A Fast and Accurate Framework to Analyze and Optimize Cache Memory Behavior

XAVIER VERA, NERINA BERMUDO, JOSEP LLOSA, and ANTONIO GONZÁLEZ
Universitat Politècnica de Catalunya-Barcelona

The gap between processor and main memory performance increases every year. In order to overcome this problem, cache memories are widely used. However, they are only effective when programs exhibit sufficient data locality. Compile-time program transformations can significantly improve the performance of the cache. To apply most of these transformations, the compiler requires a precise knowledge of the locality of the different sections of the code, both before and after being transformed.

Cache miss equations (CMEs) allow us to obtain an analytical and precise description of the cache memory behavior for loop-oriented codes. Unfortunately, a direct solution of the CMEs is computationally intractable due to its NP-complete nature.

This article proposes a fast and accurate approach to estimate the solution of the CMEs. We use sampling techniques to approximate the absolute miss ratio of each reference by analyzing a small subset of the iteration space. The size of the subset, and therefore the analysis time, is determined by the accuracy selected by the user. In order to reduce the complexity of the algorithm to solve CMEs, effective mathematical techniques have been developed to analyze the subset of the iteration space that is being considered. These techniques exploit some properties of the particular polyhedra represented by CMEs.

Categories and Subject Descriptors: C.1.0 [**Processor Architectures**]: General; C.4 [**Performance of Systems**]: *Measurement techniques*; D.3.4 [**Programming Languages**]: Processors—*Compilers optimization*

General Terms: Design, Performance

Additional Key Words and Phrases: Cache memories, optimization, sampling

1. INTRODUCTION

Memory latency is critical for the performance of current computers, which have the memory organized hierarchically in such a way that the lower levels are smaller and faster. The lowermost level typically has a very short latency

This work has been supported by the ESPRIT project MHAOTEU (EP 24942) and the CICYT project 511/98.

Authors' addresses: X. Vera, currently at Institutionen för Datateknik, Mälardalens Högskola, P.O. BOX 883, Västerås SE-721 23, Sweden; email: xavier.vera@mdh.se; N. Bermudo, currently at CD Lab Compilation Techniques, Institute for Computer Languages, Vienna University of Technology, Argentinierstrasse 8/4, Vienna A-1040, Austria; email: nerina@complang.tuwien.ac.at; J. Llosa and A. González, Computer Architecture Department, Universitat Politècnica de Catalunya-Barcelona, Jordi Girona 1-3, Barcelona 08034, Spain; email: {josepll,antonio}@ac.upc.es.

(e.g., one to two cycles) but the latency of the upper levels may be a few orders of magnitude larger (e.g., main memory latency may be around 100 cycles). Various hardware and software approaches have been proposed lately for increasing the effectiveness of memory hierarchy. Software-controlled prefetching [Mowry et al. 1992] hides the memory latency by overlapping a memory access with computation and other accesses. Another useful optimization is applying loop transformations such as tiling [Carr and Kennedy 1992; Coleman and McKinley 1995; Lam et al. 1991; Wolf and Lam 1991] and data layout transformations [Chatterjee et al. 1999; Kandemir et al. 1999; Rivera and Tseng 1998; Temam et al. 1993]. In all cases, a fast and accurate assessment of a program's cache behavior at compile time is needed to make an appropriate choice of parameter values.

Data cache behavior is very hard to predict. Simulators are used to describe it accurately. However, they are very slow and do not provide too much insight into the causes of the misses. Thus, current approaches are based on simple models (heuristics) for estimating locality [Carr et al. 1994; Coleman and McKinley 1995; Lam et al. 1991; Rivera and Tseng 1998, 1999]. Such models provide very rough performance estimates, and, in practice, are too simplistic to statically select the best optimizations.

Cache miss equations (CMEs) [Ghosh et al. 1999] are an analytical method that describes the cache behavior accurately. CMEs allows studying each reference in a particular iteration point independently of all other memory references. Deciding whether a reference causes a miss or a hit for a given iteration point is equivalent to deciding whether it belongs to the polyhedra defined by the CMEs. Unfortunately, even though the computation cost of generating CMEs is a linear function of the number of references, solving them is an NP-complete problem [Ghosh 1999], and thus trying to study a whole program may be infeasible.[1]

This article presents an efficient method for analyzing cache memory behavior. It consists of a set of techniques that, built on the top of the CMEs, make it feasible to use them as a cost model for implementing optimizations. Although CMEs are limited to perfect nested loops due to the lack of reuse analysis, our techniques can be applied to any kind of CME-polyhedra, independently of the kind of loop nest that is being analyzed.

Central to our approach are polyhedral analysis and the application of sampling techniques, which allow the cache analysis to be both fast and accurate. The contributions of this work are summarized below:

—*Formulation of CMEs*. We develop a new formulation of the CMEs that describes in a low level the relationship between iteration space, memory references, and cache parameters. This new formulation allows us to develop specific techniques to deal with the CMEs.

—*CMEs Emptiness analysis of CMEs*. We derive some algorithms to reduce the number of polyhedra to be considered in our analysis. Since those polyhedra

[1]It is equivalent to deciding whether a solution exists to a system of equalities and inequalities, which is NP-complete [Banerjee 1988].

that do not contain integer points inside are useless, some techniques have been developed in order to identify them.

—*Sampling*. We estimate the result of the CMEs by means of sampling techniques. This approach represents a compromise between expensive simulations and potentially imprecise heuristics; it is very fast but the result is given as a confidence interval instead of a single value, which in practice is enough for many purposes. The user can set both the width of the interval and the confidence level.

—*Iteration points analysis*. We use mathematical techniques to analyze the behavior of the different iteration points efficiently. By exploiting some intrinsic properties of the particular types of polyhedra generated by CMEs, we reduce the complexity of the algorithm that checks whether an iteration point results in a miss, which translates to very high speedups compared to standard algorithms.

—*Prototyping implementation*. We have implemented our system in the Polaris Compiler. We used the Ictineo library in order to obtain low-level information. Our prototype obtains the reuse vectors, generates the CMEs, detects those that are empty, and solves them applying sampling.

—*Experimental results*. We present results for a set of loop fragments drawn from different SPECfp95 programs. Our experimental results show that the proposed method can compute their miss ratios in a few seconds. Moreover, the analysis time is generally several orders of magnitude faster than simulation on a typical workstation. We also show that the results obtained using our model are close to those from real execution on a Pentium-4 machine. This opens the possibility of including this analysis framework in production compilers to support and guide many optimizations.

The rest of the paper is organized as follows. Section 2 provides the mathematical terminology used in this work. Section 3 introduces our program model, defines the cache architecture used, and outlines our contributions to have a fast implementation of the CMEs. Next, we introduce our method in detail. Section 4 describes, for each equation, its mathematical formulation, presents the techniques used to remove empty polyhedra, and explains how the iteration points are analyzed. Then we describe our sampling technique in Section 5. Section 6 shows the accuracy and speed of our method. Section 7 presents a review of the related work and discusses some applications of our approach. We summarize the conclusions in Section 8.

2. BACKGROUND

This section reviews the definitions of some basic mathematical concepts.

2.1 Polyhedral Definitions

We give some basic definitions on polyhedra in order to introduce the concepts that are used in further sections.

Definition 2.1. Given the points $x_1, \ldots, x_n$ and scalars $\lambda_1, \ldots, \lambda_n$, we define a *convex combination* of $x_1, \ldots, x_n$ as $\sum_{i=1}^{n} \lambda_i x_i$ where $\sum_{i=1}^{n} \lambda_i = 1$ and all $\lambda_i \geq 0$.

Definition 2.2. A *vertex* of a set K is any point in K which cannot be expressed as a convex combination of any other distinct points in K.

Definition 2.3. A set K is *convex* $\iff$ every convex combination of any two points in K is also a point in K.

Definition 2.4. A *convex polyhedron* P is the intersection of a finite family of closed linear half-spaces which has the form $\{\vec{x} | \vec{a}\vec{x} \geq c\}$ where a is a nonzero row vector and c is a scalar constant.

We only consider bounded convex polyhedra because polyhedra defined by CMEs are convex and always bounded. Since the points of a convex bounded polyhedron can be expressed as convex combinations of their vertices, a polyhedron is fully described by its vertices. Therefore, a polyhedron can be given by either a system of linear constraints or a set of vertices [Wilde 1993]. Both representations will be considered in the development of our techniques.

Definition 2.5. We define the *real domain* of a variable x in a polyhedron P as the range of real values it takes inside P.

Definition 2.6. We define the *integer domain* of a variable x in a polyhedron P as the range of integer values it takes inside P.

2.2 Statistical Overview

This subsection presents the basic statistical concepts that will be used in this work. We first introduce random variables, which are used to model the number of misses. Then, we explain how we can estimate their behavior.

2.2.1 *Discrete Random Variables.* Random variables are functions defined over the probability space [DeGroot 1998].

Let $\mathcal{S} = (\Omega, \mathcal{A}, P)$ be a probability space (where Ω is the sample space, $\mathcal{A} \subset \wp(\Omega)$,[2] and P is the probability function). Let $X : \Omega \rightarrow \mathbb{R}$ be a real random variable (RV) defined over $\mathcal{S}$. X is said to be a discrete random variable when the image set is finite or numerable. There are several RVs that have been deeply studied due to their importance and the number of usual phenomena that they describe. Now we review two of them that are used in our model.

Let X be a real discrete random variable:

—We say that it follows a Bernoulli distribution ($X \sim B(p)$) when the image set has only two elements. Bernoulli RVs describe the random experience in which only two things can happen: success or miss. We define $\mathcal{T} \subset \Omega$ as the set of results obtained that we consider as "success." Thus:

$$\begin{aligned} X : \Omega &\longrightarrow \mathbb{R}, \\ \omega &\longmapsto \begin{cases} 0 \iff w \notin \mathcal{T}, \\ 1 \iff w \in \mathcal{T}. \end{cases} \end{aligned}$$

We note $P[X = 0] = p$ as the probability that the RV X is 0. Therefore, the probability $P[X = 1]$ is $q = 1 - p$, since $p + q$ must be 1.

[2] $\wp(X)$ is the set of all the possible subsets of X.

—Binomial distribution (represented by $X \sim Bin(n, p)$) models phenomena where n different and independent experiments modeled by Bernoulli RVs take place. This RV represents the number of successes.
Once $\mathcal{T} \subset \Omega$ is defined, we obtain

$$\begin{aligned} X : \Omega^n &\longrightarrow \mathbb{R}, \\ (\omega_1, \ldots, \omega_n) &\longmapsto card\{i|\omega_i \in \mathcal{T}\}. \end{aligned}$$

$P[X = k]$, $k = 0 \cdots n$ represents the probability that k experiments out of n succeed. Thus,

$$P[X = k] = \binom{n}{k} p^k (1-p)^{(n-k)}.$$

2.2.2 *Parameters Estimation.* Sometimes it is desired to study a certain property of a large set of elements (also called *population*), but it is impossible due to its size. In these cases it is interesting to reduce the problem size. One way to overcome this problem is to analyze a subset and infer to the population the results obtained for the sample. In our case, we model the behavior of a reference using a binomial-RV, where the different experiments consist in taking an iteration point and checking whether it results in a miss.

Let $X \sim Bin(n, p)$, and assume that p (the probability of success) is unknown. The way to obtain an approximation of p is to evaluate the behavior of a subset of the population (called *sample*), which yields the empiric values of the parameters that describe the sample-RV, and to infer these values to the population-RV.

Let $\mathcal{Q} \subset \Omega^n$ be the sample, $N = card(\Omega^n)$, and $k = card(\mathcal{Q})$. The value $\hat{p}$ is defined as

$$\hat{p} = \frac{successes \in \mathcal{Q}}{k}$$

and $Y \sim Bin(k, \hat{p})$ is the RV that describes the behavior of the sample.

If k is large enough, the sample does not contain repeated elements, and the value of the parameter p is not close either to 0 or 1, we can approximate $\hat{p}$ by means of the well-known normal or Gauss distribution:[3]

$$\hat{p} \simeq \mathrm{N}\left(p, \sqrt{\frac{pq}{k}}\right).$$

Thus,

$$\frac{(\hat{p} - p)}{\sqrt{\frac{pq}{k}}} \sim \mathrm{N}(0, 1),$$

which allows estimating Y by means of a normal-RV. We summarize and formalize the conditions in the following list:

(1) The sample does not contain repeated elements.
(2) $\frac{k}{N} \leq 0.05$.
(3) $\hat{p}k \geq 5$ and $\hat{q}k \geq 5$.
(4) $k \geq 30$.

[3] $Z \sim \mathrm{N}(0, 1)$ is the normalized Gauss distribution; $z_\alpha = P[-\alpha < Z < \alpha]$.

```
parameter (N)
REAL*8 a(N,N), b(N,N), c(N,N)
do i = 1, N
  do j = 1, N
    do k = 1, N
      a(i,j) = a(i,j) + b(i,k) * c(k,j)
    enddo
  enddo
enddo
```

Fig. 1. A running example: matrix multiply algorithm.

The approximation of the value of p is calculated in terms of *confidence* intervals. The meaning of the *confidence* (c%) is such that if we generate many samples and compute $\hat{p}$, $\hat{p}$ lies in the calculated interval in c% of the cases (e.g., if the percentage is 95%, it represents that, for 95 out of every 100 different samples, $\hat{p}$ will belong to the confidence interval).

Once a confidence $\alpha = 1 - c$ is chosen, a confidence interval for the value of p is given by the following expression:

$$p \in \hat{p} \pm z_{\frac{\alpha}{2}} \sqrt{\frac{\hat{p}\hat{q}}{k}}.$$

3. TERMINOLOGY AND OVERVIEW

3.1 Program Model

The current model applies to numerical codes consisting of perfectly nested loops. The usual data accesses in these codes are array accesses.[4] In this article, all programs are in FORTRAN 77. Thus, all arrays are stored in column major.

Definition 3.1 *(Iteration point).* Let us consider an n-dimensional nested loop with loop indexes $I_1, \ldots, I_n$. An execution of the loop when $I_1 = i_1, \ldots, I_n = i_n$ is identified by the vector $i = (i_1, \ldots, i_n)$. Since each of these vectors represents the coordinates of a point in $\mathcal{Z}^n$, we call them *iteration points*.

Definition 3.2. The *iteration space* of an n-dimensional loop nest is the polytope bounded by the bounds of the n enclosing loops.

Let us introduce a running example we will use through this article. For illustration purposes, we consider the matrix multiply algorithm as shown in Figure 1. It has a nested loop of depth 3. Each loop iterates N times, with the lower bound equal to 1 and the upper bound equal to N. Therefore, the iteration space of this nested loop is defined by $\{(i, j, k) \mid 1 \le i \le N, 1 \le j \le N, 1 \le k \le N\}$. Figure 2 illustrates the shape of the iteration space when N is 3.

Definition 3.3. A *reference* is a static read or write in the program. Highlighted in Figure 1 we can see the four different references of our running example.

[4]A scalar is represented either as register-allocated or as a one-dimensional (1-D) array.

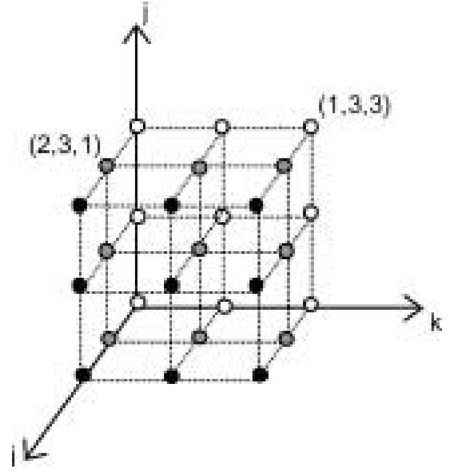


Fig. 2. Net representing the iteration space of Figure 1 when ($N = 3$).

Definition 3.4. A *memory access* is the execution of a reference at a particular iteration point of the loop nest enclosing the reference. For instance, *b(1,1)* and *b(1,3)* are memory accesses of the reference *b(i,j)*.

We assume some constraints that ensure that our analysis can be done in the polyhedral model [Feautrier 1996]. Loop bounds and subscript expressions are affine expressions of the enclosing loop nests. These constraints are not very restrictive for static analytical models, since they are limited to static control flows and memory accesses. Indeed, these constraints are assumed by all different analytical models reviewed in Section 7.

In addition, our model analyzes cache behavior statically. Therefore, the base addresses of all arrays and the sizes of all their dimensions must be known statically.

3.2 Cache Model

We consider a uniprocessor with two levels of memory. The cache is virtually indexed and an LRU replacement policy is used.

In a k-way set-associative cache, each cache set contains k different cache lines. Cache size (C) defines the number of bytes a cache can hold, whereas the line size (L) determines how many contiguous bytes are fetched from memory when a cache miss occurs. Thus, $C = N \times L \times k$, where N denotes the number of sets in the cache. Formally defined:

Definition 3.5. A *memory line* refers to a cache-line-sized block in main memory.

Definition 3.6. A *cache line* refers to the actual cache block in which a memory line is mapped. In a set-associative cache, the set of cache lines a memory line can map to is called a *cache set*.

Understanding reuse is essential to predict cache behavior, since a datum will only be in the cache if its line was referenced sometime in the past. *Reuse*

Table I. Reuse Vectors for References in Figure 1. *R* Stands for READ, *W* for WRITE

Reusing reference	Reused reference		Reuse vector
a(i, j) (*R*)	Self-spatial		(1, 0, 0)
	Self-temporal		(0, 0, 1)
b(i, k)	Self-spatial		(1, 0, 0)
	Self-temporal		(0, 1, 0)
c(k, j)	Self-spatial		(0, 0, 1)
	Self-temporal		(1, 0, 0)
a(i, j) *(W)*	a(i, j) *(R)*	Group temporal	(0, 0, 0)
	Self-spatial		(1, 0, 0)
	Self-temporal		(0, 0, 1)

happens whenever the same data item is referenced multiple times. Trying to determine all iterations that use the same data is extremely expensive. Thus, we use a concrete mathematical representation that describes the direction as well as the distance of the reuse in a methodical way. The shape of iterations that reuses the same data is represented by a *reuse vector space* [Wolf and Lam 1991]. We use the *reuse vectors* for a basis that represents such space.

Definition 3.7. If a reference R accesses the same memory line in iterations $\vec{i_1}$ and $\vec{i_2}$, where $\vec{i_1} \leq \vec{i_2}$, we define the reuse vector $\vec{r}$ as $\vec{r} = \vec{i_2} - \vec{i_1}$.

Given a reference, we may observe four different kinds of reuse:

—*Self-temporal*. A self-temporal reuse takes place when a reference accesses the same data element in different iterations of the loop.
—*Self-spatial*. A self-spatial reuse takes place when a reference accesses the same memory line in different iterations of the loop.
—*Group-temporal*. A group-temporal reuse takes place when two different references access the same data element in different iterations of the loop.
—*Group-spatial*. A group-spatial reuse takes place when two different references access the same memory line in different iterations of the loop.

Whereas self-reuse (both spatial and temporal) and group-temporal reuse are computed in an exact way, group-spatial reuse is only considered among uniformly generated references, that is, references whose array subscripts differ at most in the constant term [Gannon et al. 1988]. Table I lists all reuse vectors for the references of our running example shown in Figure 1.

3.3 An Overview

CMEs [Ghosh et al. 1999] are an analysis framework that describes the behavior of cache memory. Briefly, the general idea is to obtain for each memory reference a set of constraints and equations that represents the cache misses. These equations are defined over the iteration space.

In order to describe reuse among memory accesses, CMEs make use of the well-known concept of reuse vectors [Wolf and Lam 1991]. To discover whether a reuse translates to locality we need to know all the data brought to the cache between the two accesses and the particular cache architecture we are analyzing.

CMEs set up a set of equations[5] that describe the iteration points where the reuse is not realized. For each reuse vector, two kinds of equations are generated:

—*Cold*[6] *equations:* Given a reference, they represent the first time a memory line is touched. We may distinguish between spatial and temporal reuse:
  —*Temporal reuse:* The reuse is not realized when the studied reference reuses from an iteration point outside the iteration space (*cold miss equations*).
  —*Spatial reuse:* The reuse does not hold when either the analyzed reference reuses from data mapped to another cache line (*cold miss bounds*), or the reference reuses data from an iteration point outside the iteration space (*cold miss equations*).

—*Replacement*[7] *equations:* Given a reference, replacement equations represent the interferences with any other reference, including itself (self-conflicts).

Even though generating the equations is linear in the number of references, solving them can be very time consuming (the appendix shows the existing methods to derive information such as the number of misses as well as the cause of these misses from the equations).

Our goal consists in obtaining the number of cache misses in a reasonable time for any cache configuration. Therefore, we have developed a technique that builds upon traversing the iteration space (see Section A.2). We summarize the difficulties when solving the CMEs below:

(1) All the iteration points should be analyzed, which is $O(\#iteration\_points)$.
(2) CMEs define convex bounded polyhedra (see Section 2.1) with the enclosed integer points representing potential cache misses. Thus, the complexity of any method to count CMEs is a function of the number of CMEs polyhedra. The number of polyhedra is $O(\#references^2)$.
(3) The cost of checking whether an iteration points belongs to a particular polyhedron is exponential to the number of dimensions and their domains. The number of dimensions of each polyhedron is $O(nesting\_depth)$.

Our approach uses statistical techniques to reduce the computation time. Another part of our approach deals with the CMEs polyhedra. Taking advantage of their particular topology, we have developed efficient techniques to both remove a large number of those that contain no information and analyze the behavior of the different iteration points.

Next, we introduce the different steps of our analysis framework, which tackle the problems shown above.

(1) *Sampling.* This approach uses statistical techniques to estimate the number of cache misses. It allows us to study only a small subset of the iteration

[5]The term *equation* has been used loosely to represent a set of simultaneous equalities and inequalities.
[6]They represent compulsory misses.
[7]They represent both conflict and capacity misses.

space. The reduction of the number of analyzed points translates to a small computation time while having a high accuracy.

(2) *Remove empty polyhedra.* A CME-polyhedron is considered empty when it does not contain any integer point, although it may contain real points. Hence, we have developed techniques to determine whether a CME-polyhedron is empty and thus reduces the number of polyhedra to be analyzed.

(3) *Analyze iteration points.* When analyzing an iteration point, we need to know whether it is a solution of the CMEs, that is, whether it fulfills the equations. We have developed some specific techniques for each type of polyhedron to check whether an iteration point is a solution to the equations or not. Since the dimension of most CMEs polyhedra is larger than the iteration space, after substituting the induction variables for the values given by the iteration point there is still a set of inequations left (which describes a new polyhedron). Therefore, checking whether an iteration point is solution of a CME is equivalent to checking whether the remaining polyhedron is empty or not.

The following sections explain in detail how we apply these techniques to have a fast and accurate framework to predict cache behavior.

4. CMEs MODELING

This section introduces our polyhedral model for the CMEs. We start giving some notation that is used through this section. Then, we give a formal mathematical definition of the equations. For each type of equation, we explain our techniques to remove those that are empty and show how we determine whether an iteration point results in a miss for the remaining equations.

4.1 Introduction

The part of our study that focuses in the treatment of CMEs polyhedra is mainly based on their structure. Therefore, the interpretation of the different constants that appear in the definition of the equations is avoided except in some special cases, where the significance of some of these constants is useful for the development of our techniques.

We assume that $f_1, \ldots, f_m, g_1, \ldots, g_m$ are integer values. For each induction variable i_k $(k = 1, \ldots, m)$, ub_k and lb_k stand for the upper and lower bounds of this variable in the iteration space.

4.1.1 *A Criterion to Detect Empty Polyhedra.* We present a general technique to identify whether a polyhedron is empty. It provides a criterion for detecting the emptiness of polyhedra that is repeatedly used in different steps of our approach.

General criterion: If there exists a variable that cannot take any integer value, there will not be any integer point inside the given polyhedron. Notice that this condition is sufficient for the polyhedron to be empty of integer points. This gives us a general criterion for detecting empty polyhedra, although it does not detect all of them. For each variable x_k, its definition domain $[a_k, b_k]$

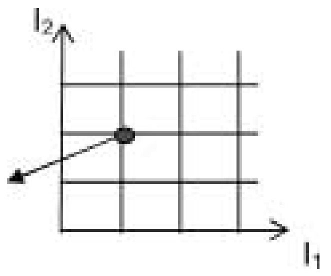


Fig. 3. Cold miss equations.

in the polyhedron is calculated (see Definition 2.5). Let lb_k and ub_k be the lower and upper bounds of the integer domain (see Definition 2.6), respectively. If $ub_k < lb_k$, there is no integer value inside the real domain $[a_k, b_k]$ of the variable; thus the polyhedron is empty.

This criterion does not detect all empty polyhedra. In order to increase the number of detected empty polyhedra, specific criteria for each type of polyhedron have been developed. These criteria rely on the structure of the equations and their interpretation in terms of the cache behavior.

4.2 Cold Miss Equations (Temporal or Spatial Reuse)

These equations describe the iteration points where a reuse does not translate to locality because the reference reuses from an iteration point outside the iteration space (see Figure 3). The obtained polyhedra are defined over the iteration space. This means that only induction variables appear in their definition (in the linear inequalities that characterize the set). The cold miss equations add a restriction on the possible values of one of the variables inside the iteration space.

(CM)

$$\begin{aligned} i_l &\le d_l, && \text{for a fixed } l \in [1, \dots, m], \\ lb_k \le i_k &\le ub_k, && k = 1 \cdots m, \end{aligned}$$

where i_l is the lth variable of the iteration space, $d_l \in \mathbb{Z}$, and the first equation represents an additional restriction on one of the variables.[8] Note that this equation could introduce a lower bound of the variable i_k instead of an upper bound. The other $2m$ constraints determine the iteration space.

4.2.1 *Empty Cold Miss Equations.* Since each of these polyhedra consists of the iteration space and an additional restriction on one of the variables, it will be empty if the restriction is incompatible with the iteration space. If the additional restriction has the form $i_l \le d_l$ and $d_l < lb_l$, there is a contradiction between the two conditions and we conclude that the polyhedron is empty. The same happens when the restriction has the form $i_l \ge d_l$ and $d_l > ub_l$. Hence, the time taken to compute the emptiness is $O(1)$.

[8] $i_l - r_l$ must belong to the domain of the lth variable of the iteration space.

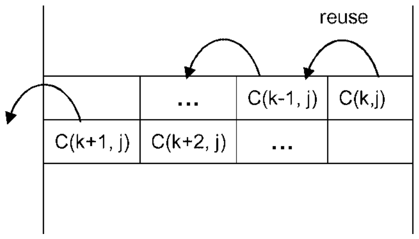


Fig. 4. Cold miss bounds.

4.2.2 *Determining an Iteration Point.* Let $\vec{i_0} = (i_{01}, i_{02}, \ldots, i_{0m})$ be the iteration point we want to study. The only inequality it might not fulfill is

$$i_l \leq d_l \tag{1}$$

since the others are just a characterization of the iteration space. So, $\vec{i_0}$ is a point inside the given cold miss polyhedron $\Longleftrightarrow$ its lth component fulfills the previous equation.

4.3 Cold Miss Bounds (Spatial Reuse)

These equations describe the iteration points where a reuse is not realized because the reference reuses data that are mapped to a different cache line (see Figure 4).

Given reference R_A, there is a cold miss along the reuse vector $\vec{r}$ in the iteration point $\vec{i}$ if the following equation holds:

$$Memory\_Line_{R_A}(\vec{i}) \neq Memory\_Line_{R'_A}(\vec{i} - \vec{r}),$$

where R'_A is R_A (if $\vec{r}$ is a self-reuse vector), or a different reference (if $\vec{r}$ is a group-reuse vector).

When extending this identity we obtain a set of inequations that describes a convex polyhedron which is defined over $\mathbb{R}^{m+1}$, m being the dimension of the iteration space. A new variable z is introduced for linearity reasons [Clauss 1996]. In fact, there is a version of the CMEs that ignores this variable [Ghosh et al. 1998], but we focus on the more precise model that includes it.

The equations have the following form:

(CMB)

$$\begin{aligned} f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - \ \mathrm{Lz} &\geq LB_1, \\ f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - \ \mathrm{Lz} &\geq LB_2, \\ f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - \ \mathrm{Lz} &\leq UB, \\ lb_k \leq i_k \leq ub_k, \qquad k &= 1 \cdots m, \end{aligned}$$

where $LB_1, LB_2, UB \in \mathbb{Z}$, and L stands for the cache line size. Note that the three first equations can also have the form

$$\begin{aligned} f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - \ \mathrm{Lz} &\geq LB, \\ f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - \ \mathrm{Lz} &\leq UB_1, \\ f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - \ \mathrm{Lz} &\leq UB_2, \end{aligned}$$

where $LB, UB_1, UB_2 \in \mathbb{Z}$. One of these constraints is redundant, so the polyhedron can be expressed as follows, where $UB = min\{UB_1, UB_2\}$:

(CMB)

$$\begin{array}{l} f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - \mathtt{Lz} \geq LB, \\ f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - \mathtt{Lz} \leq UB, \\ lb_k \leq i_k \leq ub_k, \qquad k = 1 \cdots m. \end{array}$$

4.3.1 *Empty Cold Miss Bounds.* Since the domains of $i_1, \ldots, i_m$ are explicitly given, and they are not constrained by any other equation, the only variable that might have a domain without integer values inside is variable z. Let us observe the constraints involving variable z.

$$f_1 i_1 + \cdots + f_m i_m - UB \leq \mathtt{Lz} \leq f_1 i_1 + \cdots + f_m i_m - LB$$

Let us define

$$\mathtt{z}_{max} = \frac{\max_{(i_1,\ldots,i_m)\in I}\{f_1 i_1 + \cdots + f_m i_m\} - LB}{\mathtt{L}},$$

$$\mathtt{z}_{min} = \frac{\min_{(i_1,\ldots,i_m)\in I}\{f_1 i_1 + \cdots + f_m i_m\} - UB}{\mathtt{L}},$$

where I stands for the iteration space. Thus, the integer domain of the variable z in the polyhedron (CMB) is

$$[\lceil \mathtt{z}_{min} \rceil, \lfloor \mathtt{z}_{max} \rfloor] \cap \mathbb{Z}$$

If there are no integer values inside this interval, we can conclude that (CMB) polyhedron is empty in $O(m)$. This condition is sufficient, but not necessary. That is, even if the domain of z contains integer values, the polyhedron might be empty.

4.3.2 *Determining an Iteration Point.* When an iteration point $\vec{i_0}$ is substituted, an *one-dimensional* polyhedron is obtained. Deciding whether $\vec{i_0}$ fulfills the equations is equivalent to deciding whether the *one-dimensional* polyhedron (CMB′)

$$LB' \leq -\mathtt{Lz} \leq UB'$$

is empty, where $LB' = LB - f_1 i_{01} - \cdots - f_m i_{0m}$ and $UB' = UB - f_1 i_{01} - \cdots - f_m i_{0m}$.

We obtain the integer domain z from its real domain, $[-\frac{UB'}{\mathtt{L}}, -\frac{LB'}{\mathtt{L}}] \subset \mathbb{R}$. We determine whether it is empty comparing its bounds.

4.4 Replacement Equations

Given a reference R_A and an iteration point $\vec{i}$, replacement equations represent those memory accesses which map to the same cache set as $R_A(i)$.

For each pair of references (R_A and R_B), the following expression gives the condition for a cache-set contention in a set-associative cache:

$$\begin{array}{c} Cache\_Set(\vec{i})_{R_A} = Cache\_Set(\vec{j})_{R_B}, \\ \vec{j} \in \mathcal{J}, \end{array}$$

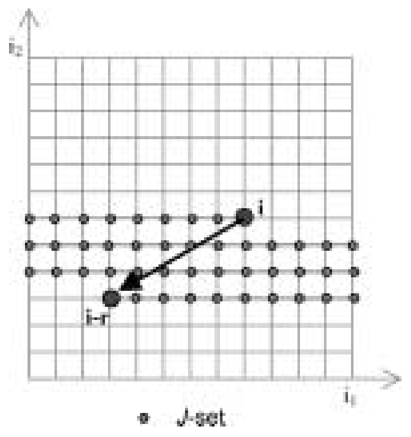


Fig. 5. J set.

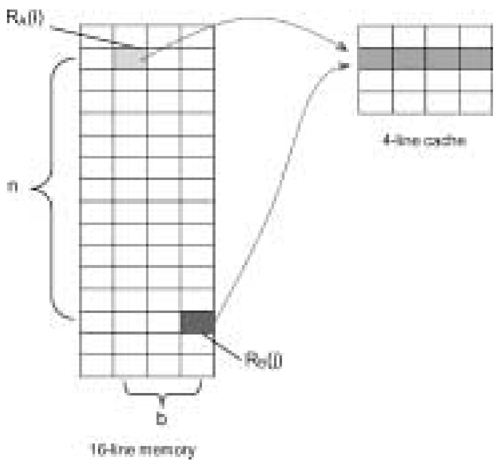


Fig. 6. Replacement equations.

where $\mathcal{J}$ represents the set of iteration points between $\vec{i}$ (the current one) and the iteration point from which R_A reuses, $\vec{i} - \vec{r}$ (see Figure 5).

This identity results in

$$\begin{aligned} Mem_{R_A}(\vec{i}) - Mem_{R_B}(\vec{j}) &= \mathbb{C}\mathtt{n} + \mathtt{b}, \\ \vec{j} &\in \mathcal{J}, \end{aligned}$$

where $\mathbb{C}$ stands for $\frac{\mathtt{C}}{\mathtt{k}}$, $\mathtt{k}$ is the associativity of the cache, $\mathtt{n}$ stands for the distance between R_A and R_B in cache size units, and $\mathtt{b}$ is the difference between the offset of each reference with respect to the beginning of their respective lines (see Figure 6).

This is the type of polyhedron obtained from the CMEs that has the most complex topology. A replacement polyhedron is contained in $\mathbb{R}^{2m+3}$. $2m$ of its variables $(i_1, \ldots, i_m,\ j_1, \ldots, j_m)$ refer in some way to the iteration space and the remaining variables (b, n, and z) are introduced for linearity reasons. Replacement equations have the following form:

(RCM)

$$\begin{aligned}
&-\mathtt{b} - \mathbb{C}\mathtt{n} + f_1 i_1 + \cdots + f_m i_m + g_1 j_1 + \cdots + g_m j_m = A,\\
&\mathtt{Lz} - \mathtt{b} + g_1 j_1 + \cdots + g_m j_m \geq B_1,\\
&\mathtt{Lz} + g_1 j_1 + \cdots + g_m j_m \leq B_2,\\
&\mathtt{Lz} + g_1 j_1 + \cdots + g_m j_m \geq B_1,\\
&\mathtt{Lz} - \mathtt{b} + g_1 j_1 + \cdots + g_m j_m \leq B_2,\\
&\mathtt{n} \neq 0,\\
&\vec{j} \in \mathcal{J},
\end{aligned}$$

where L stands as usual for the cache line size, $A, B_1, B_2 \in \mathbb{Z}$, and $p_k, q_k \in \mathbb{Z}, \forall k = 1, \ldots, m$.

We present some remarks below.

(1) The condition $\mathtt{n} \neq 0$ is split up into $\mathtt{n} \leq -1$ and $\mathtt{n} \geq 1$. Each of these inequalities combined with the remaining inequalities above defines a different polyhedron.

(2) $\mathcal{J}$ is the set of all potentially interfering points (see Figure 5). Depending on the access order of the references whose interferences are being studied, this set has one of the following forms:

$$\begin{aligned}
\mathcal{J} &= [\vec{i} - \vec{r}, \vec{i}),\\
\mathcal{J} &= (\vec{i} - \vec{r}, \vec{i}],\\
\mathcal{J} &= (\vec{i} - \vec{r}, \vec{i}),
\end{aligned}$$

where $\vec{i}$ stands for the iteration point and $\vec{r}$ stands for the considered reuse vector.

In general, $\mathcal{J}$ is not convex (see Figure 5). Therefore, it is divided in several convex regions [Ghosh et al. 1999]. The regions obtained have the following form:

$$\begin{aligned}
\alpha_k * i_k - j_k &\leq q_k, \qquad k = 1 \cdots m,\\
\beta_k * i_k - j_k &\geq p_k, \qquad k = 1 \cdots m,
\end{aligned}$$

where $\alpha_k, \beta_k \in \{0, 1\}$.

We obtain a replacement polyhedron for each convex region. In our exposition, we will assume $\alpha_k = 1$ and $\beta_k = 1$, since this fact does not introduce significant changes in the techniques proposed.

(3) The coefficient of the variable b in the equation is -1. Thus, we isolate b in the equation and express it as a function of the other variables:

$$\mathtt{b} = f_1 i_1 + f_2 i_2 + \cdots + f_m i_m + g_1 j_1 + g_2 j_2 + \cdots + g_m j_m - \mathbb{C}\mathtt{n} - A.$$

Substituting this expression of b in the inequations, a more simple form of the replacement equations is obtained.

Regarding the previous considerations, the standard replacement polyhedron is described by the following equations:

(RCM)

$$\begin{aligned}
&-\mathtt{Lz} - \mathbb{C}\mathtt{n} + f_1 i_1 + \cdots + f_m i_m \geq AL,\\
&-\mathtt{Lz} - \mathbb{C}\mathtt{n} + f_1 i_1 + \cdots + f_m i_m \leq AU,\\
&\mathtt{Lz} + g_1 j_1 + \cdots + g_m j_m \geq BL,\\
&\mathtt{Lz} + g_1 j_1 + \cdots + g_m j_m \leq BU,\\
&\mathtt{n} \geq 1,\\
&p_k \leq i_k - j_k \leq q_k, \qquad k = 1 \cdots m,\\
&lb_k \leq i_k \leq ub_k, \qquad k = 1 \cdots m,
\end{aligned}$$

where $AU, AL, BU, BL \in \mathbb{Z}$.

4.4.1 *Empty Replacement Equations.* Different criteria to detect empty replacement polyhedra have been developed. In this case, not only the information given by the equations is considered, but also its interpretation in terms of the cache behavior.

We consider a *replacement polyhedron empty if:*

(1) *Its "convex regions" do not belong to the iteration space.* In a replacement polyhedron, there is a subset of equations which relates the variables i_k and j_k for $k = 1, \ldots, m$:

$$\begin{aligned}
&i_k - j_k \geq p_k,\\
&i_k - j_k \leq q_k,\\
&k = 1, \ldots m.
\end{aligned}$$

These equations result from the division in convex regions of the domains of variables $j_1, \ldots, j_m$ [Ghosh et al. 1999]. In order to detect empty replacement polyhedra, we check whether these constraints are consistent with the fact that $\vec{i}$ and $\vec{j}$ must belong to the iteration space. The worst case complexity for checking this is $O(m)$.

(2) *The variable* n *and the expression* $Mem_{R_A} - Mem_{R_B}$ *have different sign.* Recall that replacement equations result from the following identity:

$$Mem_{R_A}(\vec{i}) - Mem_{R_B}(\vec{j}) = \mathbb{C}\mathtt{n} + \mathtt{b}.$$

Since the placement of the two references R_A and R_B in the memory is fixed, their relative position will not change; thus $Mem_{R_A}(\vec{i}) - Mem_{R_B}(\vec{j})$ has constant sign for all $\vec{i}$, $\vec{j}$. Besides, this sign must be the same as the sign of the variable n, because this variable represents the distance, in terms of cache size units, between the two references. A replacement polyhedron is empty if the range of feasible values of the expression $Mem_{R_A}(\vec{i}) - Mem_{R_B}(\vec{j})$ (which depends on the variables $i_1, \ldots, i_m$ and $j_1, \ldots, j_m$) causes a contradiction with the restriction that determines the sign of the variable n. This can be done in $O(1)$.

(3) *The range of the expression* $Mem_{R_A} - Mem_{R_B}$ *is incompatible with the restriction of the variable* n. The first two equations of (RCM) are

$$\begin{aligned}
&AL \leq -\mathtt{Lz} - \mathbb{C}\mathtt{n} + f_1 i_1 + \cdots + f_m i_m \leq AU,\\
&BL \leq \mathtt{Lz} + g_1 j_1 + \cdots + g_m j_m \leq BU.
\end{aligned}$$

Hence, the following expressions hold:

$$AU - BU \geq f_1 i_1 + \cdots + f_m i_m + g_1 j_1 + \cdots + g_m j_m - \mathbb{C}\mathtt{n}, \quad (2)$$
$$AL - BL \leq f_1 i_1 + \cdots + f_m i_m + g_1 j_1 + \cdots + g_m j_m - \mathbb{C}\mathtt{n}. \quad (3)$$

Depending on the constraint on the variable n, one of the two inequalities is chosen and gives a criterion to detect the emptiness of the polyhedron.

*n ≤ -1

As this restriction gives an upper bound of n, that is, a lower bound of $-$n, we consider the second restriction.

$$\begin{aligned} AU - BU &\geq f_1 i_1 + \cdots + f_m i_m + g_1 j_1 + \cdots + g_m j_m - \mathbb{C}\mathtt{n} \\ &\geq f_1 i_1 + \cdots + f_m i_m + g_1 j_1 + \cdots + g_m j_m + \mathbb{C} \\ &\geq \min_{\vec{i}\in I, \vec{j}\in J} \{ f_1 i_1 + \cdots + f_m i_m + g_1 j_1 + \cdots + g_m j_m \} + \mathbb{C}. \end{aligned}$$

*n ≥ 1

In this case the considered inequality is the first one.

$$\begin{aligned} AL - BL &\leq f_1 i_1 + \cdots + f_m i_m + g_1 j_1 + \cdots + g_m j_m - \mathbb{C}\mathtt{n} \\ &\leq f_1 i_1 + \cdots + f_m i_m + g_1 j_1 + \cdots + g_m j_m + \mathbb{C} \\ &\leq \max_{\vec{i}\in I, \vec{j}\in J} \{ f_1 i_1 + \cdots + f_m i_m + g_1 j_1 + \cdots + g_m j_m \} + \mathbb{C}, \end{aligned}$$

where I stands for the iteration space and J for the domains of $(j_1, \ldots, j_m)$.

In each of these cases, if the constraint does not hold, we conclude that the polyhedron is empty in $O(m)$.

(4) *The variable* n *cannot take integer values*. The inequations (2) and (3) are used in order to compute the domain of the variable n. If it contains no integer points, the polyhedron is empty.

Hence, the worst-case complexity to detect empty replacement polyhedra is $O(m)$. Note that by means of these techniques we might not detect all empty replacement polyhedra. However, we show in Figure 7 in Section 6 that the number of them is considerably high.

4.4.2 *Determining an Iteration Point.* When considering a k-way set-associative cache with LRU replacement, an iteration point $\vec{i_0}$ fulfills the equations if the polyhedron contains a set of integer points with k different values of the variable n (that represent k distinct contentions, k ≥ 1).

In this section we present a criterion for determining whether a replacement polyhedron is empty. For those polyhedra that are not filtered out by this criterion, the number of points inside them will be counted.

We propose a method for counting integer points inside replacement polyhedra that works for either direct-mapped or set-associative caches.

The polyhedron (RCM′) obtained after substituting $\vec{i_0} = (i_{01}, i_{02}, \ldots, i_{0m})$ in the equations of a replacement polyhedron has the following form:

(RCM′)

$$\begin{gathered} AL' \le \mathtt{Lz} + \mathbb{C}\mathtt{n} \le AU', \\ BL \le \mathtt{Lz} + g_1 j_1 + \cdots + g_m j_m \le BU, \\ \mathtt{n} \ge 1, \\ q'_k \le j_k \le p'_k, \qquad k = 1 \cdots m. \end{gathered}$$

This polyhedron is empty if there exists no integer combination of the variables $j_1, j_2, \ldots, j_m, \mathtt{z}, \mathtt{n}$, that fulfills the equations. By construction, $BU - BL = AU - AL = \mathtt{L} - 1$, where $\mathtt{L}$ is the cache line size. Thus, $AU' - AL' = \mathtt{L} - 1$.

Note that as we are only interested in integer solutions, the inequalities $AL' \le \mathtt{Lz} + \mathbb{C}\mathtt{n} \le AU'$ are equivalent to the following system of diophantine equations:

$$\begin{gathered} \mathtt{Lz} + \mathbb{C}\mathtt{n} = D, \\ D = AL', \ldots, AU', D \in \mathbb{Z}. \end{gathered} \tag{4}$$

According to the linear diophantine equations theory, an equation of the form $\mathtt{Lz} + \mathbb{C}\mathtt{n} = D$ has a solution $\iff Gcd(C, L)$ divides D [Banerjee 1993]. Since $\mathbb{C} = \frac{\mathtt{C}}{\mathtt{k}}$ represents the cache size, $\mathbb{C} = N * \mathtt{L}$ for a certain $N \in \mathbb{N}$. It is straightforward that

$$Gcd(\mathbb{C}, \mathtt{L}) = \mathtt{L}.$$

Therefore, the previous equation $\mathtt{Lz} + \mathbb{C}\mathtt{n} = D$ has a solution $\iff$ $\mathtt{L}$ divides D, and thus the system of Equations (4) has a solution $\iff$ there exists a value of D in $[AL', AU']$ that is multiple of $\mathtt{L}$.

Since $AU' - AL' = \mathtt{L} - 1$, the interval $[AL', AU']$ *mod* $\mathtt{L}$ contains all the values of $\mathbb{Z}_L$.[9] As the previous interval can be written as $AL' + [0, \mathtt{L} - 1]$, we have that

$$\forall x \in \mathbb{Z}_L \quad \exists D \in [AL', AU'] \mid x \equiv D \ mod\ \mathtt{L}.$$

In fact, since the number of integer points in $[AL', AU']$ is $\mathtt{L}$, the relation defined before is bijective. For this reason, since $0 \in \mathbb{Z}_L$, there will always be only one value of D for which the equation $\mathtt{Lz} + \mathbb{C}\mathtt{n} = D$ has a solution. Let $D0$ be that value:

$$D0 = \left\lceil \frac{AL'}{\mathtt{L}} \right\rceil * \mathtt{L}.$$

Then, (RCM′) can be written as follows:

(RCM′)

$$\begin{gathered} \mathtt{Lz} = D0 - \mathbb{C}\mathtt{n}, \\ BL - D0 \le -\mathbb{C}\mathtt{n} + g_1 j_1 + \cdots + g_m j_m \le BU - D0, \\ \mathtt{n} \ge 1 \\ q'_k \le j_k \le p'_k, \qquad k = 1 \cdots m. \end{gathered}$$

[9] $\mathbb{Z}_L = \mathbb{Z}\ mod\ \mathtt{L}$.

Let be $BL' = BL - D0$ and $BU' = BU - D0$. We obtain

(RCM′)

$$\begin{gathered} BL' \leq -\mathbb{C}\mathrm{n} + g_1 j_1 + \cdots + g_m j_m \leq BU', \\ \mathrm{n} \geq 1, \\ q'_k \leq j_k \leq p'_k, \qquad k = 1 \cdots m. \end{gathered}$$

We assume that $q'_k \neq p'_k$, $\forall k$. Otherwise, we substitute the value of the respective j's in the inequalities and we obtain a polyhedron with the same structure, but lower dimension.

Let us observe the system of linear diophantine equations corresponding to the first two inequations above:

$$-\mathbb{C}\mathrm{n} + g_1 j_1 + \cdots + g_m j_m = E, \qquad E = BL', \ldots, BU' \quad E \in \mathbb{Z}.$$

Each of these equations has solution $\iff G = Gcd(g_1, \ldots, g_m, \mathbb{C})$ divides E. However, neither the existence nor the uniqueness of a value $E0$ for which the corresponding equation has a solution is ensured.

$$E0 = \left\lceil \frac{BL'}{G} \right\rceil * G.$$

$E0$ is the smallest integer value of E, greater than BL', for which an equation of the form $-\mathbb{C}\mathrm{n} + g_1 j_1 + \cdots + g_m j_m = E$ has a solution. If $E0$ does not belong to the domain of E, that is, if $E0 > BU'$, then (RCM′) is empty.

On the other hand, a feasible value of E does not ensure that the polyhedron is not empty and, in this case, the number of integer points inside the polyhedron must be counted. Due to the particular form of this polyhedron, the number of integer solutions can be computed in a more efficient way than for general polyhedra. Assuming that all the Gcd are computed, the complexity of the algorithm is $O(m)$.

4.4.3 *Counting Integer Points.* The method for counting presented next is based on the fact that the vertices of a polyhedron are extreme points. This implies that the greatest and smallest values that any variable can have inside a polyhedron can be found in the vertices. Therefore, the computation of the domain of a variable can be done using its vertices. We first introduce a general method, and then we extend it with a new technique to compute the domains of the variables.

The general method is as follows: Let P be a polyhedron in $\mathbb{R}^p$. We take a variable x_i and calculate its integer domain $[lb_i, ub_i]$. Then, for every integer value z in this domain, we consider the *(p-1)-dimensional* polyhedra that result from giving the variable x_i the value z. This process is repeated recursively, until we have polyhedra defined by only one variable.

Let $P^1_1, \ldots, P^1_M$ be these polyhedra. The number of integer points inside one of them is $ub - lb + 1$, where ub and lb are the upper and lower bounds of the corresponding variable. The total number of integer points in the polyhedron is obtained by adding the points of $P^1_1, \ldots, P^1_M$.

The following are some remarks on this method:

(1) The selection of the variable to be fixed is not irrelevant. In general, the domain of a variable in a polyhedron is a function of the other variables. Thus we choose every time the variable that has the smallest domain in order to minimize the number of nodes in the recurrence tree. Although we do spend some time in choosing the variable, this criterion helps us to reduce the time consumed for counting the number of integer points inside the polyhedron.

(2) The domains of the variables are calculated as follows: Let x_k be the variable whose domain we want to determine. Let V_P be the set of vertices of P. Then the bounds of the integer domain of the variable in P are

$$lb_k = \left\lceil \min_{v=(v_1,\ldots,v_n)\in V_P} v_k \right\rceil,$$

$$ub_k = \left\lfloor \max_{v=(v_1,\ldots,v_n)\in V_P} v_k \right\rfloor.$$

Unfortunately, computing the vertices of a polyhedron is a problem with exponential complexity. Our approach avoids this expensive phase of the computation.

From the definition of (RCM′) we can derive the following conclusion: the domains of the variables $j_1, \ldots, j_m$ are explicitly given in the expression of the polyhedron, so they do not need to be calculated. The domain of the variable $\mathtt{n}$ can be calculated by means of the next two inequations:

$$g_1 j_1 + \cdots + g_m j_m - BU' \le \mathbb{C}\mathtt{n} \le g_1 j_1 + \cdots + g_m j_m - BL'. \tag{5}$$

Let us define

$$\begin{aligned} \mathtt{n}_{max} &= \frac{\max_{(j_1,\ldots,j_m)\in J}\{g_1 j_1 + \cdots + g_m j_m\} - BL'}{\mathbb{C}}, \\ \mathtt{n}_{min} &= \frac{\min_{(j_1,\ldots,j_m)\in J}\{g_1 j_1 + \cdots + g_m j_m\} - BU'}{\mathbb{C}}, \end{aligned}$$

where J stands for the domain of $\vec{j} = (j_1, \ldots, j_m)$. Then the integer domain of the variable $\mathtt{n}$ in the polyhedron (RCM′) is

$$[\lceil \mathtt{n}_{min} \rceil, \lfloor \mathtt{n}_{max} \rfloor] \cap \mathbb{Z}.$$

Hence, we conclude that the domains of all variables are easily computed and that the explicit computation of the vertices is not needed.

Since the domains of the variables $j_1, \ldots, j_m$ may change when the variable $\mathtt{n}$ is fixed, the order in which the variables will be fixed cannot be determined at the beginning. Thus, the real domain of these variables must be recalculated every time. This is done in a similar way to the computation of the domain of $\mathtt{n}$ in the initial polyhedron: for each variable j_k, its greatest and lowest values given by the two inequations (Equation (5)) are calculated. The actual domain of this variable is the intersection of this interval and the explicit domain given by the equations of the polyhedron.

In order to detect empty polyhedra, the search of empty integer domains must be done for all the variables. Theoretically, the complexity is $O(\#iteration\_points)$, but in practice it is $O(1.5^m)$ for our benchmarks.

4.5 Summary and Review

Overall, we have introduced our polyhedral model to handle the CMEs. We have formally described each class of equation, and shown all steps to transform a high-level definition into a low-level characterization. Then, based on this new formulation, we have developed a set of methods that allows us to remove many empty polyhedra. Furthermore, we introduce new methods to check in a fast way whether an iteration point belongs to an equation.

Now, we review three different general-purpose methods to check the emptiness of polyhedra. We compare them to the specific algorithms shown above.

The Omega test [Pugh 1991] checks if a set of affine constraints has an integer solution, which is equivalent to check whether the associated polyhedron is empty. It manipulates Presburguer formulas [Kreisel and Krevine 1967] with parameters, and the best upper bound on the complexity of this algorithm is $2^{2^{2^p}}$, where p is the domain of the variables that appear in the formula. In our case, p corresponds to the size of the longest dimension of the iteration space.

The method described by Pugh for counting integer points in polyhedra consists of a set of techniques that are iteratively applied [Pugh 1994]. Choosing the technique to be applied at each step is done by hand and no systematic approach has been proposed.

M. Haghighat and C. Polychronopoulos presented a method for volume computation of polyhedra [Haghighat and Polychronopoulos 1993]. They defined a set of rules, but nothing was said about how to decide which rules to apply at each step.

Ehrhart polynomials allow computing the number of integer solutions in a set of linear constraints [Clauss 1996]. It is a method oriented to parameterized polyhedra. The first step of this approach is the computation of the parameterized vertices, for which it uses the Fourier-Motzkin transformation. Then the number of points of a given number of nonparameterized polyhedra is computed in order to determine each coefficient of the polynomial. The complexity of the Fourier-Motzkin transformation is $O(\#constraints^{\lfloor \frac{\#variables}{2} \rfloor})$ and the complexity of the remaining steps to compute the parameterized vertices is $O(\#variables * p^3)$, where p is the number of parameters. In our case the number of constraints is $2m + 3$ and the number of variables is $m + 1$, where m is the dimension of the loop nest.

5. SAMPLING

Obtaining the information from the CMEs is not straightforward. Since we want to analyze both direct-mapped and set-associative caches, our proposal builds upon the second method to solve the CMEs (traversing the iteration space as shown in the Appendix). This approach to solve the CMEs allows us to study each reference in a particular iteration point independently of all other memory references. Based on this property, a small subset of the iteration space is analyzed, strongly reducing the computation cost. In particular, we use random sampling to select the iteration points. This sampling technique cannot be applied to a cache simulator. A simulator cannot analyze an isolated memory reference, since it requires information of all previous references to decide whether it results in a miss or a hit.

5.1 Modeling CMEs

This subsection describes how the statistical techniques (see Section 2.2) are used to analyze the CMEs.

We are interested in finding the number of misses that a loop nest produces ($\#m$). In order to obtain it, we define a RV for each reference within the loop nest that returns the number of misses. Below, we show that this RV follows a binomial distribution. Thus, we can use the statistical techniques shown in Section 2.2.2 to compute the parameters that describe it.

For each memory instruction, we can define a Bernoulli RV $X \sim B(p)$ as follows:

$$\begin{aligned} X : \mathit{Iteration\ Space} &\longrightarrow \mathbb{R}, \\ \vec{i} &\longmapsto \{0, 1\}, \end{aligned}$$

such that $X(\vec{i}) = 1$ if the memory instruction results in a miss for iteration $\vec{i}$, $X(\vec{i}) = 0$ otherwise. Note that X describes the experiment of choosing an iteration point and checking whether the memory instruction produces a miss for it, and p is the probability of success. The value of p is $p = \frac{\#m}{N}$, where N is the number of iteration points.

Then, we repeat the experiment N times, using a different iteration point for each experiment, obtaining $X_1, \ldots, X_N$ different RVs. We point out that

—all the $X_i, i = 1 \cdots N$ have the same value of p;
—all the $X_i, i = 1 \cdots N$ are independent.

The variable $Y = \sum X_i$ represents the total number of misses for all N experiments. This new variable follows a binomial distribution Bin(N, p) [DeGroot 1998] and it is defined over all the iteration space. By generating random samples over the iteration space, we infer the total number of misses as shown in Section 2.2.2.

5.2 Generating Samples

Now, we discuss the methodology used to obtain samples. The key issues to create a good sample are

—it is important that all the population is represented in the sample;
—the size of the sample.

In our case, we have to keep in mind another constraint: the sample cannot have repeated iteration points (one iteration point cannot result in a miss twice).

To fulfill these requirements, we use *simple random sampling* [McCabe 1989]. The size of the sample is set according to the required width of the confidence interval and the desired confidence (see Section 2.2.2). Table II shows

Table II. Size of the Sample and Required Size of the Population for a Set of Accuracy Configurations

I. width	Confidence	#Points	Min. population
0.05	0.90	656	13120
	0.95	1082	21640
	0.99	2164	43280
0.10	0.90	164	3280
	0.95	270	5400
	0.99	541	10820
0.15	0.90	72	1440
	0.95	120	2400
	0.99	240	4800

Table III. Execution Time Required to Analyze the Matrix Multiply Algorithm for a 32 kB Cache; Interval Width = 0.05, Confidence = 95%

N	Miss %	Sim. time	CME time	Samp. time
1000	28.31	2 h 37 m	211 h	6 s
100	7.35	9 s	10 m	5 s
10	0.9	0.09 s	0.4 s	N/A

some possible values for those parameters. For each of them, we show the required size of the sample and the minimum size of the population according to the statistical requirements. If we ask for the best accuracy (i.e., the narrowest interval and the highest confidence), the size of the sample is about 2000 points, and the iteration space must exceed 43000 points. In most numeric programs loop nests have iteration spaces bigger than the values shown in the table. Otherwise, the iteration space is small enough to test all the iteration points in it.

5.3 Example

Let us recall our running example (see Figure 1). We illustrate the effectiveness of the sampling approach analyzing its cache behavior for a 32 kB direct-mapped cache. Table III shows the results obtained for an accuracy defined by an interval width of 0.05 and a 95% confidence.[10] For large problem sizes ($N =$ 1000), simulators are very slow and analyzing all the iteration points through the CMEs (see the classic analysis in the appendix) is not feasible, whereas we obtain the same results as the simulator[11] in a few seconds. For small problems ($N = 100$), sampling is faster than simulators. In the smallest one ($N = 10$), the size of the sample must be all the iteration space, which has the same low computation cost as the classic analysis due to the small number of points. For an extensive validation of our model against actual execution, see Section 6.3.3.

6. PERFORMANCE EVALUATION

Next we evaluate the accuracy of the proposed method and the speed/accuracy tradeoffs. We first compare to what extent our method is better than previous

[10]Our approach and the simulator run on a Sun UltraSparc I at 167 MHz.

[11]In this particular case, the central point of the confidence interval practically coincides with the simulation result.

Table IV. Properties of the Analyzed Loop Nests; #Ref, #R.V., and #Eq. Represent Average Values Across the Different Analyzed Loops

SPEC	#L	%VLN	#Ref.	#R.V.	#Eq.
tomcatv	9	100%	10.6	30.8	261
swim	22	59%	10.5	18.5	33
su2cor	13	84%	14.0	25.8	296.4
hydro2d	42	76.1%	7.1	11	196
mgrid	8	100%	29	627.5	70229
applu	55	100%	10.2	24.3	286.7
Average	18.6	77.25	13.6	123	11883.7

Note: L: loops; VLN: analyzed loop nests; Ref.: references; R.V.: reuse vectors; Eq.: equations.

approaches to remove empty polyhedra. Then we analyze the accuracy of the sampling technique. Finally, we report results showing the efficiency of our approach for analyzing the cache behavior.

6.1 Experimental Framework

The CMEs have been implemented for FORTRAN 77 codes through the Polaris Compiler [Padua et al. 1994] and the Ictineo library [Ayguadé et al. 1995]. These libraries allow us to obtain all the compile-time information needed to generate the equations.

The evaluation of CMEs has been implemented in C++ following the techniques outlined in the previous sections and using our own polyhedra representation.

Due to CMEs restrictions, only isolated perfect nested loops in which the array subscript expressions are affine functions of the induction variables are analyzed [Ghosh et al. 1999]. The loop nests considered are obtained from the SPECfp95 suite, choosing for each program the most time-consuming loop nests that in total represent between 60–70% of the whole execution time. Basically, for each program, we consider its loop nests, analyzing the references inside the loop as they were in an isolated loop nest. Each program is analyzed using the reference input data.

Four SPECfp95 programs have not been evaluated for the following reasons:

—*125.turb3d and 141.apsi*. The loop nests that represent the 65% of the execution have not enough iteration points to use sampling.

—*145.fpppp*. The section of the code that represents the main part of the execution time does not contain perfect nested loops.

—*146.wave5*. The array subscripts are functions of other arrays, and thus the CMEs cannot be obtained.

CMEs have been generated for a 32 kB cache of arbitrary associativity, with 32-B lines. Table IV shows the number of loops and the percentage of loop nests in which we can apply sampling.[12] The "L" entry lists the number of analyzable

[12]At least 200 points must be tested.

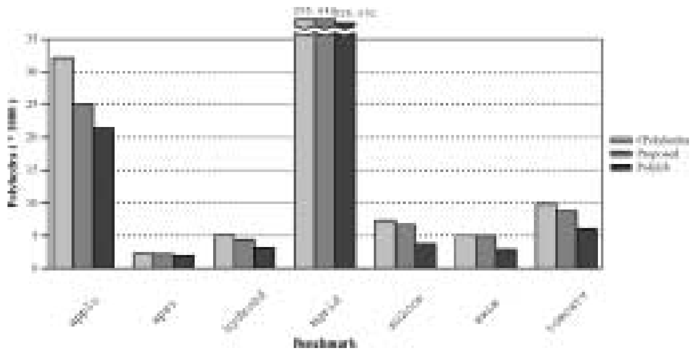


Fig. 7. Empty polyhedra for the different analyzed SPECfp95.

loops, whereas the third column presents the percentage of associated loop nests that can be analyzed using our sampling technique.

The next three columns illustrate the problem size. The table shows that on average, each loop nest contains 13.6 references and 123 reuse vectors. In addition, the number of equations per loop nest for a direct-mapped cache is 11883.7. Since the number of misses depends on the organization of the cache, so does the number of equations.

The simulation values are obtained using a trace-driven cache simulator.[13] The traces are obtained by instrumenting the program with Ictineo [Ayguadé et al. 1995]. For the evaluation of the execution time, a Sun UltraSparc I running at 167 MHz has been used.

6.2 Empty Polyhedra

In this section we evaluate the effectiveness of our proposal for detecting empty polyhedra. We assume a direct-mapped cache, since the polyhedra generated for a k-way set-associative cache of size C are the same as the ones generated for a direct-mapped cache of size $\mathbb{C} = \frac{C}{k}$ (see Section 4.4).

Figure 7 compares our method with the technique implemented in the Polylib [Wilde 1993] for detecting empty polyhedra. Only replacement polyhedra have been considered, as their evaluation is the most time consuming among all CMEs polyhedra. The first column shows the number of replacement polyhedra obtained for each program. The second column depicts the number of empty polyhedra detected by our approach, whereas the third column shows the number of empty replacement polyhedra detected through Polylib. We can see that our approach detects a significantly higher number of empty polyhedra. This is due to the fact that Polylib, which is a general-purpose library, only detects real empty polyhedra.[14]

Columns 1 and 2 of Table V show the execution times required by both methods to check the emptiness of all polyhedra. Due to the complexity of the computation of the vertices of a polyhedron, our proposal is much faster than

[13]A locally written simulator has been used in all our experiments. It has been validated over the years against Dinero III trace-driven simulator [Hill n.d.].
[14]It checks whether the real domains (see Section 2.1) are empty.

Table V. Execution Time (in Seconds) to Detect Empty Polyhedra

SPEC	Empty polyhedra	
	Proposed	Polylib
applu	36.70	1933.87
apsi	1.30	17.31
hydro2d	3.16	47.51
mgrid	235.75	5495.10
su2cor	3.08	34.27
swim	3.00	48.71
tomcatv	9.60	280.75
total	292.59	7857.52

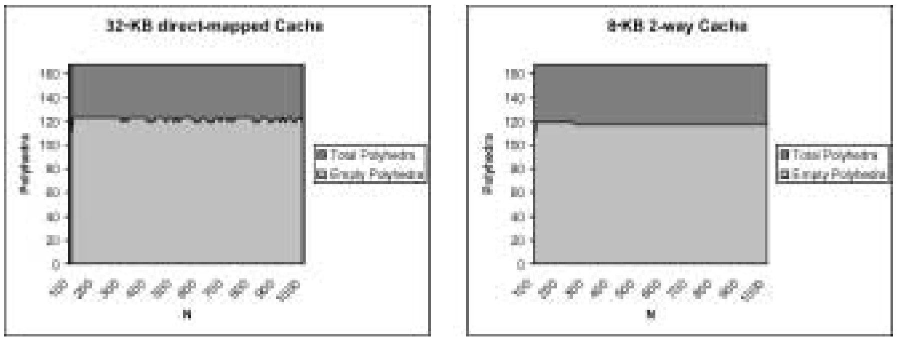


Fig. 8. Distribution of empty polyhedra when analyzing our running example (Figure 1) for different problem sizes.

Polylib's technique; the complexity of the proposed method is $O(m)$. On the other hand, Polylib's method relies on computing the vertices of each polyhedron. The complexity of its algorithm is $O(\#constraints^{\lfloor \frac{\#variables}{2} \rfloor})$. For replacement polyhedra, the number of constraints is $2m + 3$ and the number of variables is $m + 1$, where m is the nesting depth of the loop nest. Thus, the complexity of Polylib's approach is $O(m^{\lfloor \frac{m}{2} \rfloor})$.

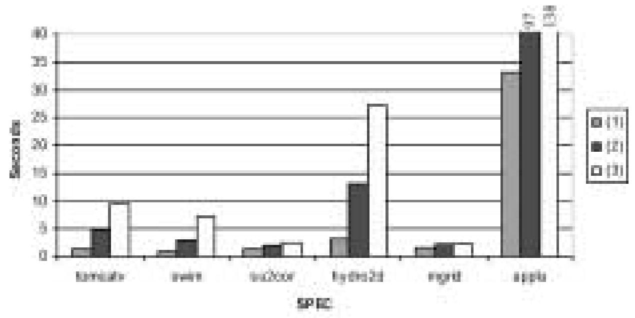
Finally, Figure 8 shows the number of empty polyhedra detected by our method when analyzing the matrix multiply algorithm (see Figure 1). We present the results for two different cache configurations: (32-kB, 32-B) direct-mapped cache and a (8-kB, 64-B) two-way set-associative cache. In each graph, we plot the number of polyhedra against the size of the matrices. Note that the total number of polyhedra only depends on the reuse vectors and the number of references. Since we are studying the same program with different problem sizes, the total number of polyhedra is the same.

We can observe that the number of empty polyhedra relates to the miss ratios shown in Figure 11. For the 8-kB cache the number of empty polyhedra is almost the same for all problem sizes, and so are the miss ratios. Miss ratios fluctuate a bit more for the 32-kB cache, and so does the number of empty polyhedra. Those drops in empty polyhedra correspond to the peaks in miss ratios in Figure 11.

Table VI. Analyzed Loops

SPEC	(1)	(2)	(3)
tomcatv	9	9	9
swim	13	13	13
su2cor	11	11	10
hydro2d	32	32	32
mgrid	8	8	6
applu	55	51	51

Note: (1) stands for $\alpha = 0.05$ and interval width = 0.10; (2) stands for $\alpha = 0.05$ and interval width = 0.05; (3) stands for $\alpha = 0.01$ and interval width = 0.05.



	I width	Confidence
(1)	0.10	95%
(2)	0.05	95%
(3)	0.05	99%

Fig. 9. Execution time for different accuracies.

6.3 Accuracy

In this section we analyze the accuracy of our method for predicting cache miss ratios. We first analyze the tradeoffs between accuracy and analysis time. In order to check the accuracy of our approach, we compare against a simulator that simulates exactly the same memory accesses. Finally, we show the accuracy of our model comparing the predicted average stall times due to cache misses with that from real executions.

6.3.1 *Sampling Accuracy.* We have experimented with different accuracy configurations. Table VI shows the different configurations (from less accurate to more accurate) and the number of loops for which sampling can be used. For more accurate configurations fewer loops can be analyzed using sampling because there are not enough points in the iteration space. In these cases the whole iteration space must be traversed. However, this situation arises only for small iteration spaces, which can be fully analyzed in a reasonable time.

Figure 9 shows the time in seconds required to analyze the SPECfp95 programs for the three accuracy configurations for a (32-kB, 32-B) direct-mapped cache. Note that the analysis time is reasonable in all cases and that the more accurate the configuration is, the more time is required since more points are

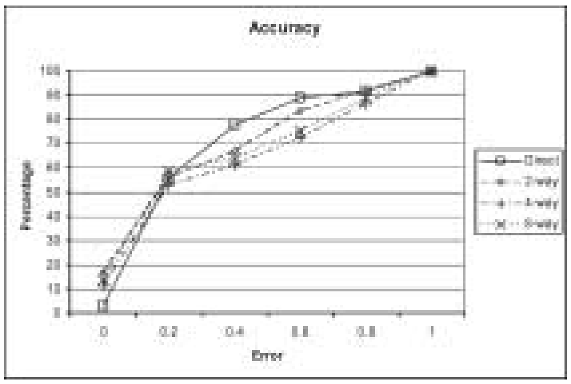


Fig. 10. Sampling error for different cache configurations.

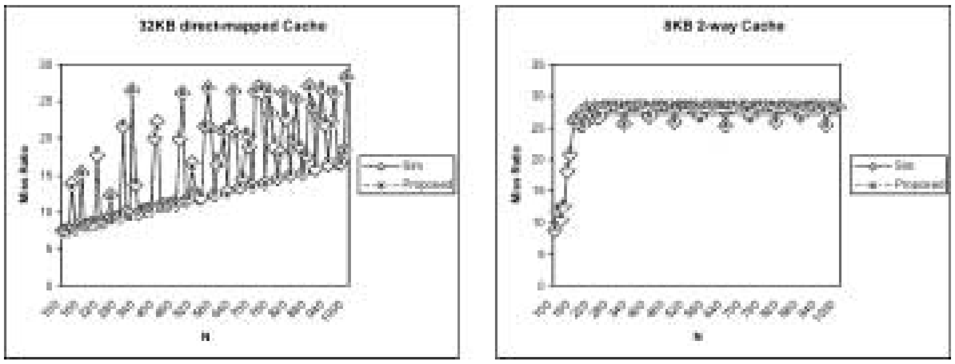
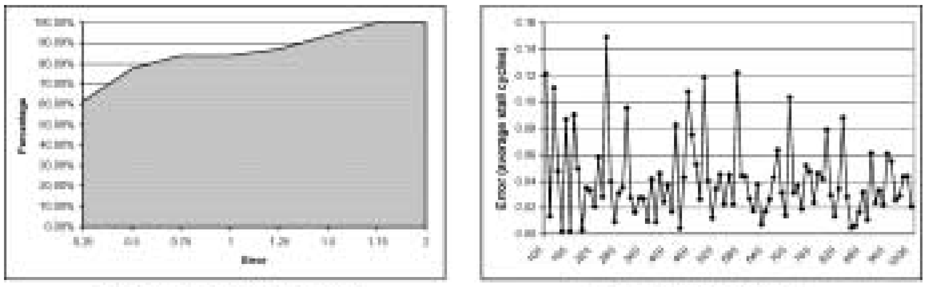


Fig. 11. Predicted and simulated miss ratios for our running example.

considered. From our experiments we observe that a confidence of 95% and an interval of 0.05 is a good tradeoff between analysis time and accuracy.

6.3.2 *Comparison with Simulation.* We depict in Figure 10 the cumulative distribution of the difference between the miss ratio and the central point of the confidence interval (also known as *empirical estimate*) for all the programs, with a 95% confidence and an interval width of 0.05. The Y-axis represents the percentage of loop nests that has an error less than or equal to the corresponding value in the X-axis. This graph shows that the absolute differences between the actual miss ratio and the miss ratio obtained from our analyzer is usually lower than 0.2 and never higher than 1 (i.e., the actual miss ratio is always in the interval $[(x-1)\%, (x+1)\%]$ where x is the central point of the confidence interval) for all the different cache configurations.

Finally, Figure 11 compares the predicted miss ratios against those from simulation for two different cache configurations: (32-kB, 32-B) direct-mapped cache and a (8-kB, 64-B) two-way set-associative cache. In all experiments, the predicted miss ratios are very close to the simulated ones, which shows the accuracy of our approach when varying the problem size (the average absolute errors are 0.30 in both cases).



(a) Kernels from SPECfp95 (b) Running example

Fig. 12. Average stall time due to L1 misses.

6.3.3 *Comparison with Real Execution.* In order to evaluate the ability of our model to predict actual cache miss ratios, we have run all our benchmarks on a modern out-of-order processor and quantified the error between predicted and actual cache miss ratios. For the sake of concreteness, we have compiled all codes with “g77 -O3” on a Pentium-4 machine running Linux, and obtained the actual accesses and misses using the hardware counters.

We present results in terms of average stall time per memory access. Considering that an access to L2 cache takes 24 cycles, we set up the following formula [Ailamaki et al. 1999]:

$$avg\ stall\ cycles = \frac{\#misses}{\#accesses} \times 24\ cycles.$$

Figure 12(a) shows the cumulative distribution of the difference between the predicted average stall time and the actual one for all programs. The Y-axis represents the percentage of loop nests that has an error less than or equal to the corresponding value in the X-axis. We can observe that the absolute differences between the predicted average stall times and the actual ones are less than 0.75 cycles for 84% of the loops, and never higher than 1.75 cycles. The larger errors are due to extra memory accesses that are not currently considered in our compiler Ictineo, such as spill code and stack accesses. Figure 12(b) compares the predicted average stall time against that from execution for our running example. We show the absolute errors. In all experiments, the predicted average stall times are very close to the actual ones (the average error is 0.04, while the largest one is 0.15). This confirms the accuracy of our approach when the memory references are known.

6.4 Execution Time: Analyzing Iteration Points

Next, we evaluate the effectiveness of the proposed technique to determine whether an iteration point results in a miss. We compare it with an algorithm that counts the number of integer points inside the polyhedra by means of the general method for counting presented in Section 4.4.3. The computation of the vertices of the polyhedra needed for this second method (*Vertices*) is done by means of functions from the Polylib library.

Table VII shows the time in seconds required to analyze the different SPECfp95 for four different organizations of set-associative caches, for both

Table VII. Execution Time for Different Cache Organizations

		Applu	Hydro2d	Mgrid	Su2cor	Swim	Tomcatv
(1)	1 way	99.63 s	12.21 s	1.89 s	1.91 s	2.18 s	4.57 s
	2 way	89 s	12 s	2.36 s	2 s	4 s	5.2 s
	4 way	91 s	12.8 s	7 s	2.02 s	7.7 s	8.4 s
	8 way	97 s	14.19 s	14.19 s	2.17 s	15 s	15 s
(2)	1 way	18 m 48.55 s	1 m 26.23 s	9 m 6.67 s	19.18 s	21.22 s	31 m 51 s
	2 way	6 m 44.92 s	1 m 53.27 s	9 m 27.35 s	27.6 s	26.66 s	34 m 23.73 s
	4 way	6 m 39.62 s	2 m 0.33 s	13 m 32.59 s	17.77 s	48.32 s	1 h 8 m 24 s
	8 way	6 m 37.85 s	1 m 52.84 s	18 m 46.30 s	15.30 s	1 m 29.6 s	1 h 7 m 25 s
speedup 1 way		11.7	7	283	10	9.7	418.1

Note: (1) proposed; (2) vertices.

the proposed method and the *Vertices* method, with a 95% confidence and an interval width of 0.05.

The speedup of our approach is very important, due to the different complexities of both algorithms. For a direct-mapped cache, it is between 7 and 418 times faster than the *Vertices* method and 30 times faster on average. The speedup for different set-associative configurations is even higher. For instance, the average speedup for a four-way set-associative cache is 42.

The difference between these two algorithms relies on the approach to compute the domains of all variables. The proposed method does it with a complexity of $O(m^2)$. The *Vertices* method is split in two steps: first the vertices of the polyhedron are computed with a complexity of $O(m^{\lfloor \frac{m}{2} \rfloor})$, as explained in the previous subsection. Then, the domains of the variables are computed with a complexity of $O(m * \#vertices)$ by means of the vertices.

Note that most programs can be analyzed by the proposed approach in less than a minute, and the most expensive one is *Applu*, which takes about 1.5 minutes, whereas the approach based on the *Vertices* method takes several minutes and in the worst case it takes more than 1 hour.

7. RELATED WORK

There are different approaches to analyze data locality which provide different tradeoffs between: accuracy, speed, flexibility (i.e., adaptability to different memory configurations), and information provided.

Memory simulation techniques are very accurate and flexible and can provide rich information. They are usually based on trace-driven simulation [Kennedy et al. 1990; Goldberg and Hennessy 1991; MIPS 1988; Sugumar 1993; Gee et al. 1993; Magnusson 1993; Goldschmidt and Hennessy 1993; Bedichek 1995; McKinley and Temam 1996; van der Deijl et al. 1997]. However, these techniques may demand a lot of space to store traces and are very slow (typical slowdowns are several orders of magnitude). For instance, the slowdown exhibited by all simulators surveyed in Uhlig and Mudge [1997] is in the range of 45 to 6250.

There are some innovative methods that have been proposed with the objective of reducing the exhibited slowdown [Martonosi et al. 1992; Lebeck and

Wood 1994; Witchel and Rosenblum 1996]. However, these methods provide little information (usually only miss ratios), trading information for speed.

Martonosi et al. [1993] introduced the use of trace sampling techniques in order to further reduce the overhead of such simulators (the slowdown shown is between 3 and 8). They take samples from the full reference trace so that they are representative of the full trace. The sizes of the samples as well as the number of samples depend on both the cache that is analyzed and the characteristics of the program being traced. Even though the results present a good degree of accuracy, neither the error can be chosen, nor the sample process can be set, to achieve a degree of accuracy. In addition, when sampling is applied to simulators, inaccuracy can result from the unknown state of the cache at the beginning of the sample.

There are other tools based on hardware counters (e.g., Ammons et al. [1997]) provided by some microprocessors. These tools are fast and accurate. However they have no flexibility since they can only be used to analyze the memory architecture of the actual microprocessor. In addition they provide a limited set of results depending on the particular counters provided by a particular machine. Information like conflict misses between two particular memory references cannot be obtained with current hardware counters.

Analytical models describe cache behavior by means of mathematical formulas that specify cache misses. Temam et al. [1994] computed footprints to estimate cache misses of isolated perfectly nested loops for direct-mapped caches. Fraguela et al. [1999] used a probabilistic method to estimate cache miss ratios for set-associative caches. While allowing imperfect nests, they only analyzed reuse among references in the same nest. These references form a subset of the uniformly generated references analyzed by the CMEs. Recently, Chatterjee et al. [2001] introduced a new model for exactly analyzing the cache behavior of loop nests for set-associative caches. They derived formulas for imperfect nested loops, and dealt with some IF statements. However, the current implementation is yet far from being practical: they discussed matrix-vector product, presenting a formula when $N = 100$, but they have not solved it.

Ghosh et al. [1999] introduced the cache miss equations to specify the cache behavior of a single perfect nested loop. They used Wolf and Lam [1991] reuse vectors to express locality. They discussed the use of sampling [Ghosh et al. 2000] to speed up the process of solving them. Instead of using confidence intervals, they analyzed a fraction of the iteration space. While improving the performance of the solver, they used the rule of thumb to decide the size of the sample, and they could not choose the degree of accuracy.

Static analysis techniques have limited accuracy due to unknown information at compile time. For instance, unknown loop bounds or unknown initial addresses of data structures can degrade the accuracy of the results.

A solution to this problem is to use hybrid techniques such as SPLAT [Sánchez and González 1998]. SPLAT is a static analysis technique improved with some profile (dynamic) information. This hybrid technique is fast and flexible and can provide much different information like other static techniques. In addition it is accurate because a profiling provides the information unknown at compile time. The use of hybrid techniques is not restricted to

SPLAT; other static approaches (like CMEs) can also improve their accuracy by using profile data.

A typical cause for lack of accuracy in static or hybrid techniques is the simplifications in the analysis. For instance, SPLAT is not capable of analyzing interferences in applications with complex interference patterns and can only analyze direct mapped caches. Cache miss equations [Ghosh et al. 1999] provide very precise information about cache behavior. If combined with profile information, they can be as accurate as simulators and can provide the same information and be as flexible as other static approaches. Unfortunately, solving CMEs is an NP-complete problem that makes them slower than simulators.

7.1 Some Applications of the CMEs

The effectiveness of memory hierarchy is critical for the performance of current processors. The proposed approach can analyze the locality of most SPECfp95 programs in just a few seconds, which allows using this analysis for guiding optimization steps of a compiler or an interactive program transformation tool. Recent implementations of padding [Vera et al. 2002] and tiling [Abella et al. 2002] are examples of such possible optimizations. A genetic algorithm is used to compute tile and pad factors that enhance the program behavior. CMEs are used to evaluate each combination of parameter values. Results show that they can remove practically all replacement misses among variables in the SPECfp95 suite, targeting all the different cache levels simultaneously.

Clustering is an approach that many microprocessors are adopting in recent times in order to mitigate the increasing penalties of wire delays. Sánchez and González [2000] proposed a novel clustered architecture with a partitioned cache memory. They presented a modulo scheduling scheme which takes into account memory intercluster communication, making use of our approach in order to have a schedule that favors cluster locality in cache references. For instance, given a memory instruction, it may be beneficial to schedule it in a cluster where there are already other instructions from which it reuses data.

Nevertheless, many steps have to be done in order to statically analyze whole programs. Recently, Vera and Xue [2002] presented a new approach that deals with imperfect loop nests, call statements, and IF conditionals. Based on a new characterization of reuse vectors, they made use of the techniques presented in this work to obtain a feasible tool that analyzes whole programs. Thus, our techniques are not limited to perfect loop nests but can be used in other approaches where CME polyhedra are used. However, data-dependent constructs and indirection arrays still represent an interesting and challenging future work. We plan to investigate locality techniques to analyze them.

8. CONCLUSIONS

Cache miss equations provide an analytical and precise description of the cache memory behavior. The main drawback of CMEs is that solving them to know the exact number of misses is an NP-complete problem that makes them infeasible for most applications.

In this paper we propose the use of both mathematical and statistical techniques to solve CMEs. With these techniques we can perform memory analysis extremely fast, regardless of the size of the iteration space. For instance, it takes the same time (6 seconds) to analyze a matrix multiplication loop nest of size 100×100 as to analyze one of size 1000×1000. On the contrary, it takes 9 seconds to simulate the first case and more than 2 hours to simulate the second.

The use of sampling along with inference theory allows the user to set the desired accuracy by means of the confidence and the width of the interval. The larger the accuracy, the bigger the set of points to analyze and therefore the more time required to perform the analysis. This way the user can trade accuracy for speed at its will. In our experiments we have found that, using a confidence of 95% and an interval width of 0.05, the absolute error in miss ratio was smaller than 0.2% in 65% of the loops from the SPECfp95 programs and was never larger than 1.0%. Furthermore, the analysis time for each program was usually just a few seconds and never more than 2.3 minutes.

Some mathematical techniques are proposed that exploit some intrinsic properties of the particular polyhedra generated by CMEs. These techniques significantly reduce the complexity of the algorithms and result in speedups of more than one order of magnitude for the SPECfp95 benchmarks. The average speedup for all these benchmarks is 17.94.

Overall, the proposed approach can analyze the locality of most SPECfp95 programs in just several seconds, which allows the use of this analysis in order to guide optimization steps of a compiler or an interactive program transformation tool. Padding [Vera et al. 2002] and tiling [Abella et al. 2002] are examples of such possible optimizations.

APPENDIX: SOLVING CMEs

CMEs contain precise information about the cache behavior, but obtaining this information from the equations is not a trivial problem. In this section we present two methods [Ghosh et al. 1999] to solve CMEs. A direct-mapped cache is now assumed. The application of these methods when an associative configuration is considered will be discussed later.

Each equation represents a convex polyhedron in $\mathbb{R}^n$ (see Section 2.1), where n depends on the type of equation. The integer points inside each convex polyhedron represent the potential cache misses. This leads us to consider several ways for computing them:

A.1 Analytical Method

In this section we give an analytical description of the solution set of the CMEs. This solution set represents the cache misses, and its volume the number of misses.

THEOREM 1. *The set of all misses of a reference along a reuse vector is given by the union of all the solution sets of the equations corresponding to that reuse vector.*

Given a reference and a reuse vector, an iteration point results in a miss if it is either a cold or a replacement miss.

THEOREM 2. *The set of all miss instances of a reference is given by the intersection of all the miss-instance sets along the reuse vectors.*

Given a reference, an iteration point results in a hit if it exploits the locality of at least one of the reuse vectors.

Thus, given a reference R with m reuse vectors and n_k equations for the kth reuse vector, the polyhedron that contains all the iteration points that result in a miss is [Ghosh et al. 1999]

$$Set\_Misses = \cap_{k=1}^{m} \cup_{j=1}^{n_k} Solution\_Set\_Equation_j$$

For this approach we need to count the number of points inside the polyhedra, which is an NP-complete problem. Writing the set of misses as a function of the complementary sets, we have that

$$Set\_Misses = \cup_{j=1}^{n_k} \cap_{k=1}^{m} Solution\_Set\_Equation_j^C$$

where $Solution\_Set\_Equation_j^C$ represents the set of all points in $\mathbb{Z}^n$ that are not solution to the jth equation.

Note that in general, the union of convex polyhedra is not convex. In order to count the number of integer points inside the set of misses described above, the set of misses must be expressed as the union of disjoint convex polyhedra. Thus, according to measure theory, the union of s sets can be computed as follows:

$$\begin{aligned}\mu\big(\cup_{i=1}^{s} A_i\big) = {} & \mu(A_i) + \cdots + \mu(A_s) \\ & - \sum_{i \neq j} \mu(A_i \cap A_j) \\ & + \sum_{i \neq j \neq k} \mu(A_i \cap A_j \cap A_k) \\ & + \cdots \\ & + (-1)^{(i-1)} \mu\big(\cap_{i=1}^{s} A_i\big),\end{aligned}$$

where $\mu(P)$ is the number of points inside polyhedron P. As the expression shows, the number of polyhedra that must be counted is 2^s, making this problem infeasible due to its huge computing time.

Furthermore, this method does not work for set-associative caches. We are interested in counting the number of points from the iteration space that potentially results in a miss. In particular, we want to know how many iteration points $\vec{i} = (i_1, \ldots, i_m)$ verify the replacement equation. Since an iteration point verifies a set of replacement equations if there exists any integer combination of the variables $j_1, \ldots, j_m, \mathrm{n}, \mathrm{z}$ so that the point $(i_1, \ldots, i_m, j_1, \ldots, j_m, \mathrm{n}, \mathrm{z})$ belongs to the polyhedron defined by these equations, the number of combinations of the variables $j_1, \ldots, j_m, \mathrm{n}, \mathrm{z}$ is not important. For this reason, not all the points inside a replacement polyhedron have to be counted.

A.2 Traversing the Iteration Space

A big advantage of CMEs over simulators is that the behavior of every iteration point can be studied independently from the rest of the iteration space, whereas simulators need to have processed previous iteration points to get to know what happens at a certain iteration point. The method we present next is based on this fact.

Given a reference, all iteration points are tested independently. We study the equations in order: from the equations generated for the shortest reuse vector to the equations generated for the longest one.

Next, we give an intuitive description of the algorithm.

Let us consider the reference R. For this reference several CMEs are generated for every reuse vector. The reuse vectors are studied in a lexicographical[15] ascendent order. After one reuse vector has been treated, some iteration points will be identified as resulting in a miss or a hit. Others might rest undetermined. These are the points that will have to be studied when considering the next reuse vector.

Given a reference and a reuse vector, the iteration points are studied as follows:

—If an iteration point is a solution to a cold CMEs, the reuse along $\vec{r}$ is not realized in this iteration point, but we cannot take any definitive decision about the character of this iteration point until all reuse vectors have been studied. Therefore, this point is considered as undetermined.

—On the other hand, if an iteration point is not a solution to any of the cold CMEs, it is declared as a miss if it is a solution to a replacement equation, and as a hit if it is no solution to any equation either.

Note that the set of undetermined points will generally decrease when treating new reuse vectors. The algorithm stops when all iteration points have been characterized.

A.3 Set-Associative Caches

CMEs give an analytical precise description of the cache memory behavior for both direct-mapped and set-associative caches. Note that cold misses are not influenced by the associativity of the cache. Therefore, in this subsection we will focus on replacement equations.

Although the form of the replacement equations is not affected by the considered configuration, the way of interpreting them is different in each case.

When considering a k-way set-associative cache, a replacement miss occurs at the iteration point $\vec{i_0}$ if there exist k integer combinations of the variables $j_1, \ldots, j_m, \mathtt{n}, \mathtt{z}$, $(j_1^s, \ldots, j_m^s, \mathtt{n}^s, \mathtt{z}^s)$, $s \in \{1, \ldots, k\}$ such that $n^r \neq n^s \iff r \neq s$ which makes $(i_{01}, \ldots, i_{0m}, j_1^s, \ldots, j_m^s, n^s, z^s)$ belong to the replacement polyhedra[16] for all $s \in \{1, \ldots, k\}$.

[15]If iteration $\vec{i_2}$ executes after $\vec{i_1}$ we say that $\vec{i_2}$ is lexicographically greater than $\vec{i_1}$.

[16]The replacement polyhedra that model the behavior of the reference that is being studied.

The analytical method presented to solve CMEs does not allow us to make any distinction on the points we count. Therefore, it only works for direct-mapped caches.

On the other hand, the second approach can be used for both direct-mapped and set-associative organizations.

ACKNOWLEDGMENTS

We would like to thank Manel Fernández and Toni Palau for their useful comments and reviews. We would also like to thank the anonymous referees for providing helpful comments on earlier drafts of this paper.

REFERENCES

ABELLA, J., GONZÁLEZ, A., LLOSA, J., AND VERA, X. 2002. Near-optimal loop tiling by means of cache miss equations and genetic algorithms. In *Proceedings of 31st International Conference on Parallel Processing* (ICPP'02) .

AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND WOOD, D. A. 1999. DBMSs on a modern processor: where does time go? In *Proceedings of the 25th VLDB Conference* (Edinburgh, Scotland).

AMMONS, G., BALL, T., AND LARUS, J. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'97). 85–96.

AYGUADÉ, E. ET AL. 1995. A uniform internal representation for high-level and instruction-level transformations. Tech rep. UPC-DAC-95-02. Universitat Politècnica de Catalunya, Barcelona, Spain.

BANERJEE, U. 1988. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA.

BANERJEE, U. 1993. *Loop transformations for restructuring compilers: The Foundation*. Kluwer Academic Publishers, Norwell, MA.

BEDICHEK, R. 1995. Talismam: Fast and accurate multicomputer simulation. In *Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS'95). 14–24.

CARR, S. AND KENNEDY, K. 1992. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing* (SC'92). 114–124.

CARR, S., MCKINLEY, K., AND TSENG, C.-W. 1994. Compiler optimizations for improving data locality. In *Proceedings of the VI International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS'94). 252–262.

CHATTERJEE, S., JAIN, V. V., LEBECK, A. R., MUNDHRA, S., AND THOTTETHODI, M. 1999. Nonlinear array layout for hierarchical memory systems. In *Proceedings of the ACM International Conference on Supercomputing* (Rhodes, Greece) (ICS'99). 444–453.

CHATTERJEE, S., PARKER, E., HANLON, P. J., AND LEBECK, A. R. 2001. Exact analysis of the cache behavior of nested loops. In *ACM SIGPLAN '01 Conference on Programming Language Design and Implementation* (PLDI'01). 286–297.

CLAUSS, P. 1996. Counting solutions to linear and non-linear constraints through Ehrhart polynomials. In *Proceedings of ACM International Conference on Supercomputing* (Philadelphia) (ICS'96). 278–285.

COLEMAN, S. AND MCKINLEY, K. S. 1995. Tile size selection using cache organization and data layout. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'95). 279–290.

DEGROOT, M. 1998. *Probability and Statistics*. Addison-Wesley, Reading, MA.

FEAUTRIER, P. 1996. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, G. R. Perrin and A. Darte, Eds. Lecture Notes in Computer Science, vol. 1132. Springer-Verlag, Berlin, Germany, 79–103.

Fraguela, B. B., Doallo, R., and Zapata, E. L. 1999. Automatic analytical modeling for the estimation of cache misses. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (PACT'99).

Gannon, D., Jalby, W., and Gallivan, K. 1988. Strategies for cache and local memory management by global program transformations. *J. Parallel. Distrib. Comput. 5*, 587–616.

Gee, J., Hill, M., Pnevmatikatos, D., and Smith, A. 1993. Cache performance of the spec92 benchmark suite. *IEEE Micro 13*, 4 (Aug.), 17–27.

Ghosh, S. 1999. Compiler analysis framework for tuning memory behavior. Ph.D. dissertation. Princeton University, Princeton, NJ.

Ghosh, S., Martonosi, M., and Malik, S. 1998. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS'98). 228–239.

Ghosh, S., Martonosi, M., and Malik, S. 1999. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Trans. Programm. Lang. Syst. 21*, 4, 703–746.

Ghosh, S., Martonosi, M., and Malik, S. 2000. Automated cache optimizations using CME driven diagnosis. In *Proceedings of the International Conference on Supercomputing* (ICS'00). 316–326.

Goldberg, A. and Hennessy, J. 1991. Performance debugging shared memory multiprocessor programs with mtool. In *Proceedings of Supercomputing* (SC'91). 481–490.

Goldschmidt, S. and Hennessy, J. 1993. The accuracy of trace-driven simulation of multiprocessors. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS'93). 146–157.

Haghighat, M. R. and Polychronopoulos, C. D. 1993. Symbolic analysis: A basis for parallelization, optimization and scheduling of programs. In *1993 Workshop on Languages and Compilers for Parallel Computing (LCPC'93)*. Springer Verlag, Portland, Ore., 567–585.

Hill, M. n.d. *DineroIII: a uniprocessor cache simulator (http://www.cs.wisc.edu/˜larus/warts.html).*

Kandemir, M., Choudhary, A., Banerjee, P., and Ramanujam, J. 1999. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Trans. Parallel Distrib. Syst. 10*, 2 (Feb.), 115–135.

Kennedy, K., Callahan, D., and Porterfield, A. 1990. Analyzing and visualizing performance of memory hierarchy. In *Instrumentation for Visualization*. ACM Press, New York, NY.

Kreisel, G. and Krevine, J. L. 1967. *Elements of Mathematical Logic*. North-Holland, Amsterdam, The Netherlands.

Lam, M., Rothberg, E. E., and Wolf, M. E. 1991. The cache performance of blocked algorithms. In *Proceedings of the IV International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS'91).

Lebeck, A. and Wood, D. 1994. Cache profiling and the spec benchmarks: A case study. *IEEE Comput. 27*, 10 (Oct.), 15–26.

Magnusson, P. 1993. A design for efficient simulation of a multiprocessor. In *Proceedings of the Western Simulation Multiconference on International Workshop on MASCOTS-93*. (La Jolla, CA). 69–78.

Martonosi, M., Gupta, A., and Anderson, T. 1992. Memspy: Analyzing memory system bottlenecks in programs. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS'92). 1–12.

Martonosi, M., Gupta, A., and Anderson, T. 1993. Effectiveness of trace sampling for performance debugging tools. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS'93).

McCabe, M. 1989. *Introduction to the Practice of Statistics*. Freeman & Co., New York, NY.

McKinley, K. S. and Temam, O. 1996. A quantitative analysis of loop nest locality. In *Proceedings of the VII Int. Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS'96).

MIPS. 1988. *RISCompiler Languages Programmer's Guide*. MIPS Computer Systems, Sunnyvale, CA.

Mowry, T., Lam, M., and Gupta, A. 1992. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the V International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS'92). 62–73.

Padua, D. et al. 1994. *Polaris Developer's Document*. Available online at http://polaris.is.uiuc.edu/polaris/polaris–developer/polaris–developer.html.

Pugh, W. 1991. The omega test: A fast and practical integer programming algorithm for dependence analysis. *Proceedings of the ACM/IEEE Conference of Supercomputing* (SC'91) (Albuquerque, NM). 4–13.

Pugh, W. 1994. Counting solutions to presburguer formulas: How and why. In *Proceedings of the International Conference on Programming Language Design and Implementation* (PLDI'94).

Rivera, G. and Tseng, C.-W. 1998. Data transformations for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'98). 38–49.

Rivera, G. and Tseng, C.-W. 1999. A comparison of compiler tiling algorithms. In *Proceedings of the 8th International Conference on Compiler Construction* (CC'99).

Sánchez, F. and González, A. 1998. Fast, flexible and accurate data locality analysis. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (PACT'98).

Sánchez, F. and González, A. 2000. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *Proceedings of the International Symposium on Microarchitecture* (MICRO-33).

Sugumar, R. 1993. Multi-configuration simulation algorithms for the evaluation of computer designs. Ph.D. thesis, University of Michigan.

Temam, O., Fricker, C., and Jalby, W. 1994. Cache interference phenomena. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS'94). 261–271.

Temam, O., Granston, E., and Jalby, W. 1993. To copy or not to copy: A compile-time technique for accessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing* (SC'93). 410–419.

Uhlig, R. A. and Mudge, T. N. 1997. Trace-driven memory simulation: a survey. *ACM Comput. Surv. 29*, 3 (Sept.), 128–170.

van der Deijl, E., Kanbier, G., Temam, O., and Granston, E. 1997. A cache visualization tool. *IEEE Comput. 30*, 7 (July), 71–78.

Vera, X., Llosa, J., and González, A. 2002. Near-optimal padding for removing conflict misses. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computers* (LCPC'02).

Vera, X. and Xue, J. 2002. Let's study whole program cache behaviour analytically. In *Proceedings of the International Symposium on High-Performance Computer Architecture* (Cambridge, U.K.). (HPCA 8).

Wilde, D. 1993. A library for doing polyhedral operations. Tech. rep. 785, Oregon State University.

Witchel, E. and Rosenblum, M. 1996. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS'96).

Wolf, M. and Lam, M. 1991. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'91). 30–44.

Received January 2002; revised August 2002, May 2003; accepted June 2003