplified representation of $\{REG1, +, 0\}$, that describes the access pattern in the first dimension, is an invariant BR. In addition, as the BR $\{1, +, 1\}$ associated with the second dimension has a constant function $g = 1$, the subscript is known to be an affine function of `J`. Thus, a regular access PME models the behavior of the reference in this loop.

A different situation arises in the scope of $\texttt{do}_\texttt{K}$ at nesting level one. The BR for the first dimension has unknown ϕ_0 and g, which is represented as $\{?, +, ?\}$ in Figure 2. Besides, the graph of dependence relations depicted in Figure 2 shows that there is a path from the first dimension of `B(REG1, J)` to loop index `K` that contains another array reference `C(K)` whose values are stored in the scalar `REG1` (see lines 2, 4 and 6 of Figure 1). Thus, the subscript `REG1` is

```
procedure $analyze\_code()$ {
   foreach $loop\_nest$ of the code {
      foreach $reference$ in the $loop\_nest$ {
         $misses += number\_of\_misses(reference, outermost\_loop(loop\_nest), R_{full})$
      }
   }
}
procedure $number\_of\_misses(reference, loop, region)$ {
  if $is\_regular(reference, loop)$ {
     return $regular\_access\_PME(reference, loop, region)$
  } else {
     if $is\_monotonic(reference, loop)$ {
        return $irregular\_monotonic\_access\_PME(reference, loop, region)$
     } else {
        return $irregular\_nonmonotonic\_access\_PME(reference, loop, region)$
     }
  }
}
procedure $irregular\_monotonic\_access\_PME(reference, loop, region)$ {
  if $is\_innermost\_loop\_containing(loop, reference)$ {
     return $L_{Ri} * miss\_probability(region) + (Ni - L_{Ri}) * miss\_probability(interference\_region(loop, 1))$
  } else {
     misses=0.0
      foreach $inner\_loop$ in $inner\_loops\_containing(loop, reference)$ {
         $misses += L_{Ri} * number\_of\_misses(reference, inner\_loop, region)$
                 $+(Ni - L_{Ri}) * number\_of\_misses(reference, inner\_loop, interference\_region(loop, 1))$
      }
     return misses
   }
}
procedure $interference\_region(loop, num\_iterations)$ {
  $region\_set = \emptyset$
   foreach $reference$ in $loop$
     if $reference$ has one dimension {
        case $BR$ from $reference$ in $num\_iterations$ of $loop$ {
           $\{\_, +, 1\}^M$ : $R = R_s(M)$ // M consecutive elements. The wildcard _ can take any value
           $\{\_, +, N\}^M$ : $R = R_r(\frac{M}{N}, 1, N)$ // $\frac{M}{N}$ groups of 1 element separated by a distance N
           ...
        }
     } else {
         case $MCR$ from $reference$ in $num\_iterations$ of $loop$ {
            $\{\{\_, +, 1\}^N, +, M\}^P$ : return $R = R_r(\frac{P}{M}, N, M)$ //$\frac{P}{M}$ groups of N elements separated by a distance M
            $\{\{\_\}, +, 1\}^N$ : return $R = R_s(N)$ // N consecutive elements
            ...
         }
      }
      $region\_set = region\_set \cup R$
   }
   return $region\_set$
}
```

Figure 3: The PME model algorithm

known to be irregular. The accuracy of the prediction can be raised by taking advantage of the knowledge that `C` is the column array of a sparse CRS matrix since, assuming that the column indices are ordered within each matrix row, the sequence of values of `C(K)` is known to be monotonic. As a result, the monotonic irregular access PME is applied. Note that such information is not available in the scope of the outermost loop because `C(K)` is not monotonic across different iterations of $\texttt{do}_{\texttt{I}}$. In this case, the non-monotonic irregular access PME is used.

Two parameters are required to build a PME at nesting level i: N_i, the number of iterations of the loop and S_{Ri}, the stride between the elements that reference `R` accesses in two consecutive loop iterations. Using these values L_{Ri}, the number of loop iterations for which `R` cannot exploit any reuse, can be calculated. In our algorithm, N_i is the average number of times that the BR that characterizes the values of the loop index is evaluated. As for S_{Ri}, if there is not any dependence path between the reference and the loop index, $S_{Ri} = 0$. Otherwise, it is calculated as the product of the constant g of the BR associated with the loop index by the distance between two consecutive elements of the array referenced by R in the dimension indexed by the loop index. This latter value is calculated using the dimensions of the affected array and the mapping of the array into the linear memory model (i.e., row-major or column-major). Finally, L_{Ri} is calculated as a function of N_i and S_{Ri} (see the details in [3]).

In regular codes, N_i is usually available at compile time, and thus the average number of times that the BR of the loop index is evaluated can be computed (see the BR $\{1, +, 1\}^H$ of $\texttt{do}_{\texttt{J}}$ in Figure 2). However, this is not the case in irregular codes. Consider the loop index `K` of the offset and length construct of Figure 1. In the scope of $\texttt{do}_{\texttt{I}}$, `K` is used in `A(K)` and `C(K)` to access the whole sparse CRS matrix. Thus, N_i is the number of nonzeros nnz, as shown in the BR $\{1, +, 1\}^{nnz}$ of Figure 2. In contrast, in the scope of $\texttt{do}_{\texttt{K}}$, N_i is given by the symbolic expression $R(I+1) - R(I)$. In general, this expression takes a different value in each iteration of the outer loop $\texttt{do}_{\texttt{I}}$. However, from a statistical point of view, $N_i = \beta = \frac{nnz}{M}$ can be a good approximation for CRS sparse matrices with a uniform distribution of the entries, M being the number of rows of the sparse matrix. Thus, as $\texttt{do}_{\texttt{K}}$ traverses the elements of a row of the CRS matrix, the values taken by K could be represented by $\{R(I), +, 1\}^{R(I)+\beta}$. This situation also affects the calculation of the stride for the array reference `C(K)` in the scope of $\texttt{do}_{\texttt{I}}$. Loop index `I` affects `C(K)` through its dependence with the loop index `K`. As a result, the stride of `C(K)` with respect to loop $\texttt{do}_{\texttt{I}}$ will be the number of iterations of $\texttt{do}_{\texttt{K}}$ (i.e., β), because both loops define an offset and length construct.

4.2. COMPUTING THE INTERFERENCE REGIONS

The PME model estimates how different array references interfere in the use of the cache. For this purpose, the model maps the chains of recurrences that represent the access patterns of each reference in the scope of each enclosing loop into regions of a linear memory model. For instance, it considers the region $R_s(N)$, that represents the access to N consecutive elements of a data structure; and $R_r(N_r, T_r, L_r)$, that represents the access to N_r groups of T_r elements each separated by a distance L_r. Notice that both BR functions and region functions do not hold information about the order in which accesses take place. This order is in fact taken into account during the construction of the formulae (covered in Section 4.1) as they determine the reuse distances between two consecutive accesses to a same cache line. However, interference

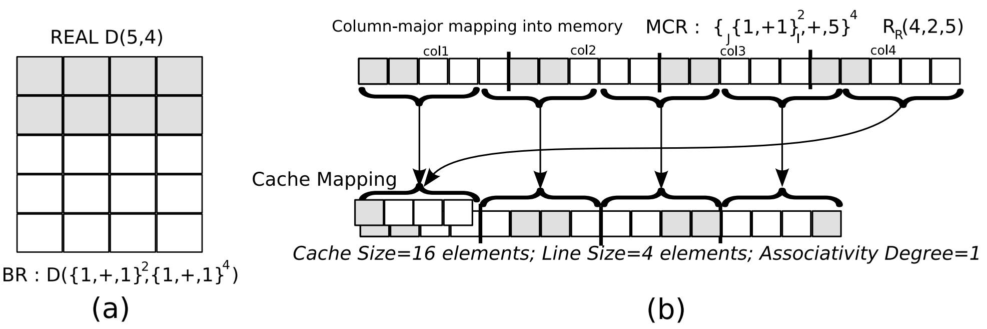

REAL D(5,4)
BR : D({1,+,1}^2,{1,+,1}^4)
(a)
Column-major mapping into memory
MCR : {_J{1,+1}_I^2,+,5}^4
R_R(4,2,5)
col1
col2
col3
col4
Cache Mapping
Cache Size=16 elements; Line Size=4 elements; Associativity Degree=1
(b)

Figure 4: Matrix mapping in memory and in cache for reference $\mathtt{D(I,J)}$ of Figure 1 during 2 iterations of loop $\mathtt{do_I}$

regions correspond to the cache footprint of the accesses that take place during a given reuse distance. This way, they do not depend on the order in which such accesses take place. Our model estimates the miss probability of the attempts of reuse from these footprints. Figure 4 shows an example memory region using the array reference $\mathtt{D(I,J)}$ of the loop $\mathtt{do_I}$ of Figure 1.

The information supplied by the BRs and the MCRs is used in the procedure *interference_region* (see Figure 3) to identify memory regions as follows:

- Let $\{\phi_0, +, g\}^\Gamma$ be the BR of a unidimensional array reference. If $g = 1$, a region $R_s(\Gamma)$ is computed. Otherwise a region $R_r(\frac{\Gamma}{g}, 1, g)$ is associated with the array reference.
- In the case of multi-dimensional arrays, the analysis focuses on the MCR that represents the access pattern once the array has been mapped onto the linear memory model. For the sake of the explanations, consider the MCR $\{\{\phi_1, +, g_1\}^{\Gamma_1}, +, g_2\}^{\Gamma_2}$ of a bi-dimensional array reference, where ϕ_1, g_1 and Γ_1 are associated with the first array dimension, and g_2 and Γ_2 with the second dimension. In this case, a region $R_r(\frac{\Gamma_2}{g_2}, \frac{\Gamma_1}{g_1}, g_2)$ is computed. Sometimes a simplified representation of the access pattern described by the MCR can be obtained by linearizing the MCR. The resulting BR is processed as described for unidimensional arrays.

The PME model estimates the impact on the cache of a set of regions in terms of a probability of interference in the call to the function *miss_probability* in Figure 3. The miss probability estimation is based on the calculation of the distribution of the number of cache lines of the footprints per cache set. The details about this process can be found in [2]. In the example code of Figure 1, during the analysis of the reference $\mathtt{D(I,J)}$ in the scope of the loop $\mathtt{do_J}$, the BR for the first dimension $\{\mathtt{I}\}$ indicates that the index is a loop invariant, while that of the second dimension $\{1, +, 1\}^{\mathtt{H}}$ shows that the subscript $\mathtt{J}$ takes consecutive values in the regular interval $[1, \mathtt{H}]$. As shown in Figure 2, the MCR $_{\mathtt{J}}\{_{\mathtt{I}}\{\mathtt{I}\}, +, \mathtt{M}\}^{\mathtt{M*H}}$ of $\mathtt{D(I,J)}$ can be linearized as the BR $\{?, +, M\}^{M*H}$, the unknown ϕ_0 indicating that $\mathtt{do_J}$ is analyzed in the scope of an undetermined $\mathtt{do_I}$ iteration. Applying the rule of unidimensional array references, the memory region $R_r(H, 1, M)$ of a row of array $\mathtt{D}$ is computed. When the access pattern for $\mathtt{D(I,J)}$ is analyzed in the next outer loop $\mathtt{do}_{\mathtt{K}}$, at nesting level 1, the BRs and MCRs for both dimensions are the same ones as in the innermost loop $\mathtt{do}_{\mathtt{J}}$ because none of the dimensions depend on loop index K. Thus, the same region $R_r(H, 1, M)$ is computed. A different situation arises in the scope of the outermost loop, where there is a different BR $\{1, +, 1\}^{\mathtt{M}}$ for the first dimension. As the M rows of the matrix are accessed, the linearized MCR $\{1, +, 1\}^{\mathtt{M*H}}$ contains a $\phi_0 = 1$ that reflects the access to the whole array D resulting in a region $R_s(M * H)$.

5. XARK EXTENSION FOR THE PME MODEL AUTOMATION

The automation of the PME model is addressed using the XARK compiler [4] as a powerful information-gathering framework. XARK operates on top of a high-level intermediate representation resembling the original source code that consists of the forest of abstract syntax trees (ASTs) that represent the statements of the Gated Single Assignment (GSA) form of the code. In an AST, a tree represents an operation so that the root node is the operator (e.g., assignment, scalar fetch, array reference, plus, product) and its children are the operands. The intermediate representation is completed with use-def chains that exhibit the dependences between the statements of the code. XARK performs a demand-driven analysis that proceeds as follows. A post-order traversal is carried out on each AST. At each node, a transfer function that gathers information about each operator in the program is applied once the analysis of the children subtrees has finished. When an occurrence of a variable defined in a different AST is found, the post-order traversal is deferred until the analysis of that AST is completed. This demand-driven behavior assures that all the information needed at a given node has been computed before the transfer function is actually executed.

Transfer functions are organized in layers devoted to specific tasks. The bottom layer addresses the recognition of the kernels computed in the source code (e.g., generalized induction variables, irregular reductions, array recurrences), which includes the characterization of the regular and irregular access patterns of the array references that appear in the source code. Upper layers implement extensions of the XARK compiler that benefit from the information recognized in the source code. Information interchange between layers is carried out by means of three containers that are available in all the transfer functions: *pgm* holds information at the program unit level; *stm* at the statement level; and *node* in the scope of a node of the AST of a statement. The pseudo-code of the extension that builds the interface between XARK and the PME model is shown in Figure 5. Due to space limitations, the details about the computation of the BRs and the MCRs have been omitted from the transfer functions. The containers are represented as data structures whose fields correspond to pieces of information retrieved from the source code.

In order to illustrate the operation of XARK, consider the forest of ASTs and the use-def chains (dashed arrows) depicted in Figure 6. The details about the GSA form have been omitted for the sake of clarity. The picture shows the last step of the post-order traversal of the AST that represents the loop header `DO K=R(I),R(I+1)-1`. Hatched nodes highlight expressions and statements whose analysis has already been completed. When transfer function T_{do} is applied, the kernel recognition layer characterizes `R(I)` and `R(I+1)-1` as loop-variant expressions whose value is not known at compile-time. This is denoted by the annotation

```
struct {...                           struct {...                           struct {...
    $graph\_of\_array\_refs$;              $set\_of\_array\_refs$;                $set\_of\_array\_refs$;
} pgm;                                } stm;                                } node;
```

```
 procedure $T_{a(s_1,...,s_n)}$ {        // Extensions of transfer function of array references
1    insert $a(s_1,...,s_n)$ in $pgm.graph\_of\_array\_refs$
2    foreach $s_i$ with subscripted access pattern {
3       foreach $reference \in node_{s_i}.set\_of\_array\_refs$ {
4          insert a use-def chain from $a(s_1,...,s_n)$ to $reference$ in $pgm.graph\_of\_array\_refs$
5       }
6    }
7    insert $a(s_1,...,s_n)$ in $node.set\_of\_array\_refs$
 }

 procedure $T_x$ {            // Extensions of transfer function of identifiers
1    if $x$ is not invariant {
2       foreach $reference \in set\_of\_array\_refs$ of the definition statement of $x$ {
3          insert $reference$ in $node.set\_of\_array\_refs$
4       }
5    }
 }

 procedure $T_{do}$ {         // Extensions of transfer function of loop headers
1    $stm.set\_of\_array\_refs = node_{init}.set\_of\_array\_refs \cup node_{limit}.set\_of\_array\_refs \cup node_{step}.set\_of\_array\_refs$
2    if $stm$ is an offset and length construct {
3       if $stm$ at nesting level 1 {
4          rewrite symbolic BR $\{R(I),+,1\}^{R(I+1)-1}$ as $\{1,+,1\}^{nnz}$
5       }
6    }
 }

 procedure $T_{stm}$ {            // Extensions of transfer function of assignment statements
1    $stm.set\_of\_array\_refs = node_{rhs}.set\_of\_array\_refs$
 }
```

Figure 5: Extension of XARK for building the interface with the PME model

`subscripted` in the corresponding nodes of the AST. Expressions corresponding to invariant and linear access patterns are annotated as `invariant` and `linear`, respectively. It is also attached to each node the BR that captures the interval in which the expression takes values, which is computed by applying the rules defined in the CR algebra [17] (see the example in Section 3). Next, the extension of T_{do} presented in Figure 5 is executed. First, the loop header `DO K=R(I),R(I+1)-1` is recognized as an offset and length construct because `R(I)` and `R(I+1)-1` are subscripted accesses to consecutive elements of a unique array `R`, and each expression is the source of a use-def chain whose target is the outermost loop $\mathtt{do_I}$. Under these conditions, T_{do} rewrites the BR $\{\mathtt{R(I)},+,1\}^{\mathtt{R(I+1)-1}}$ as $\{1,+,1\}^{\mathtt{nnz}}$ to indicate that the loop index `K` traverses the whole sparse matrix during the execution of $\mathtt{do_I}$ (see lines 2 to 6 of procedure T_{do} in Figure 5). The demand-driven analysis of the forest of ASTs continues, and

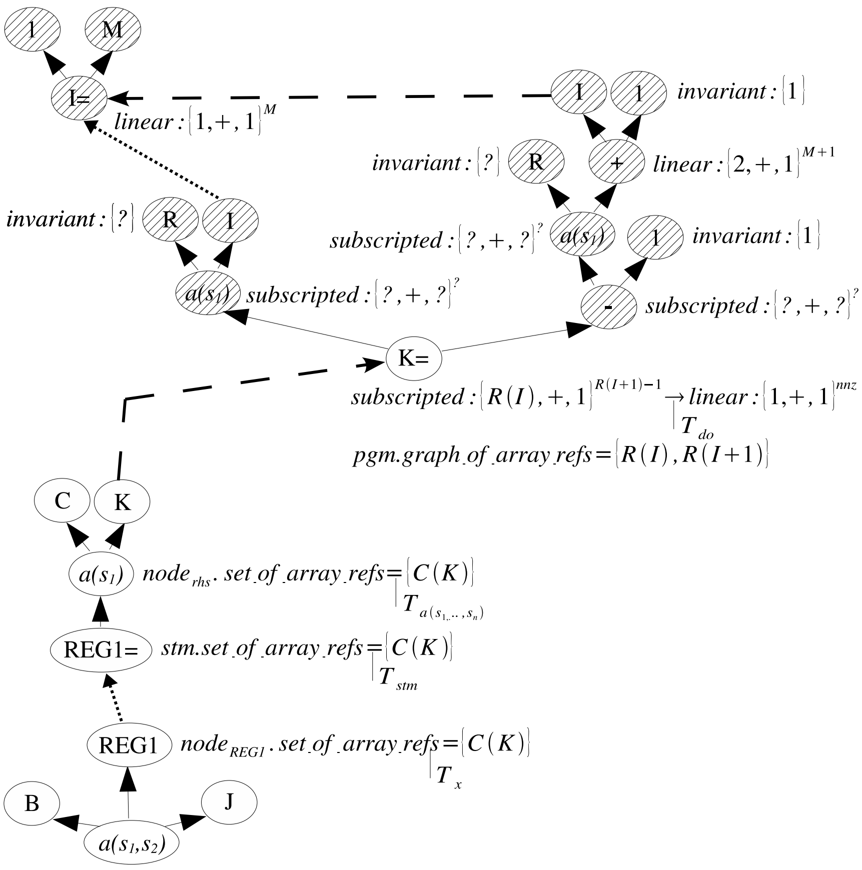

1
M
I=
$linear:\{1,+,1\}^{M}$
I
1
$invariant:\{1\}$
$invariant:\{?\}$
R
+
$linear:\{2,+,1\}^{M+1}$
$invariant:\{?\}$
R
I
$subscripted:\{?,+,?\}^{?}$
$a(s_1)$
1
$invariant:\{1\}$
$a(s_1)$
$subscripted:\{?,+,?\}^{?}$
-
$subscripted:\{?,+,?\}^{?}$
K=
$subscripted:\{R(I),+,1\}^{R(I+1)-1} \xrightarrow{} linear:\{1,+,1\}^{nnz}$
T_{do}
$pgm.graph\_of\_array\_refs=\{R(I),R(I+1)\}$
C
K
$a(s_1)$
$node_{rhs}.set\_of\_array\_refs=\{C(K)\}$
$T_{a(s_1,\ldots,s_n)}$
REG1=
$stm.set\_of\_array\_refs=\{C(K)\}$
T_{stm}
REG1
$node_{REG1}.set\_of\_array\_refs=\{C(K)\}$
T_x
B
J
$a(s_1,s_2)$

Figure 6: Forest of ASTs and use-def chains of the offset and length construct and the array reference B(REG1, J) of the example code of Figure 1.

the access pattern of array reference C(K) is characterized as a linear pattern given by the BR $\{1,+,1\}^{\texttt{nnz}}$.

5.1. CONSTRUCTION OF THE GRAPH OF REFERENCES

Apart from the characterization of the access patterns of array references, the interface between XARK and the PME model exhibits the dependence relationships between array references and loop indices. As shown in Section 4, such information is used to build the formulae that capture the cache behavior of the source code. The graph of dependences is built as follows. Each time the transfer function of array references $T_{a(s_1,\ldots,s_n)}$ is executed, the corresponding array reference is inserted in $pgm.graph\_of\_array\_refs$ (see line 1 of procedure $T_{a(s_1,...,s_n)}$ in Figure 5). Thus, when T_{do} is applied in Figure 6, $pgm.graph\_of\_array\_refs$ is $\{\mathtt{R(I)}, \mathtt{R(I+1)}\}$. As a result, a list of all array references in the source code has been built.

In order to construct the graph, it is necessary to identify indirections as well as the array references that appear in subscript expressions. This task is accomplished by taking advantage of the access pattern characterization provided by the kernel recognition layer. The demand-driven nature of XARK assures that the access pattern of each subscript s_l $(1 \leq l \leq n)$ has been characterized before the transfer function is applied. Thus, $T_{a(s_1,...,s_n)}$ recognizes array references that are not indirections by checking that there is not any subscripted access pattern, and inserts the reference in the container available for each node of the ASTs, in particular, in $node.set\_of\_array\_refs$ (line 7 of $T_{a(s_1,...,s_n)}$ in Figure 5). If an indirection is recognized, the demand-driven analysis carried out by XARK assures that $node_{s_j}.set\_of\_array\_refs$ contains the array references that appear in the subscript expression of the $\mathtt{j}-th$ array dimension. Next, $T_{a(s_1,...,s_n)}$ inserts in $pgm.graph\_of\_array\_refs$ a set of use-def chains whose source is $a(s_1, ..., s_n)$ and whose targets are the array references included in $node_{s_j}.set\_of\_array\_refs$ (lines 2-6 of $T_{a(s_1,...,s_n)}$ in Figure 5). Note that the sets of array references are transferred through scalar definition statements and loop headers. On the one hand, T_x transfers information from the container of the AST where x is defined (see lines 2-4 of T_x in Figure 5) to the local container node associated with the node where x is referenced. On the other hand, T_{stm} and T_{do} annotate the statements of the code with the list of array references that appear as operands of the right-hand side operators (see $node_{rhs}$, $node_{init}$, $node_{limit}$ and $node_{step}$ in Figure 5). As the ASTs are analyzed only once during the demand-driven analysis, the annotation of statements enables the retrieval of the set of array references for different occurrences of a scalar variable.

For illustrative purposes, consider the construction of the graph depicted in Figure 2 for the scope $\mathtt{do_K}$. In particular, focus on the subscript $\mathtt{REG1}$ of the first dimension of $\mathtt{B(REG1, J)}$. When the AST of $\mathtt{REG1 = C(K)}$ is analyzed, $T_{a(s_1,...,s_n)}$ inserts $\mathtt{C(K)}$ in $node_{rhs}.set\_of\_array\_refs$ and later T_{stm} annotates the statement by copying $\mathtt{C(K)}$ into $stm.set\_of\_array\_refs$. Next, the occurrence $\mathtt{REG1}$ in $\mathtt{B(REG1, J)}$ is processed by T_x, which obtains $\mathtt{C(K)}$ from the AST container of the statement where $\mathtt{REG1}$ is defined. As a result, the array reference $\mathtt{C(K)}$ is available at T_x, which copies $\mathtt{C(K)}$ in the local container $node_{REG1}.set\_of\_array\_refs$ to expose such information to $T_{a(s_1,...,s_n)}$. Finally, $T_{a(s_1,...,s_n)}$ updates the global container $pgm.graph\_of\_array\_refs$ with a use-def chain from $\mathtt{B(REG1, J)}$ to $\mathtt{C(K)}$.

6. EXPERIMENTAL RESULTS

The automation of the PME model within the XARK compiler was tested with a set of benchmarks that carry out operations with sparse CRS matrices: sparse matrix-dense matrix product with IKJ (shown in Figure 1), JIK and IJK loop orderings, sparse matrix-vector product, and sparse matrix transposition. The model was validated against a trace-driven simulation for a variety of sparse matrices and cache configurations.

The accuracy of the PME model is measured by the metric $\Delta_{MR}\%$, the absolute value of the difference between the miss rate obtained by a trace-driven simulation and the miss rate

Table I. $\overline{MRSim}\%$, $\overline{MRMod}\%$ and $\Delta_{MR}\%$ values obtained for the benchmarks performing more than 12800 tests for all the possible cache configurations combining cache sizes from 8KBytes to 1MByte, lines from 16 to 128 bytes and associativity degree from 1 to 8, using in these parameters only powers of two. All the cache configurations were tested with sizes per dimension of the involved arrays varying from 500 to 5000 with step 500 using only square matrices and sparse matrix densities from 0.5% to 25.5% with step 2.5%.

Code	$\overline{MRSim}\%$	$\overline{MRMod}\%$	$\Delta_{MR}\%$
SPMXV	9.64%	9.45%	0.92%
SPMXDMIKJ	48.95%	47.92%	1.41%
SPMXDMIJK	22.20%	21.42%	0.79%
SPMXDMJIK	11.68%	11.28%	0.70%
TRANSPOSE	18.98%	19.22%	1.60%

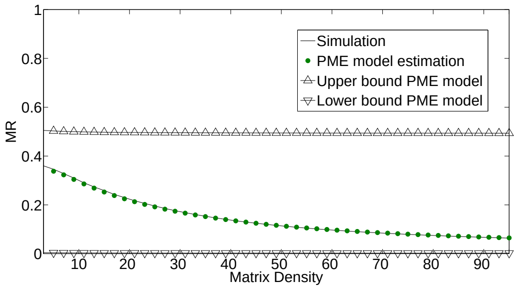


Figure 7: Miss rate as a function of the matrix density for the sparse matrix-dense matrix product with IJK ordering, where $M = N = H = 500$ in a cache of 64KB with line size 64 bytes and associativity degree 4

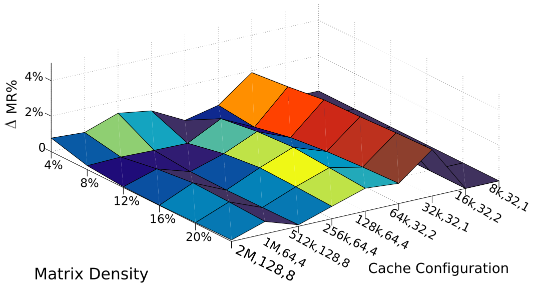


Figure 8: $\Delta_{MR}\%$ as a function of the matrix density and the cache configuration for the sparse matrix-dense matrix product (IKJ), for $M = N = H = 750$. Cache configuration is expressed as C_s,L_s,k where C_s is the cache size in bytes, L_s is the line size in bytes and k is the associativity degree

predicted by the model expressed as a percentage. The average values of $\Delta_{MR}\%$ presented in Table I show that the accuracy of the prediction is good for all the tests performed with the benchmarks. The prediction is a little worse for the sparse matrix transposition, the most complex code studied in this paper. Columns $\overline{MRSim}\%$ and $\overline{MRMod}\%$ contain the average values of the miss rate simulated by a trace-driven simulation and the miss rate predicted by the PME model in the set of experiments, respectively. They show that the relative error of our prediction is also small in relation to the measured miss rate.

Our prediction errors are smaller than those of the comparable works in this field. In Figure 7, the sparse matrix-dense matrix product with IJK loop ordering is used to compare the miss rate simulated and the miss rate predicted, an upper bound of the prediction considering all irregular accesses as misses in the PME model, and a lower bound obtained by ignoring the irregular accesses that appear in the code. The sizes of the data structures that appear in the code and the cache configuration were kept constant, while the density of the sparse matrix took values between 1% and 100%. Figure 7 shows that the PME model estimates the miss rate accurately, while the other two simple approaches provide poor estimations. This justifies the interest of our model.

Figure 8 shows the evolution of $\Delta_{MR}\%$ for the sparse matrix-dense matrix product with IKJ ordering, keeping the sizes of the data structures constant and changing not only the matrix density but also the cache configuration. Note that the deviation of the model is always less than 2.75%. For each code there is a cache configuration where the maximum error is achieved for a given input data set, because the relative positions of the involved arrays can have a big influence on the final number of misses. Larger caches diminish this effect, since they are big enough to avoid the overlapping of large pieces of the involved arrays. This effect is also attenuated in smaller caches because the involved array regions fully overlap in the cache, no matter which their relative positions are. Figure 8 shows that the tested code suffers from this effect for the 32K,32,1 cache configuration when the input data set size M=N=H=750 is used.

The accuracy of the model and its low computational cost, less than 1 second for each test (including XARK and PME model times), makes it suitable for driving compiler optimizations. As an example, we used our tool to predict at compile-time which is the optimal loop ordering for the sparse matrix-dense matrix product in terms of the lowest number of CPU idle cycles caused by cache misses in the memory hierarchy. The experiments were conducted both on a PowerPC 7447A and an Itanium 2 at 1.5Ghz using 7 different matrix configurations. Our model always predicted JIK as the optimal loop ordering (see Table I), which was confirmed by the execution time obtained by compiling the codes with `g77 3.4.3` and a `-O3` optimization level.

7. RELATED WORK

While there are several models that predict accurately the cache behavior for codes with regular access patterns in an automated way [9, 10, 16], only the PME model has achieved this purpose for codes with irregular access patterns. Some works that have faced this kind of models [12, 8] are not systematic enough to be automated. For example, [12] is an ad-hoc model whose scope of application is limited only to direct-mapped caches, and it does not consider the interaction between different interleaved access patterns. These limitations were overcome in our probabilistic model [8] which was not systematic enough to be automatable.

Other works [7, 14] model this kind of codes automatically, but their accuracy is low in many cases. Cascaval's indirect accesses model [7] is integrated in a compiler framework, but it is an inaccurate heuristic that estimates the number of cache lines accessed rather than the real number of misses. For example, it does not take into account the distribution of the irregular accesses and it does not account for conflict misses, since it assumes a fully-associative cache. The interface with the compiler framework is a simple dependence graph between references that access the same data position, the arcs being labeled with the number of different memory locations accessed between the source and the target of the dependence. This approach gives an ad-hoc representation that is not so extensible and manageable as the one used in the PME model, based on chains of recurrences. Another automated approach is SPLAT [14], a tool that analyzes codes in several phases. The reuse and volume phases, where compulsory and capacity misses are computed, respectively, considering a fully-associative cache; and the interference phase, where conflict misses are calculated assuming a direct-mapped cache. Irregular accesses due to conditional statements and loops with a variable number of iterations are modeled using the information derived from a previous code profiling, something that is not necessary in our model.

8. CONCLUSIONS

Codes with irregular access patterns due to indirections are difficult to analyze automatically and require advanced symbolic analysis techniques. This work addresses the automation of the modeling of the cache memory behavior for this kind of codes using the XARK compiler. Most previous analytical models for irregular codes were not automatable. The few automatable ones did not provide accurate estimations of the cache behavior. The model presented in this work is the first one that has been automated and that provides good degrees of accuracy in its predictions of the cache behavior of irregular codes. The paper has shown how to take advantage of the demand-driven nature of the XARK compiler in order to meet the hard information requirements of the PME model. An interface between XARK and the PME model has been defined using graphs of dependences between data structures and the chains of recurrences formalism for the representation of the access patterns. Our experiments have shown that XARK and the PME model can work together to obtain accurate cache behavior estimations. The experiments have also shown that the model can be used as a compile-time tool to guide code optimizations without any human aid, even in codes with indirections. As future work, the automated PME model will be evaluated for a wider set of representative irregular codes, and complex optimization processes will be driven using the information provided by XARK.

ACKNOWLEDGEMENT

We want to acknowledge the Galician Supercomputing Center (CESGA), Santiago de Compostela, Spain, for the use of its supercomputers to get experimental results on the Itanium 2 architecture.

REFERENCES

1. G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. *18th ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pages 85-96, Las Vegas, 1997.
2. D. Andrade, B.B. Fraguela, and R. Doallo. Analytical Modeling of Codes with Arbitrary Data-Dependent Conditional Structures. *Journal of Systems Architecture*, 52(7): 394-410, November 2006.
3. D. Andrade, B.B. Fraguela, and R. Doallo. Precise Automatable Analytical Modeling of the Cache Behavior of Codes with Indirections. *ACM Trans. on Architecture and Code Optimization* (Accepted for publication).

4. M. Arenaz, J. Touriño, and R. Doallo. A GSA-based Compiler Infrastructure to Extract Parallelism from Complex Loops. *17th ACM Int. Conf. on Supercomputing*, pages 193-204, San Francisco, June 2003.
5. M. Arenaz, J. Touriño, and R. Doallo. Compiler Support for Parallel Code Generation through Kernel Recognition. *18th Int. Parallel and Distributed Processing Symposium*, page 79b (10 pages), Santa Fe, April 2004.
6. R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Philadelphia: Society for Industrial and Applied Mathematics, 1994.
7. C. Cascaval, L. DeRose, D. A. Padua, and D. A. Reed. Compile-time Based Performance Prediction. *12th Int. Workshop on Lang. and Compilers for Parallel Computing*, volume 1863 of *Lecture Notes in Computer Science*, pages 365-379. Springer-Verlag, 2000.
8. B. B. Fraguela, R. Doallo, and E. L. Zapata. Modeling Set Associative Caches Behavior for Irregular Computations. *ACM Int. Conference on Measurement and Modeling of Computer Systems SIGMETRICS/PERFORMANCE*, pages 192-201, Madison, USA 1998.
9. B. B. Fraguela, R. Doallo, and E. L. Zapata. Probabilistic Miss Equations: Evaluating Memory Hierarchy Performance. *IEEE Trans. on Computers*, 52(3):321-336, March 2003.
10. S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Trans. on Prog. Lang. and Systems*, 21(4):702-745, July 1999.
11. V. Kislenkov, V. Mitrofanov, and E. Zima. Multidimensional Chains of Recurrences. *Int. Symposium on Symbolic and Algebraic Computation*, pages 199-206, Rostock, Germany, 1998.
12. R. E. Ladner, J. D. Fix, and A. LaMarca. Cache Performance Analysis of Traversals and Random Accesses. *10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 613-622, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.
13. Y. Lin and D. A. Padua. On the Automatic Parallelization of Sparse and Irregular Fortran Programs. *4th Workshop of Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 41-56, Pittsburgh, 1998.
14. J. Sánchez and A. Gónzalez. Analyzing Data Locality in Numeric Applications. *IEEE Micro*, 20(4):58-66, July 2000.
15. R.A Uhlig and T.N. Mudge. Trace-Driven Memory Simulation: A Survey. *ACM Computing Surveys*, 29(2):128-170, June 1997.
16. X. Vera, N. Bermudo, J. Llosa, and A. González. A Fast and Accurate Framework to Analyze and Optimize Cache Memory Behavior. *ACM Trans. on Prog. Lang. and Systems*, 26(2):263-300, 2004.
17. E. Zima. Simplification and Optimization of Transformations of Chains of Recurrences. *Int. Symposium on Symbolic and Algebraic Computation*, pages 42-50, Montreal, Canada, 1995.