Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior

SOMNATH GHOSH, MARGARET MARTONOSI and SHARAD MALIK
Department of Electrical Engineering, Princeton University
{sghosh,martonosi,sharad}@ee.princeton.edu

With the ever-widening performance gap between processors and main memory, cache memory, used to bridge this gap, is becoming more and more significant. Caches work well for programs that exhibit sufficient locality. Other programs, however, have reference patterns that fail to exploit the cache, thereby suffering heavily from high memory latency. In order to get high cache efficiency and achieve good program performance, efficient memory accessing behavior is necessary. In fact, for many programs, program transformations or source-code changes can radically alter memory access patterns, significantly improving cache performance. Both hand-tuning and compiler optimization techniques are often used to transform codes to improve cache utilization. Unfortunately, cache conflicts are difficult to predict and estimate, precluding effective transformations. Hence, effective transformations require detailed knowledge about the frequency and causes of cache misses in the code.

This paper describes methods for generating and solving Cache Miss Equations (CMEs) that give a detailed representation of cache behavior, including conflict misses, in loop-oriented scientific code. Implemented within the SUIF compiler framework, our approach extends traditional compiler reuse analysis to generate linear Diophantine equations that summarize each loop's memory behavior. While solving these equations is in general difficult, we show that is also unnecessary as mathematical techniques for manipulating Diophantine equations allow us to relatively easily compute and/or reduce the number of possible solutions, where each solution corresponds to a potential cache miss. The mathematical precision of CMEs allows us to find true optimal solutions for transformations like blocking or padding. The generality of CMEs also allows us to reason about interactions between transformations applied in concert. The paper gives examples of their use to determine array padding and offset amounts that minimize cache misses, and also to determine optimal blocking factors for tiled code. Overall, these equations represent an analysis framework that offers the generality and precision needed for detailed compiler optimizations.

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: Experimentation, Design, Performance

Additional Key Words and Phrases: Cache Memories, Compilation, Optimization, Program Transformation

Authors' address: Dept. of Electrical Engineering, Princeton University, Princeton, NJ 08540

1. INTRODUCTION

Data caches are widely used to bridge the cycle-time gap between fast microprocessors and relatively slow main memories. But, over the past two decades, the speed of processors has increased at a much faster rate than that of main memory. As this disparity between processor and main-memory speed increases—by approximately 50 percent per year—cache performance becomes increasingly critical. Although caches generally work well, some programs fail to use them effectively. In fact, programs without sufficient locality in their access patterns spend a significant portion of their execution time transferring data between main memory and cache. Programmers often hand-tune their code in order to improve its memory behavior, but this process can be time-consuming and error-prone. In other cases, automatic compiler transformations can improve memory behavior and reduce the programmer's burden. Either way, programmers or compilers need detailed, accurate assessments of when and why cache misses occur. Prior approaches for analyzing cache behavior have been based either on simulation, which can be slow, or on compiler heuristics which can be imprecise. In this paper we present a precise mathematical framework that can be used to guide a range of memory optimizations.

There has been extensive research on improving the cache performance of numerical programs [McKinley et al. 1996; Ferrante et al. 1991; Lam et al. 1991; Wolf and Lam 1991; Wolfe 1989]. Most of this work targets loop nests with predictable and regular data accesses. Loop optimization plays a significant role in compiler optimization as scientific programs spend a considerable amount of time processing large arrays within loops. Tiling, strip-mining, loop interchanging, and loop skewing are widely used to transform a loop for better temporal and spatial locality for a given cache size. However, such analysis primarily targets capacity misses that occur when the working set of the loop exceeds the cache size. The loops can also suffer heavily due to conflict misses [Hennessy and Patterson 1996; Lam et al. 1991; McKinley and Temam 1996; Temam et al. 1994], thereby precluding effective cache utilization. Conflict misses can be particularly significant in caches with low associativity. In such situations programmers often rely on time-consuming cache profiling and performance tuning [Lebeck and Wood 1994; Martonosi et al. 1992]. There has also been compiler work in tailoring code to reduce conflict misses [Bacon et al. 1994; Coleman and McKinley 1995; Lam et al. 1991]. Unfortunately, conflict misses are highly sensitive to slight variations in problem size and base addresses [Bacon et al. 1994; Lam et al. 1991] and hence we need more precise characterization to understand the underlying cause behind such conflict misses.

Most previous compiler techniques to optimize loop nests either use simple cost models to guide loop transformations [McKinley et al. 1996; Wolf and Lam 1991] or are targeted towards some specific optimization [Bacon et al. 1994; Lam et al. 1991; Rivera and Tseng 1998]. There has also been some initial work on estimating the number of cache misses in numerical code [Ferrante et al. 1991; Temam et al. 1994]. Though the strategies given in previous papers help in reducing cache misses, they give little insight about the causes of such misses. Their limited focus or approximate modeling restricts their applicability. This paper attempts to fill this gap by finding precise relationships among the loop indices, array sizes and base addresses, and the cache parameters for the cache misses in a loop nest. Those relationships are used to generate a set of equations—called the *Cache Miss Equations* (or *CM equations* or *CMEs*)—representing *all the cache misses in a loop nest.* This simple, precise characterization allows one to better understand the cause behind such misses, and helps reduce cache misses in a methodical way.

CMEs are unique in unifying both the loop structure and the data layout within a simple, equations-based analytical model representing the cache misses. CMEs are a system of linear Diophantine equations. While solving these equations is difficult in general, we show that is also unnecessary as mathematical techniques for manipulating Diophantine equations allow us to relatively easily compute and/or reduce the number of possible solutions, where each solution corresponds to a potential cache miss. CMEs are generated statically at compile time. Any variable whose value is dependent on runtime information or whose optimizing value needs to be determined for some cache optimization is kept as a parameter in the CMEs. Compiler optimizations can then use these CMEs generated based on available information.

The CMEs provide a general framework that can be used to: (i) help a compiler in performing code transformations to improve cache usage, (ii) improve the simulation speeds of tools that simulate caches, and (iii) tighten bounds on program performance estimates. This paper focuses on the first application; we discuss how our equations can guide memory optimizations without going through time-consuming cache simulation. Our ultimate goal is to automate the analysis of the equations to build an efficient code optimizer.

We have implemented our algorithm to automatically generate the CMEs within the SUIF compiler system [Wilson et al. 1994]. We have tested our system by automatically generating the equations for many numerical loop nests including matrix-multiply, Gaussian elimination, successive over-relaxation (SOR), and loops from the SPECfp benchmarks. We have, in this paper, provided the accuracy of finding cache misses by our method. We have also shown, with the help of examples, how the precision and generality of the CME framework help in better cache optimizations when compared to earlier work.

The rest of this paper is organized as follows. Section 2 provides the underlying models and background information along with an overview of CMEs. Section 3 describes the algorithm to generate the CMEs. Since solutions to each CME represent *potential* cache misses, Section 4 describes how we can compose the effects of multiple CMEs to find the loop's actual cache misses. In addition, Section 4 gives experimental results on the accuracy of this method. Section 5 shows how these equations can be used to choose data padding/offset amounts or to choose a blocking factor in tiled code. Section 6 provides a discussion on the computational requirements of generating and using CMEs. Section 7 describes the future extensions to this work. Finally, Section 8 discusses related work and Section 9 contains the concluding remarks.

2. BACKGROUND AND OVERVIEW

A system of CMEs couches a loop's reference stream and cache conflict patterns in a mathematical framework that can be analytically manipulated. This section describes the abstractions and terminology we use to facilitate this.

2.1 Program Model

Our model applies to references in which the array subscript expressions and the bounds of the loop index are affine combinations of the enclosing loop indices, a common model for research in compiler memory analysis. All loops are assumed to be normalized such that the step value is 1 [Allen and Kennedy 1987]. We consider only perfectly nested loops and some imperfectly nested loops if they have only a single basic block in between the loops of a nest. We also assume that loops contain no conditional expressions. We plan to extend this work to handle conditionals and to perform inter-nest analysis. For the sake of uniformity, all arrays discussed here are assumed to be arranged in column-major order as in Fortran, but the techniques do not depend on a specific layout order. We also assume that all the load/store references inside a nest correspond to only the array references. Scalars can be considered as a special case of 1-D arrays.

In practice, however, the constraints in our program model are not too restrictive as shown by the empirical study presented in Table 2.1. It summarizes the number of loops which can be analyzed in a collection of programs taken from the SPECfp benchmarks based on our assumptions given above. For each program, Table 2.1 first gives the total number of *for* or *DO* loops found. It also lists the number of loops that are declared non-analyzable due to (i) function call (denoted by "Fcn call" in the table) or (ii) return instruction ("Ret") inside the loop body, (iii) non-affine loop bounds, (iv) non-constant step value, or (v) non-perfect loops. The "non-perfect loops" entry counts all the non-perfectly nested loops including those with conditional statements inside them. A single loop can be counted under more than one of the above categories. Non-affine array accesses are not listed here as we have not found a single case falling in that category.

Table 2.1 shows that loops with non-affine bounds and non-constant step are negligibly small. Non-perfectly nested loops and loops with function calls each constitute a small fraction of the total number of loops. Loops with function calls could sometimes be made analyzable if interprocedural analysis were used. The "variable bound" entry shows the number of loops which have variables in their loop bounds that cannot be determined at compile time. It shows that many of the loops have variable bounds. The "analyzable" column lists the number of loops and the associated loop nests that are analyzable with all loop bounds known at compile time. Loops that fall exclusively under the variable bounds classification are declared as *parametrically analyzable*. (We do not consider analyzable loops also as parametrically analyzable loops.) We can form our equations for such loops with the variables in the bounds represented by separate parameters. By treating these parameters as another equation variable, our analysis can make headway even though the loop bound may not be known until runtime.

Overall, the loop statistics show that scientific loop nests are mostly simple and regular, and we can analyze, absolutely or parametrically, a significant number of loops, approximately 70% of the total number of loops found in the SPECfp benchmarks.

Table I. Statistics on the number of SPECfp loops amenable to our analysis. (#L refers to the number of loops. #N refers to the number of loop nests.)

Program	Total #L	Fcn call	Ret	Non affine bound	Non constant step	Non perfect loops	Variable bound
TOMCATV	18	0	0	0	0	2	13
DNASA7	140	12	0	5	4	27	63
MDLJDP2	31	4	1	2	0	8	13
SWIM	24	0	0	0	0	0	24
HYDRO2D	159	3	0	3	2	33	131
FPPP	33	10	0	0	0	7	25
ALVINN	23	10	0	0	0	1	0
SU2COR	127	40	0	0	0	24	104
MGRID	53	11	1	0	0	9	48
DODUC	275	33	0	4	0	27	108
TURB3D	68	8	0	0	0	18	62
Total	951	131	2	14	6	156	591

	Analyzable		Parametrically Analyzable		Non Analyzable	
Program	#L	#N	#L	#N	#L	#N
TOMCATV	5	4	11	8	2	2
DNASA7	44	26	48	37	48	31
MDLJDP2	12	12	5	5	14	11
SWIM	0	0	24	16	0	0
HYDRO2D	20	18	102	63	37	23
FPPP	3	3	14	12	16	4
ALVINN	12	8	0	0	11	6
SU2COR	10	9	53	49	64	32
MGRID	1	1	31	18	21	12
DODUC	161	161	51	51	63	48
TURB3D	2	2	40	30	26	20
Total	270	244	379	289	302	189

2.2 Compilation Model

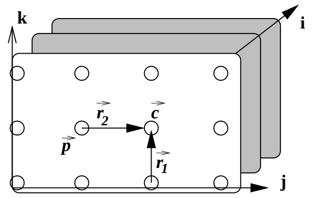
CMEs are linear Diophantine equations in constrained solution spaces. While *solving* these is difficult, we note that it is unnecessary for our approach. Mathematical techniques for manipulating Diophantine equations allow us to relatively easily compute and/or reduce the number of possible solutions without solving them.

In addition to program restrictions, it is important to clarify when CMEs are generated and used. CMEs are generated statically at compile-time, but may give data positioning hints to the linker. Since CMEs are analyzing possible cache conflicts, they need some information about the *relative* positioning of different data structures, but they do not need the *absolute* base address of any variable. Some of the optimizations we describe could be implemented by analyzing CMEs where relative variable spacings are a parameter, and then passing the linker information concerning what numeric constraints on their spacing will lead to the best performance.

In general, any variable whose value is dependent on runtime information (e.g. loop bounds) or whose optimized value is chosen by the compiler cache optimization

```
DO i = 1, N
   DO k = 1, N
      DO j = 1, N
         Z(j, i) += X(k, i) * Y(j, k)
```

(a) Matrix multiply loop nest



(b) Iteration space

Fig. 1. Matrix multiply loop nest and its iteration space. The iteration points are shown by the hollow dots in (b). $\vec{r_1} = (0,1,0)$ and $\vec{r_2} = (0,0,1)$ are two reuse vectors of $Z(j,i)$ shown at the iteration point $\vec{c} = (1,2,3)$. $\vec{r_1}$ is a self-temporal reuse vector and $\vec{r_2}$ is a self-spatial reuse vector.

is kept as a parameter in the CMEs. Compiler optimizations can then use these CMEs based on available information. However, in order to use CMEs to exactly count the number of cache misses, we need to know the values of all such parameters.

2.3 Architecture Model

The basic architecture we consider here is a uniprocessor model with a memory hierarchy. We focus on analyzing a single level of the data cache hierarchy. (Analyzing multiple levels simultaneously is not precluded, but it would complicate the equations.) The associativity of the cache is a parameter in our models, and CME methods apply to caches of any associativity from direct-mapped to fully-associative. We assume a least-recently-used (LRU) replacement policy. Writes and reads are modelled identically, so the model is of a write-allocate cache with fetch-on-write.

2.4 Terminology

Our work with CMEs draws on the substantial body of research in which iteration spaces and reuse vectors are used to analyze memory reference behavior for dependence analysis [Pugh 1992], locality optimizations [Wolf and Lam 1991], or prefetching algorithms [Mowry et al. 1992]. We build on these approaches and develop more precise mechanisms based on them.

2.4.1 **Iteration Space**. Every iteration of a loop nest is viewed as a single entity termed an *iteration point* in the set of all iteration points known as the *iteration space*. Formally, we represent a loop nest of depth n as a finite convex polyhedron of the n-dimensional iteration space $\mathcal{Z}^n$, bounded by the loop bounds [Irigoin and Triolet 1988]. Each iteration in the loop corresponds to a node in the polyhedron and is called an *iteration point*. Every iteration point is identified by its index vector $\vec{i} = (i_1, i_2, \cdots, i_n)$, where i_l is the loop index of the l^{th} loop in the nest with the outermost loop corresponding to the leftmost index. We use the matrix multiplication example given in Figure 1(a) to illustrate the concepts presented in this section. Figure 1(b) shows the iteration space of the matrix-multiply loop nest. $\vec{c}$ is an iteration point and corresponds to the iteration $i = 1, k = 2$, and $j = 3$. In this representation, if iteration $\vec{p}_2$ executes after iteration $\vec{p}_1$ we write $\vec{p}_2 \succ \vec{p}_1$ and say that $\vec{p}_2$ is lexicographically greater than $\vec{p}_1$. For example in Figure 1(b), $\vec{c} \succ \vec{p}$.

2.4.2 **Memory terminology**. We present the following two definitions to avoid any confusion regarding some commonly used terms.

Definition 1. A **reference** is a static read or write in the program, while a particular execution of that read or write at runtime is a **memory access**.

Definition 2. A **memory line** refers to a cache-line-sized block in the memory, while a **cache line** refers to the actual cache block to which a memory line is mapped to.

Throughout this paper, we denote the cache size as C_s, associativity of the cache as k, line size as L_s, and the number of cache sets as N_s. As each cache set consists of k cache lines, we can write $C_s = N_s \times k \times L_s$. The cache set, to which the memory address accessed by a reference R_A at the iteration point $\vec{i}$ is mapped to, is given by the following expression (with all elements in units of data element size):

$$\begin{aligned} &Mem_{R_A}(\vec{i}) = Memory\_Address\_of\_R_A(\vec{i}) \\ &Memory\_Line_{R_A}(\vec{i}) = \lfloor Mem_{R_A}(\vec{i})/L_s \rfloor \\ &Cache\_Set_{R_A}(\vec{i}) = \lfloor Mem_{R_A}(\vec{i})/L_s \rfloor \mod N_s \end{aligned} \tag{1}$$

where $Mem_{R_A}(\vec{i})$, the memory address accessed by R_A at $\vec{i}$, is an affine function of the loop indices and can be easily computed from the subscript expressions of R_A.

For example, the cache set of the reference $Z(j, i)$ in the matrix multiply loop nest of Figure 1(a) is given by (with all numbers in units of data element size):

$$\lfloor (4192 + 32i + j - 1)/4 \rfloor \mod 128$$

where the base address of the array Z is 4192 and the number of elements per column of Z is 32. The cache considered is an 8KB 2-way set-associative cache with 128 cache sets and 4 data elements per cache line.

2.4.3 **Reuse Vector**. Reuse vectors provide a mechanism for summarizing repeated memory access patterns in loop-oriented code [Wolf and Lam 1991]. If a reference accesses the same memory line in iterations $\vec{i}_1$ and $\vec{i}_2$, where $\vec{i}_2 \succ \vec{i}_1$, we say that there is reuse in direction $\vec{r} = \vec{i}_2 - \vec{i}_1$ and $\vec{r}$ is called a *reuse vector*. For example, the reference $Z(j, i)$ in Figure 1(a) can access the same memory line at the iteration points (i, k, j) and $(i, k, j + 1)$ and hence one of its reuse vectors is $(0, 0, 1)$ as shown in Figure 1(b). A reuse vector is repeated across the iteration space.

In order to find out whether a reference misses in the cache in a particular loop iteration, we need to know whether the memory line is being accessed for the first time or whether it is reusing a previously-accessed memory line. If it is reusing a previously-accessed memory line, we need to know when it was last accessed and the reference that accessed it. Once we have the information about the reuse, we can check if any intervening memory access evicts the memory line from the cache before it can be reused; this would result in a cache miss.

Definition 3. When a reference within a loop nest accesses a memory line that was already accessed before, it is called a **reuse** of the memory line by the reference [Wolf and Lam 1991].

Reuse vectors provide a concise mathematical representation of the reuse information of a loop nest. Reuse can be classified into four different types [Wolf and Lam 1991]. It is called *self-temporal reuse* if a reference reuses a memory line by accessing the same data. It is called *self-spatial reuse* if a reference reuses a memory line by accessing a data different from the one accessed before in the same memory line. Furthermore, the same memory line can be accessed by different references resulting in a *group reuse*. For example, the references $A(j,i)$ and $A(j+1,i)$ can access the same memory line in the same iteration or in different iterations. Again, it is called *group-temporal* reuse if a reference reuses a memory line by accessing the same data that was accessed before by another reference. It is called *group-spatial* reuse if a reference reuses a memory line by accessing a data different from the one accessed before by another reference in the same memory line.

Definition 4. If a reuse results in a cache hit we say that the **reuse is realized**.

Hence if we had an infinitely large cache, every reuse would result in a cache hit. In practice, however, reuse does not necessarily result in cache hit. The central idea behind the CMEs is to find the loop instances at which reuse does not result in cache hits.

We have extended reuse analysis as presented by Wolf and Lam [1991] and modified SUIF to generate, when needed, additional reuse vectors for more accurate analysis. These additional reuse vectors represent reuse directions that are not provided by the basic reuse vectors generated by SUIF. For example, in Figure 1, considering a cache line size of 2 data elements, there is a reuse of $Z(j,i)$ in the direction $(0,1,-1)$ which is not generated by SUIF. The approximate model used by SUIF to quantify reuse needs only the basis reuse vectors, while our precise analysis needs to know every reuse direction. As we show in Section 4, however, the basic SUIF reuse vectors are almost always sufficient for counting cache miss points with no error.

2.4.4 **Miss Along a Reuse Vector**. All the algorithms described in this section assume that only one reuse vector of a reference is present at a time. For the sake of brevity, hereafter, we use the phrase *along a reuse vector* to mean that for a reference *only that reuse vector is assumed to be present, ignoring the presence of any other reuse vectors, if any.* For example a *miss along a reuse vector* is defined as follows:

Definition 5. Consider a miss of a reference R at an iteration point $\vec{i}$. We define the **miss** to be **along a reuse vector** $\vec{r}$ of R, if that miss would occur if $\vec{r}$ were the only reuse vector present for R.

Each CME generated in our algorithm is for a particular reuse vector $\vec{r}$ of a reference R. In other words, all the *misses* of R represented by that CME are *along the reuse vector* $\vec{r}$. In Section 4, we show how all the reuse vectors interact to decide the cache misses for a reference.

3. GENERATING THE CACHE MISS EQUATIONS

In this paper, the term *equation* has been used loosely to represent a set of simultaneous equalities or inequalities. Our approach generates two types of CMEs: *cold*

Algorithm GENERATE:
Generate the CMEs for a loop nest

Input:
(1) Loop nest information: Bounds and step of each loop index;
(2) Array reference information: For each array reference the base address, size and indexing expression of each dimension;
(3) Reuse information: All the reuse vectors of each reference in the loop nest.
(4) Reference sequence: sequence in which the references appear in the generated code of the loop nest.

Output: A set of CMEs for each reference.

Algorithm:

```
For each reference in the loop nest
  For each reuse vector of this reference
    [1] Generate a cold miss equation
        (described in Section 3.1).
    [2] For each reference in the loop nest
          Generate a replacement miss equation
          (described in Section 3.2).
```

Fig. 2. Algorithm to generate all the CMEs.

miss equations (or *cold CMEs*) and *replacement miss equations* (or the *replacement CMEs*). Solutions to the cold miss equations represent potential *cold* or *compulsory misses*—misses that occur on the first access to a memory line. Solutions to the replacement miss equations represent all other misses including both *capacity* and *conflict misses* [Hill 1987; Hill and Smith 1989].

Figure 2 summarizes the algorithm to generate all CMEs of a loop nest. The two major steps: generating a cold miss equation and generating a replacement miss equation, are described in the next two subsections. In the methods described in the next two subsections, $\vec{i} = (i_1, i_2, \cdots, i_n)$ is considered the current iteration point. The constraints bounding $\vec{i}$ within the iteration space of the loop nest are given by the following inequalities:

$$l_p < i_p < u_p, \ \forall\ p:\ 1 \leq p \leq n \tag{2}$$

where l_p and u_p are the lower and upper bounds of the iteration range of the loop index variable i_p. Every CME generated investigates the conditions which could lead to a miss at $\vec{i}$ along a particular reuse vector. The constraints bounding $\vec{i}$, as given in Equation 2, are included in every CME generated. To avoid restating the same constraints, we assume the presence of these inequalities in every CME given in the rest of this section.

3.1 Forming Cold Miss Equations

Cold miss equations or cold CMEs are formed by investigating the situations when a memory line is brought into the cache for the first time. As each loop nest is treated in isolation, we assume none of the data accessed in a loop nest is already present in the cache before it starts execution. This can result in a pessimistic estimate of a loop's cache miss behavior. The simplest type of cold CME is already fairly familiar. Namely, a relationship like $(j \bmod 4) = 0$ summarizes that, in a sequence of unit-stride accesses in which four data elements fit on a cache line, every fourth access will result in a cold miss. Cold CMEs become more complicated depending on loop nesting depths, strides, access patterns or array alignments, but the basic goal remains the same.

For each reference, we form a cold CME which captures all the cold misses along each reuse vector. In this section, we show how to generate the cold CMEs for some reuse vector $\vec{r} = (r_1, r_2, \cdots, r_n)$ of a particular reference, say R_A. Depending on the type of reuse of $\vec{r}$, the methods for generating these equations are given below.

3.1.1 **Spatial reuse (self or group)**. There can be a cold miss along a spatial reuse vector for either of two reasons given below:

(1) There is a cold miss when the present access is the first access along this vector. That means the previous iteration point along this vector lies outside the iteration space. For example, in Figure 1(b), $Z(j,i)$ has spatial reuse along the vector $(0,0,1)$. So the access of $Z(j,i)$ in the iteration $(1,2,1)$ is a cold miss along that reuse vector, because the previous iteration point along this vector $(1,2,0)$ is outside the iteration space. As we are considering the reuse vector $\vec{r} = (r_1, r_2, \cdots, r_n)$, an access is a cold miss along $\vec{r}$ if the corresponding iteration point satisfies any of the following inequalities:

$$\begin{array}{c} i_1 - r_1 < l_1, \quad i_1 - r_1 > u_1, \\ \vdots \\ i_n - r_n < l_n, \quad i_n - r_n > u_n \end{array} \tag{3}$$

where $l_1, l_2, \cdots, l_n$ are the lower bounds and $u_1, u_2, \cdots, u_n$ are the upper bounds of the previous iteration point along $\vec{r}$ which is $\vec{p} = (i_1 - r_1, i_2 - r_2, \cdots, i_n - r_n)$

(2) There can be a cold miss along a spatial reuse vector also when a memory line boundary is crossed along that vector. This means that the memory line accessed in the present iteration point $\vec{i}$ by the reference R_A is different from the memory line accessed in the previous iteration point $\vec{p} = \vec{i} - \vec{r}$ along $\vec{r}$ by the same reference (if self reuse) or by a different reference, say $R_A^{'}$ (if group reuse). (Group reuse between the references R_A and $R_A^{'}$ implies that they access the same array.) Hence, there are cold misses along $\vec{r}$ in the iteration points $\vec{i}$ which satisfy the following relation:

$$\mathit{Memory\_Line}_{R_A}(\vec{i}) \neq \mathit{Memory\_Line}_{R_A^{'}}(\vec{p}) \tag{4}$$

Let us denote $m_p = \mathit{Mem}_{R_A^{'}}(\vec{p})$, memory address accessed by $R_A^{'}$ at $\vec{p}$, and $m_i = \mathit{Mem}_{R_A}(\vec{i})$, memory address accessed by R_A at $\vec{i}$. We can express Equation 4 as the two inequalities in Equation 5. This is illustrated in Figure 3.

$$\begin{array}{lll} m_i < m_p - L_{\mathit{off}} & \text{if } m_i < m_p & \text{(negative stride)} \\ m_i > m_p + (L_s - 1 - L_{\mathit{off}}) & \text{if } m_i > m_p & \text{(positive stride)} \\ \quad \text{where } L_{\mathit{off}} = m_p \ \mathit{mod}\ L_s & & \end{array} \tag{5}$$

When the array is accessed with unit stride we can further simplify Equation 5

(a) Negative Stride (b) Positive Stride

Fig. 3. Cold miss examples along a spatial reuse vector when a memory line boundary is crossed. If m_p is the array element accessed in the previous iteration, the access of array element m_i in the current iteration will result in a cold miss.

to the more familiar form: $m_i \ mod \ L_s = 0$ since $m_i - m_p = 1$ for unit stride. For example, in Figure 1(b), the cold misses of $Z(j,i)$ along the spatial reuse vector $(0,0,1)$ can be simplified to $(j \ mod \ 4) = 0$, assuming the column size of Z to be of the form $4n$, where n is any integer, and the cache line can hold four array elements.

3.1.2 **Temporal reuse (self or group)**. Cold misses along a temporal reuse vector occur only for the iteration points which lie first along the temporal reuse vector. This is similar to the first case given for spatial reuse vectors. All the cold misses along this vector are given by the same equations as in Equation 3.

The methods described here generate cold CMEs along a single reuse vector. Eventually, the CMEs of all the reuse vectors will be combined to find the actual cache misses as shown in Section 4.

3.2 Forming Replacement Miss Equations

3.2.1 **Intuition and Overview**. Replacement CMEs summarize conflict and capacity misses in which the currently accessed memory line was previously resident but has been evicted from the cache. The intuition behind these equations is fairly straightforward, if first considered in a direct-mapped cache. In a direct-mapped cache, a miss occurs if, between consecutive accesses to a particular memory line, another access occurs to a distinct memory line that maps to the same cache line.

For example, consider the tiny reference stream: R_A–R_B–R_A. A conflict clearly occurs in a direct-mapped cache if $Cache\_Line\_of\_R_A = Cache\_Line\_of\_R_B$. This happens if:

$$Memory\_Address\_of\_R_A = Memory\_Address\_of_{R_B} + n \times Cache\_Size + Line\_Size\_Range \quad (6)$$

That is, a conflict occurs, roughly speaking, between R_A and R_B whenever the memory addresses accessed by them differ by multiples of the cache size. In order to be precise, we need two further details. First, n cannot be zero, because in that case the memory addresses reside on the same memory line. Second, the *Line_Size_Range* is included in the equation to capture the situations when the memory addresses do not differ by exactly a multiple of the cache size, but they map to the same cache line. (See Figure 4.) *Line_Size_Range* is a range whose size is set to capture the offset effects based on where the memory addresses sit in their respective memory lines. Since memory addressing is an affine function, Equation 6 is a linear Diophantine equation.

For a k-way set-associative cache, a miss occurs if between consecutive accesses to a particular memory line, at least k other accesses occur to distinct memory lines

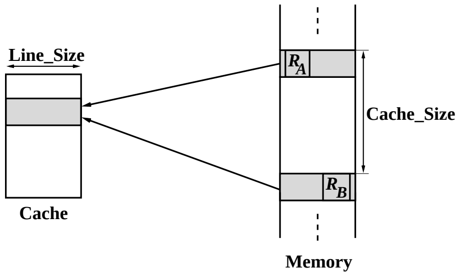


Fig. 4. An example of two memory addresses that map to the same cache line. The addresses are marked by the references that access them, namely R_A and R_B.

that map to the same cache set. Each of these conflicts satisfy an equation similar to Equation 6. Intuitively, a reuse vector provides a closed form representation of a particular reference stream over the entire iteration space for a particular reference. We utilize this representation to form equations characterizing all the conflicts along that reference stream. The formal method to form these equations is described below.

3.2.2 **Formal method**. Every replacement CME represents a contention between two references for the k cache lines in a cache set. Also, like cold CMEs, each replacement CME is formed by considering a single reuse vector at a time.

Say we want to find the replacement CMEs for the reference R_A along the reuse vector $\vec{r} = (r_1, r_2, \cdots, r_n)$. These will fall into two categories. Self-interference equations summarize inter-iteration interactions of the same reference.[1] Cross-interference equations summarize the interactions with *other* references in the loop. Here we show how to form the replacement CME representing the interferences with the reference R_B. For self-interference equation, references R_A and R_B are identical. If the current iteration point is $\vec{i} = (i_1, i_2, \cdots, i_n)$ and the reuse vector is $\vec{r}$ then the iteration point where the memory line was last accessed by R_A along $\vec{r}$ is $\vec{p} = \vec{i} - \vec{r} = (i_1 - r_1, i_2 - r_2, \cdots, i_n - r_n)$. The iteration points at which we can have an interference with R_B are all the points lying between $\vec{p}$ and $\vec{i}$ which are considered as the set of potentially-interfering points. Whether we include the points $\vec{p}$ and $\vec{i}$ also in that set depends on the relative access order of R_A and R_B in a loop nest iteration. If R_A occurs before R_B in the nest, only $\vec{p}$ has to be considered, otherwise, only $\vec{i}$ needs to be considered. In our implementation, we extract access order information from the code generation phase automatically. We represent all points in the set of potentially-interfering points as $\vec{j} = (j_1, j_2, \cdots, j_n)$, where

$$
\begin{aligned}
\vec{j} &\in [\vec{p}, \vec{i}) \text{ if access of } R_A \text{ is before } R_B, \\
&\in (\vec{p}, \vec{i}] \text{ if access of } R_A \text{ is after } R_B, \\
&\in (\vec{p}, \vec{i}) \text{ if both } R_A \text{ and } R_B \text{ represent the same reference} \\
&\qquad \text{(for the self-interference equation)} \qquad (7)
\end{aligned}
$$

[1]Other research sometimes uses self-interference more broadly to refer to any cache interferences of a data structure with itself.

The range of $\vec{j}$ comprising of the set of points given in Equation 7 are represented by sets of equalities and/or inequalities. A general method for finding such a representation for the range $[\vec{p}, \vec{i})$ is provided in Appendix A; the other ranges are represented similarly.

Cache interference occurs when the cache set accessed by R_A in iteration $\vec{i}$ is the same as any of the cache sets accessed by R_B at every potentially-interfering iteration $\vec{j}$ between iteration $\vec{p}$ and iteration $\vec{i}$. Equating the appropriate cache sets accessed gives the condition for a cache set contention along reuse vector $\vec{r}$:

$$Cache\_Set_{R_A}(\vec{i}) = Cache\_Set_{R_B}(\vec{j}) \tag{8}$$

Substituting in the expressions from Equation 1, we can simplify the resulting equation to the linear Diophantine equation shown in Equation 9. This simplification is described in detail in Section 3.3.

$$Mem_{R_A}(\vec{i}) = Mem_{R_B}(\vec{j}) + nC_s/k + b \tag{9}$$

where C_s is the cache size, k is the associativity of the cache and n is any non-zero integer. The variable b can take on values in the range $-L_{\mathit{off}} \leq b \leq L_s - 1 - L_{\mathit{off}}$ where $L_{\mathit{off}} = Mem_{R_B}(\vec{j})\ mod\ L_s$. Thus, L_{off} shows the offset of the reference R_B in its cache line, and b bounds the search for an interference within that cache line. With a slightly conservative estimate of the cache interferences, the range of the variable b can be simply written as $-(L_s - 1) \leq b \leq (L_s - 1)$. Since the loop indices are bounded, the equality holds for a bounded region.

Every solution of Equation 9 is a vector of the form $(\vec{i}, \vec{j}, n)$ where $\vec{i}$ is the iteration where R_A might suffer a miss. $\vec{j}$ is the iteration, before R_A's access in iteration $\vec{i}$, where R_B accesses the same cache set. The memory lines of R_A and R_B involved in this cache set contention are separated by (n/k) cache sizes. So, if n is 0 the memory lines are identical and there is no conflict at all. For this reason we disallow all solutions with $n = 0$. In a k-way set associative cache k distinct contentions are needed before a replacement miss will occur at $\vec{i}$ along the reuse vector $\vec{r}$. The algorithm in Section 4 shows how to combine the solutions of all the replacement CMEs of a reuse vector to provide all the replacement misses along that reuse vector.

3.2.3 **Example**. For the matrix multiply example shown in Figure 1 with $N = 32$, consider generating replacement CMEs for $Z(j, i)$ along the spatial reuse vector $\vec{r} = (0, 0, 1)$. If $\vec{i} = (i, k, j)$ then $\vec{p} = \vec{i} - \vec{r} = (i, k, j - 1)$. For an 8KB 2-way set-associative cache with 128 cache sets and 4 array elements per cache line, Equation 10 shows the replacement miss equation for the interferences with $X(k, i)$ along $\vec{r}$. (Here the access of $Z(j, i)$ is after $X(k, i)$ in each loop nest iteration.)

$$\begin{aligned}
&Cache_Set_Z(j, i) = Cache_Set_X(k', i') \\
&\qquad \text{where } (i', k', j') \in ((i, k, j - 1), (i, k, j)] \\
&\Rightarrow \lfloor (4192 + 32i + j - 1)/4 \rfloor \text{ mod } 128 \\
&\qquad = \lfloor (2136 + 32i + k - 1)/4 \rfloor \text{ mod } 128 \\
&\Rightarrow 4192 + 32i + j = 2136 + 32i + k + 512n + b
\end{aligned} \tag{10}$$

where $n > 0$, $(i, k, j) \in [(0, 0, 0), (31, 31, 31)]$, $-L_{\mathit{off}} \leq b \leq 3 - L_{\mathit{off}}$, $L_{\mathit{off}} = (2136 + 32i + k - 1)\ mod\ 4$. The range of b can be written more simply as $-3 \leq b \leq 3$ if a slightly pessimistic estimate of the interferences is acceptable. 4192 and 2136 are the base addresses (in array elements) of the arrays Z and X respectively, and 32 is the number of elements per column in the arrays.

3.3 Simplifying the Cache Miss Equations

All the constraints governing the CMEs generated in the last two subsections are in the form of linear equalities or inequalities except the one given in Equation 8. Figure 5 illustrates the simplification of Equation 8, which is simply $Cache\_Set_{R_A}(\vec{i}) = Cache\_Set_{R_B}(\vec{j})$. The memory is viewed as a contiguous set of cache-line sized blocks or memory lines as shown in Figure 5. Assume that the cache size is C_s and it consists of N_s cache sets (that is, $C_s = N_s \times k \times L_s$). The condition of Equation 8 states that the cache set accessed by R_A at $\vec{i}$ must be the same as that accessed by R_B at $\vec{j}$. In other words, both the memory lines accessed by R_A at $\vec{i}$ and by R_B at $\vec{j}$ are mapped to the same cache set. Memory lines are mapped to cache sets in a circular or modulo fashion. Hence, two memory lines are mapped to the same cache set if they are separated by a multiple of the total number of cache sets, since $(m + n \times N_s)\ mod\ N_s = m\ mod\ N_s$ where m, n are any integers. Equation 8 holds if and only if the memory line accessed by R_A at $\vec{i}$ is $n \times N_s$ cache sets away from that accessed by R_B at $\vec{j}$, where n is any integer except zero. Only non-zero values of n are considered because when n is zero the memory lines accessed are identical and so do not correspond to conflicting accesses. Let us assume that, at $\vec{j}$, R_B accesses the memory address $M_B = Mem_{R_B}(\vec{j})$ which lies in the memory line l_B as shown in Figure 5. If the memory line accessed by R_A at $\vec{i}$ is l_A, Equation 8 can be rewritten in terms of memory lines as follows:

$$l_A \;=\; l_B + n \times N_s, \quad \text{where } n \text{ is any non-zero integer} \tag{11}$$

Now, let e be the memory address located at the same offset x from the base of the memory line l_A as the address M_B from the base of the corresponding memory line l_B. If $Mem_{R_A}(\vec{i})$, the memory address accessed by R_A at $\vec{i}$, is equal to e, Equation 11 can be represented in terms of the memory addresses as follows:

$$Mem_{R_A}(\vec{i}) = Mem_{R_B}(\vec{j}) + nC_s/k, \quad \text{where } n \text{ is any non-zero integer} \tag{12}$$

$Mem_{R_A}(\vec{i})$, however, can lie anywhere within the memory line l_A to satisfy Equation 8 or Equation 11. Equivalently, it can lie anywhere within the range of $-x$ to $(y-1)$ with respect to the address e, where, $x = L_{off} = Mem_{R_B}(\vec{j})\ mod\ L_s$ and $y = L_s - x$. Hence, Equation 12 can be modified to Equation 13 which provides the generalized and simplified form of Equation 8 in terms of the memory addresses of R_A and R_B, namely Mem_{R_A} and Mem_{R_B} respectively.

$$Mem_{R_A}(\vec{i}) = Mem_{R_B}(\vec{j}) + nC_s/k + b \tag{13}$$

where n is any non-zero integer. The variable b can take on values in the range $-L_{off} \le b \le L_s - 1 - L_{off}$ where $L_{off} = Mem_{R_B}(\vec{j})\ mod\ L_s$. We can have simpler bounds for b if we ignore the relative position of $Mem_{R_B}(\vec{j})$ in its cache line. Now, in order to include all the solutions of Equation 13, in spite of ignoring the offset of $Mem_{R_B}(\vec{j})$ in its cache line, the bounds of b can be written as $-(L_s - 1) \le b \le (L_s - 1)$. This would result in a slightly more pessimistic estimate of the cache misses. As $Mem_{R_A}(\vec{i})$ and $Mem_{R_B}(\vec{j})$ are affine functions of the loop indices,

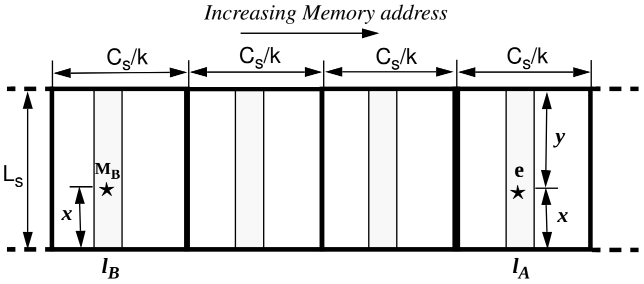


Fig. 5. Memory viewed as contiguous set of memory lines for simplifying the CMEs. A shaded area stands for a single memory line. $M_B = Mem_{R_B}(\vec{j})$, the memory address accessed by R_B at $\vec{j}$, lies in the memory line l_B. l_A, the memory line accessed by R_A at $\vec{i}$, is at $n \times C_s/k$ elements away, where n is any non-zero integer ($n = 3$ in this figure). $Mem_{R_A}(\vec{i})$, the memory address accessed by R_A at $\vec{i}$, can lie anywhere in the memory line l_A. $x = L_{off} = Mem_{R_B}(\vec{j}) \text{ mod } L_s$ and $y = L_s - x$.

Equation 13 results in a linear equation which is allowed to have only integral solutions, that is, a linear Diophantine equation. Since the loop indices are bounded, the equation holds for a bounded region.

3.4 Putting it All Together

In the previous three subsections, we have described the methods to generate and simplify the CMEs. Figure 6 summarizes the CMEs generated and simplified by these methods for the reference $Z(j, i)$ along its spatial reuse vector $(0, 0, 1)$ in the matrix multiply loop nest of Figure 1. The equations listed in Figure 6 are presented in terms of the cache and data layout parameters. Such an architectural view of the CMEs should provide useful insights into different sources of cache optimizations. The base addresses of the arrays X, Y, and Z are denoted as Offset_X, Offset_Y, and Offset_Z respectively. Col_size_X, Col_size_Y, and Col_size_Z represent the column sizes of the arrays X, Y, and Z respectively. To further generalize the loop nest, we have represented the lower and upper bounds of the loop indices i, k, j as Lbound_i, Lbound_k, Lbound_j and Ubound_i, Ubound_k, Ubound_j respectively.

4. FINDING CACHE MISSES FROM THE CACHE MISS EQUATIONS

This section describes the algorithm for finding all cache misses in a loop nest by composing the effects of multiple CMEs. This discussion is useful for building intuition about how CME solutions relate to cache miss instances. It is important to note, however, that *most of the cache optimizations including the ones described in Section 5 would never be required to execute this algorithm on a per-loop basis.* Instead, as described in Section 5, we typically use mathematical shortcuts to derive cache optimization algorithms from the CMEs once they are generated.

4.1 Algorithm

As described in Section 3, for every reference we generate a set of equations for each of its reuse vectors. For each reuse vector there are at most two cold CMEs representing cold misses along that vector [Section 3.1], as well as replacement CMEs representing the self- and cross-interferences of this reference with itself and

Cold Miss Equations:

(1) $j = \text{Lbound}_j$ for $\text{Lbound}_j \leq j \leq \text{Ubound}_j$

(2) $j \bmod L_s = 0$ for $\text{Lbound}_j \leq j \leq \text{Ubound}_j$

Replacement Miss Equations: (assuming, access of $Z(j,i)$ is after that of both $X(k,i)$ and $Y(j,k)$ in each loop iteration)

Cross-interference equations:

(3) $\text{Offset}_Z + \text{Col\_size}_Z \times i + j = \text{Offset}_X + \text{Col\_size}_X \times i' + k' + C_s \times n + b$
for $i' = i; k' = k;$
$n \geq 0$ (assuming, $\text{Offset}_X < \text{Offset}_Z$);
$L_{\text{off}} = (\text{Offset}_X + \text{Col\_size}_X \times i' + k') \bmod L_s$; $-L_{\text{off}} \leq b \leq L_s - 1 - L_{\text{off}}$;
$\text{Lbound}_i \leq i \leq \text{Ubound}_i$; $\text{Lbound}_k \leq k \leq \text{Ubound}_k$; $\text{Lbound}_j \leq j \leq \text{Ubound}_j$;

(4) $\text{Offset}_Z + \text{Col\_size}_Z \times i + j = \text{Offset}_Y + \text{Col\_size}_Y \times k' + j' + C_s \times n + b$
for $k' = k; (j-1) < j' \leq j;$
$n \geq 0$ (assuming, $\text{Offset}_Y < \text{Offset}_Z$);
$L_{\text{off}} = (\text{Offset}_Y + \text{Col\_size}_Y \times k' + j') \bmod L_s$; $-L_{\text{off}} \leq b \leq L_s - 1 - L_{\text{off}}$;
$\text{Lbound}_i \leq i \leq \text{Ubound}_i$; $\text{Lbound}_k \leq k \leq \text{Ubound}_k$; $\text{Lbound}_j \leq j \leq \text{Ubound}_j$;

Self-interference equation:

(5) $\text{Col\_size}_Z \times i + j = \text{Col\_size}_Z \times i' + j' + C_s \times n + b$
for $i' = i; (j-1) < j' \leq j;$
$-\infty < n < \infty;$
$L_{\text{off}} = (\text{Offset}_Z + \text{Col\_size}_Z \times i' + j') \bmod L_s$; $-L_{\text{off}} \leq b \leq L_s - 1 - L_{\text{off}}$;
$\text{Lbound}_i \leq i \leq \text{Ubound}_i$; $\text{Lbound}_k \leq k \leq \text{Ubound}_k$; $\text{Lbound}_j \leq j \leq \text{Ubound}_j$;

Fig. 6. CMEs of $Z(j,i)$ along its spatial reuse vector $(0,0,1)$ in the matrix multiply loop nest of Figure 1. The two cold miss equations correspond to the two cases described in Section 3.1.1. For the second equation, we have assumed that every column of the array Z starts at a cache line boundary. The third and the fourth equation represents the cross-interferences of $Z(j,i)$ with $X(k,i)$ and $Y(j,k)$ respectively. Without loss of generality, we have assumed that (i) the access of $Z(j,i)$ is after that of both $X(k,i)$ and $Y(j,k)$ in each loop iteration, and (ii) the offsets or the base addresses of both the arrays X and Y are at a lower address than that of the array Z. The last equation stands for the self-interferences of $Z(j,i)$.

others. CME solution points represent *potential* cache misses; to find actual cache misses, however, one must consider the effects of multiple reuse vectors at once. Figure 7 provides the algorithm that combines the effects of multiple reuse vectors in order to determine the set of all cache miss instances of a loop nest from the solutions of all of its CMEs.

Here, we will provide an intuitive explanation of the algorithm shown in Figure 7 with the help of an illustrative example. This algorithm is based on two theorems presented and proved in Appendix B. We will consider the iteration space shown in Figure 8 as our example. The algorithm first sorts the reuse vectors of a reference in lexicographically-increasing order. In our example, assume that a reference X has three reuse vectors $\vec{r_1}$, $\vec{r_2}$, and $\vec{r_3}$. This means it accesses the same memory line at the iteration points $\vec{i_1}, \vec{i_2}, \vec{i_3}$, and $\vec{i_4}$. The lexicographic ordering of the reuse vectors is $[\vec{r_3}, \vec{r_1}, \vec{r_2}]$. For each reuse vector, a number of CMEs are generated,

Algorithm Find_Cache_Misses_of_a_Loop_Nest
Input: for each reference, solution sets of the CMEs for every reuse vector of the reference
Output: M_X for every reference X, where M_X = set of miss points of X
{
for each reference /* Say the reference is X */
  Sort the reuse vectors lexicographically from the shortest to the longest one;
  $M_X = \phi$ (null set);
  C = Set of all iteration points;
  /* M_X keeps track of the cache miss points found */
  /* C keeps track of the iteration points that need further investigation */
  for each reuse vector of the reference X /* Say the reuse vector is $\vec{r}$ */
    if ($|C| \le \varepsilon$) **break**;
    /* No further investigation for this reference if $|C|$, the #elements in C, is less than ε; ε is set to 0 for an exact output */
    C' = union of the solutions of cold CMEs of $\vec{r}$;
    R' = union of the solutions of replacement CMEs of $\vec{r}$;
    $R'' = \phi$;
    for each $(\vec{i}, \vec{j}, n) \in R'$
      $R'' = R'' \cup \{(\vec{i}, n)\}$; /* R'' stores the distinct $(\vec{i}, n)$ tuples */
    $I = \phi$;
    for each $(\vec{i}, n) \in R''$
      $I = I \cup \{i\}$; $|\vec{i}|$ += 1; /* $|\vec{i}|$ keeps track of #occurrences of $\vec{i}$ */
      /* So, $|\vec{i}|$ counts the distinct cache set contentions */
      /* All $|\vec{i}|$'s are initialized to 0 before this loop */
    $I = I \cap C$; $C = C' \cap C$; /* Search inside C only */
    $I = I - C$; /* eliminate cold CME solution points from I */
    for each $\vec{i} \in I$
      if $|\vec{i}| \ge k$ /* k = associativity of the cache */
        /* $\vec{i}$ is a replacement miss point along $\vec{r}$ */
        $M_X = M_X \cup \{\vec{i}\}$;
  /* At this point C = cold misses of the reference */
  /* and M_X = replacement misses of the reference */
$M_X = M_X \cup C$;
}

Fig. 7. Algorithm to find the cache miss points of a loop nest from its CME solutions.

each producing a collection of CME solution points. The algorithm investigates one reuse vector at a time, starting from the shortest one which is $\vec{r_3}$ here. After investigating a reuse vector, some of its CME solution points are declared definite miss points, while others are indeterminate. If any iteration point is a solution for a cold CME of $\vec{r_3}$, there is a cold miss along $\vec{r_3}$ at that point. Hence, there is no reuse along $\vec{r_3}$ at that point. As we cannot take any further decision about these iteration points without considering other reuse vectors, we declare them as 'indeterminate' for $\vec{r_3}$. These indeterminate points are passed on to the next reuse vector for further investigation. However, if an iteration point is not a solution for a cold CME but is a replacement miss along $\vec{r_3}$, it is declared a definite miss point for the reference. This is because if the memory line is replaced after its use

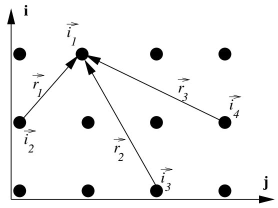


Fig. 8. Illustration of the algorithm to find cache misses for a 2D loop nest.

at $\vec{i_4}$, there is no further access (i.e., no other shorter reuse vector) to prevent the cache miss at $\vec{i_1}$. Finally, any iteration point that is neither a cold CME solution point nor a replacement miss along $\vec{r_3}$ is a guaranteed hit. That is, if the cache line is not replaced after its use at $\vec{i_4}$, X will enjoy a hit at $\vec{i_1}$ irrespective of what happens along other reuse vectors. For only the indeterminate points (i.e., cold CME solution points) of $\vec{r_3}$, we move on to consider $\vec{r_1}$. All the CME solutions of $\vec{r_1}$ are treated similar to those of $\vec{r_3}$ as we can consider $\vec{r_3}$ effectively absent for all its cold CME solution points. Finally, we consider $\vec{r_2}$ within the points that are declared indeterminate after investigating both $\vec{r_3}$ and $\vec{r_1}$.

So, in general the algorithm works as follows. We consider reuse vectors one at a time in lexicographically-increasing order. While considering each reuse vector, some of its CME solution points are declared definite misses, while others are indeterminate. Then, considering only the set of indeterminate solution points, we move on to consider the CMEs from the lexicographically-next reuse vector for this reference. Intuitively, the indeterminate points form a reduced iteration space that need further investigation. We continue investigating further reuse vectors until the number of indeterminate points is either zero, or is "sufficiently small" (as defined by a user threshold). At that point, we can stop the process, even if the reference has additional reuse vectors that we have not yet considered. Since a replacement miss point found along the current reuse vector in the algorithm is a guaranteed miss point, it is included in the global miss set M_X (Line 19 in Figure 7) immediately after it is found. In Figure 7, C maintains the set of indeterminate iteration points and ε is the tolerable error in miss count per reference. Section 4.3 will show that in practical loop nests, perfect accuracy can be obtained by considering a relatively small number of reuse vectors per reference.

Figure 9 depicts the progress of the algorithm for the load reference of $Z(j,i)$ in a 256×256 matrix multiply loop nest (Figure 1(a)). We have considered a 8KB direct-mapped cache with 32B line size and 8 data elements per cache line. Every iteration point is identified by the index vector (i,k,j). We consider three reuse vectors $\vec{r_1}$, $\vec{r_2}$, and $\vec{r_3}$ of $Z(j,i)$. Reuse vectors $\vec{r_1}$ and $\vec{r_2}$ are self-spatial reuse vectors and $\vec{r_3}$ is a self-temporal reuse vector. $\vec{r_1}$ and $\vec{r_3}$ are the basic reuse vectors generated from SUIF, while $\vec{r_2}$ is generated from our extension to SUIF. Figure 9 shows the contribution of every CME encountered towards the final miss count. Every replacement CME is denoted by 'ReplEqn' followed by the names of the interfering references. Of the 2.1M indeterminate points after considering r_1, only 8192 remain indeterminate after r_2. Considering r_3, we can deduce that all 8192 of these are true cold misses.

	Reuse Vector		
	$\vec{r_1}$: (0 0 1)	$\vec{r_2}$: (0 1 -7)	$\vec{r_3}$: (0 1 0)
Cold CMEs	2097152	8192	8192
ReplEqn_ZZ	0	0	0
ReplEqn_ZY	1835008	261120	0
ReplEqn_ZX	401408	64064	0
Repl. Misses	2236416	325184	0
Definite Misses	2236416	2561600	2569792

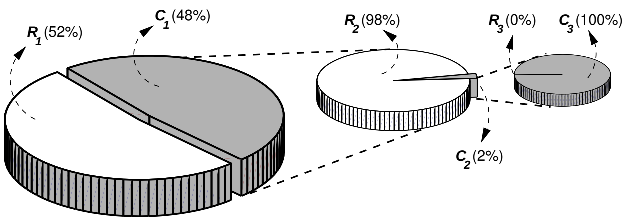


Fig. 9. Using the CME-based algorithm from Figure 7 to find cache misses for the load of $Z(j, i)$ in the matrix multiply loop nest (Figure 1(a)). The diagram and the table shows the progress of the algorithm as reuse vectors are considered one by one, each time zeroing in on the previously indeterminate points. The table shows the solution count of the CMEs and the actual misses found at every stage of considering a reuse vector. C_1, C_2, and C_3 in the diagram represent the cold CME solution points (from the row 'Cold CMEs' in the table) when we consider the reuse vectors $\vec{r_1}$, $\vec{r_2}$, and $\vec{r_3}$ respectively. Similarly, R_1, R_2, and R_3 represent the replacement misses found (from the row 'Repl. Misses' in the table). The indeterminate points are identical to the cold CME solution points. The last row in the table shows the cumulative count of actual misses found so far after each reuse vector is investigated.

4.2 Set-Associative Caches

The preceding miss-finding discussion built intuition by considering a direct-mapped cache. Composing CME solutions into cache miss points is more complex in a set-associative cache. This is because a cache miss occurs in a k-way set-associative cache only when k distinct conflicts occur.

Every solution to the CMEs can be summarized using a triple of the form $(\vec{i}, \vec{j}, n)$. (The set of all such triples is given by the set R' in Figure 7.) The first component $\vec{i}$ of each triple corresponds to the iteration point where a reference (the "victim") potentially suffers a replacement miss along a reuse vector. The second component of the triple, $\vec{j}$, denotes the iteration at which the potentially-conflicting access (the "perpetrator") occurs. The third element of the triple, n, corresponds to the number of "cache wraparounds" there are between the memory addresses of the two potentially-conflicting references. To be precise, the two potentially-conflicting references are separated by (n/k) cache sizes, where k is the associativity of the cache. In this analysis, n will never equal 0 since that is not a conflict but rather a reuse, and reuse will always be summarized in the reuse vectors.

From this triple, we wish to derive distinct miss points. For a particular iteration point $\vec{i}$, all solutions with the same value of n correspond to contention with the same memory line (since they have the same wraparound factor). Thus, to find *distinct* conflicts for an iteration $\vec{i}$, we look for distinct values of n. Note that $\vec{j}$, the cause of the miss, does not impact miss-finding, so we map the space of $(\vec{i}, \vec{j}, n)$ triples down to a space, $R^{''}$, of $(\vec{i}, n)$ pairs (Line 11 in Figure 7).

The cardinality of the set $R^{''}$ corresponds to the total number of conflicts seen, but this is different from the number of cache misses. The points in $R^{''}$ are misses at $\vec{i}$ along $\vec{r}$ if and only if there are at least k (the associativity) elements in $R^{''}$ with $\vec{i}$ as the first component. Hence, only these $\vec{i}$'s are selected as replacement miss points and included in the set M_X (Lines 17-19 in Figure 7). The mapping from $R^{''}$ to M_X performs the following: For each $\vec{i}$, if there are at least k conflicts (for a k-way set-associative cache) then add a point to M_X. If there are less than k conflicts, do not add a point. Note that if there are greater than k conflicts, still only a single point is added to M_X.

The equations generated here represent a set of linear equalities or inequalities. Methods to solve these kind of equations for most practical loops can be found in [Banerjee 1993; Pugh 1992]. Taking the unions and intersections shown in Figure 7 takes polynomial time in the number of elements of the sets. In the next section, however, we have shown how different optimizations can be analyzed without actually solving the equations.

4.3 CME Accuracy

Next, we show the accuracy of our system for finding the cache misses of loop nests using the reuse vectors generated by our current reuse analysis. Table II compares the actual misses (from DineroIII cache simulation) of some example loop nests with the misses measured from CMEs. Actual runtime values of loop bounds, array sizes, and relative base addresses of arrays are used to count the cache misses using CMEs. We have considered an 8KB direct-mapped cache with 32B line size. The loop nests considered include *mmult* (matrix multiply), *gauss* (Gaussian elimination), *sor* (successive over-relaxation), *adi* (ADI kernel after loop fusion and interchange optimizations), *trans* (matrix transpose), *alv* (loop nest from *alvinn* benchmark), and *tom* (loop nest from *tomcatv* benchmark). For all the loop nests the problem size considered is 256 and each array element size is 4 bytes. The table shows that for most of these loop nests very few reuse vectors (average of 2 per reference) are needed to attain accuracy within 1%. Furthermore, the basic reuse vectors given by SUIF are sufficient in all but one case. The inaccuracies found for *gauss* and *trans* are due to the fact that the reuse vectors used are not yet sufficient to represent all the reuse directions present in the loop nest. As a result, CME method finds more cache miss points than are actually present.

5. USING CMES TO GUIDE MEMORY OPTIMIZATIONS

We have implemented our algorithm to generate the CMEs for all the analyzable loops in a program. Our implementation is integrated into the SUIF compiler system [Wilson et al. 1994]. According to the algorithm shown in Figure 2, four different kinds of information are provided as input to our CME generator: (i) the loop indices, (ii) array references, (iii) reuse vectors of the references, and (iv) the reference sequence inside a loop nest. We extract the information about the loop indices and the array references from the SUIF intermediate code representation.

Table II. #Reuse vectors (RVs) used by our CME method to get the calculated miss count within 1% of the actual miss count (measured by DineroIII). SUIF-RV corresponds to reuse vectors extracted from SUIF analysis and Ext-RV corresponds to extra reuse vectors found from our extended reuse analysis. (In this table, 'Max.' stands for 'Maximum' and 'ref.' stands for 'reference'.)

Loop Nest	#Arrays.	Max. #refs to an array	#Refs.	Max. #RVs used per ref.	Distribution of RVs used	
					SUIF-RV	Ext-RV
mmult	3	2	4	3	7	3
gauss	1	5	5	2	4	-
sor	1	6	6	4	11	-
adi	3	3	9	1	9	-
trans	1	4	4	2	6	-
alv	2	2	5	1	5	-
tom	4	2	6	1	6	-

Loop Nest	#Data accesses	#Data cache misses		%Error
		DineroIII	CME	
mmult	67108864	7042336	7042336	0.0
gauss	16744320	1998466	2019682	1.0
sor	387096	8192	8192	0.0
adi	587520	391680	391680	0.0
trans	262144	73456	73732	0.4
alv	183150	14090	14090	0.0
tom	387096	258064	258064	0.0

The reuse analysis of SUIF is extended to generate all the reuse vectors we need. Finally, the exact reference sequence is obtained from the code generator after register allocation is done.

In this section, we show how the CMEs can be manipulated for better cache optimizations and also how to devise efficient compiler algorithms from the analysis of mathematical properties of the CMEs. We illustrate this with the help of two examples. The first example shows how the CMEs can be used to reduce cache misses by changing the base addresses and the column sizes (*i.e.* padding) of the arrays referenced. The second example shows how the equations can be used for efficient block size selection for a blocked loop nest. While these optimizations have been addressed in isolation by past work [Bacon et al. 1994; Coleman and McKinley 1995; Lam et al. 1991], these examples illustrate how CMEs provide a more accurate and unifying framework to drive optimizations.

5.1 Example 1: Padding

This example shows how CMEs are used to find appropriate intra-variable padding (increasing array dimension size) and inter-variable padding (repositioning variable base addresses) that reduce both the self-interferences of a reference and its cross-interferences with other references. We first illustrate how the CMEs are analyzed to drive this optimization with a loop nest from *alvinn* program, a SPECfp benchmark. Then we provide an automated compiler algorithm derived from such analysis along with experimental results showing the usefulness of the precision and generality of the CME framework.

5.1.1 **Padding for a Loop Nest in alvinn**. When we run our equation generator on the loop nests, it generates a collection of CMEs summarizing the memory behavior of each nest. We focus here on one of the analyzable loop nests (shown in Figure 10(a)) which suffers significantly from replacement misses. In this loop, roughly 187,000 out of 306,000 misses are replacement misses. We use the CMEs to eliminate these misses.

Equation 14 and Equation 15 are generated as the replacement CMEs for the reference to *i_h_weights(iu, hu)*. Equation 14 represents the self-interferences for this reference. Equation 15 gives the cross-interferences with the reference *i_h_w_ch_sum_array(iu, hu)*.

$$1221hu + iu = 1221hu' + iu' + nC_s + b \tag{14}$$

$$\begin{aligned} &82110 + 1221hu + iu \\ &\quad = 45480 + 1221hu' + iu' + nC_s + b \end{aligned} \tag{15}$$

where $(iu', hu') \in [(iu-1, hu+1), (iu, hu-1)]$ in Equation 14, and $(iu', hu') \in [(iu-1, hu), (iu, hu-1)]$ in Equation 15. For both the equations, $hu \in [1, 30]$, $iu \in [1, 1221]$, $b \in [-3, 3]$, and $n \in [0, \infty)$. For this example, we assume a direct-mapped cache with cache size, C_s, of 1024 and a line size of 4 elements. In Equation 15, the constant terms 82110 and 45480 are the base addresses of the arrays *i_h_weights* and *i_h_w_ch_sum_array* respectively. The coefficient 1221 is the size of each column of the arrays. The absolute value of the bounds of b is one less than the cache line size. (All the numbers given are in units of data element size.)

Using standard algebraic techniques we can simplify Equations 14 and 15 to the forms shown in Equations 16 and 17 respectively.

$$197hu_{diff} + iu_{diff} = n'C_s + b, \;\; n' \in [-30, \infty) \tag{16}$$

$$790 + 197hu_{diff} + iu_{diff} = n'C_s + b, \;\; n' \in [-65, \infty) \tag{17}$$

In Equation 16, $hu_{diff} = (hu - hu') \in [-29, -1]$ if $iu_{diff} = (iu - iu') = 1$ and $hu_{diff} \in [1, 29]$ if $iu_{diff} = 0$. In Equation 17, $hu_{diff} \in [-29, 0]$ if $iu_{diff} = 1$ and $hu_{diff} \in [1, 29]$ if $iu_{diff} = 0$. The maximum absolute value of hu_{diff} in these equations corresponds to the upper bound of the loop index hu.

Equations 14 and 15 have 232 and 269 solutions respectively. Each solution corresponds to a potential cache miss. We wish to reduce the number of solutions in order to reduce the cache interferences represented by these equations. We intend to do that by changing the base addresses and the column sizes. For this reason, we replace all terms related to the base addresses and the column sizes in Equations 16 and 17 with parameters to get the Equations 18 and 19 respectively. (The ranges of all the variables in Equations 18 and 19 are the same as given in the Equations 16 and 17 respectively.) We consider the constant term in Equation 17 (related to the base address) as a parameter B and the coefficient of hu_{diff} in Equations 16 and 17 (related to the column sizes) as a parameter P. Thus, our goal is to see which values of B and P will result in the fewest number of solutions to the equations (that is, the potential interference misses).

$$Phu_{diff} - 1024n^{'} = (b - iu_{diff}) \tag{18}$$
$$B + Phu_{diff} - 1024n^{'} = (b - iu_{diff}) \tag{19}$$

Though Equations 18 and 19 are independent equations, they are connected through the parameter P. Changing P in one equation will affect the other. We first consider Equation 18 to determine which values of P would eliminate its solutions. We then use one of these P's in Equation 19 to find the values of B that eliminate its solutions.

From basic number theory, we know that if the greatest common divisor of P and 1024 (represented as $GCD(P, 1024)$) does not divide $(b - iu_{diff}) \in [-4, 3]$, then Equation 18 has no solution [Adler and Coury 1995; Banerjee 1993]. When $(b - iu_{diff}) = 0$, $GCD(P, 1024)$ will always divide $(b - iu_{diff})$. For this case, we can again show from number theory that the equation will have no solution if $GCD(P, 1024) < 1024/max(hu_{diff})$ where $max(hu_{diff})$ is the maximum value of hu_{diff} (in this case, 29). Choosing $4 < |GCD(P, 1024)| < 36$, satisfies both of the above criteria and guarantees that Equation 18 will have no solution. We can write $P = 8t, 16t$, or $32t$ where t is any odd positive integer.

By similar reasoning as above, Equation 19 will have no solution if $GCD(B, P, 1024)$ does not divide $(b-iu_{diff})$ for $(b-iu_{diff}) \neq 0$. For $(b-iu_{diff}) = 0$, rewriting the equation as $Phu_{diff} - 1024n^{'} = -B$, the equation will have no solution if $GCD(P, 1024)$ does not divide B. If we choose $B = 8$ and $P = 16t$ we can satisfy all the above criteria and make both Equation 18 and 19 have no solution. In order to have the least amount of padding we choose $P = 208$, the least multiple of 16 above the original value of $P = 197$. (Clearly, we cannot choose to decrease P; that would correspond to negative padding.)

Setting $B = 8$ corresponds to changing the base address of the array *i_h_weights* from 82110 to 81328. Setting $P = 208$ corresponds to changing the size of each column of the arrays from 1221 to 1232. These simple changes in the array layout eliminate all the interference misses in the loop nest of Figure 10(a). (The equations for the reference *i_h_w_ch_sum_array(iu, hu)* are similar and required similar changes to eliminate misses.)

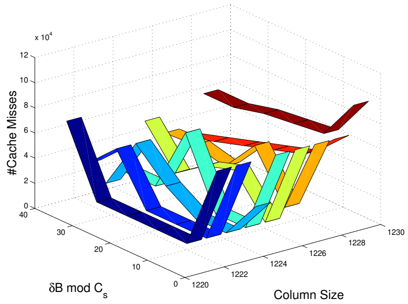
5.1.2 **Compiler Algorithm for Padding**. Here we provide a general padding algorithm, in which we sacrificed some precision from the above analysis to develop an automated approach. This section compares our algorithm to prior work.

The parameters of interest for padding are the column size and the base addresses of the arrays. Our target equations are the replacement CMEs. For our analysis, we consider the interference between two arbitrary references R_X and R_Y. For the self-interference equation of a reference, R_X and R_Y are identical. Let us consider that the references R_X and R_Y access the arrays X and Y whose base addresses are B_X and B_Y respectively. We assume that these conflicting arrays have the same column size C. Using Equation 1, the memory addresses of R_X and R_Y at iteration point $\vec{i}$ can also be written as: $B_X + C(f(\vec{i}) + c) + (f_0(\vec{i}) + c')$ and $B_Y + C(f'(\vec{i}) + d) + (f_0'(\vec{i}) + d')$ respectively, where f, f_0, f', f_0' are linear functions of the loop indices and c, c', d, d' are constants.

The replacement CMEs that correspond to the interferences between two refer-

```
DO iu = 1, 1221
  DO hu = 1, 30
     i_h_weights(iu, hu) +=
         i_h_w_ch_sum_array(iu, hu) * i_h_lrc ;
     i_h_w_ch_sum_array(iu, hu) *= ALPHA ;
```

(a)



(b)

Fig. 10. (a) The *alv* loop, a loop nest from *alvinn* benchmark. (b) A surface plot of #cache misses for different column sizes and base address positioning of the two arrays accessed in the *alv* loop. (δB is the difference in base addresses of the two arrays.) This plot shows the sensitivity of cache misses in the *alv* loop to different padding choices. The irregular pattern demonstrates the necessity of accurate analysis.

ences that access the same array are of the following type (called Type 1 in Fig. 11):

$$C(\delta f + c - d) - nC_s = b - (\delta f_0 + c' - d') \tag{20}$$

where $n \neq 0$, $b \in [-(L_s-1), (L_s-1)]$, $\delta f = f(\vec{i}) - f'(\vec{j})$, and $\delta f_0 = f_0(\vec{i}) - f_0'(\vec{j})$. The range of the intervening points $\vec{j}$ is determined by the corresponding reuse vector. From straightforward number theory [Adler and Coury 1995; Banerjee 1993], this linear Diophantine equation has no solution if the following two conditions are satisfied:

1. $\gcd(C, C_s) > \max|b - (\delta f_0 + c' - d')|$
2. $\gcd(C, C_s) < C_s / \max|\delta f + c - d|$
   if $(b - (\delta f_0 + c' - d')) = 0$

All the other replacement CMEs are of the following type (say, Type 2):

$$(B_X - B_Y) + C(\delta f + c - d) - nC_s = b - (\delta f_0 + c' - d') \tag{21}$$

Again from number theory, Equation 21 has no solution if the following conditions are satisfied:

3. $\gcd(|B_X - B_Y|, C, C_s) > \max|b - (\delta f_0 + c' - d')|$
4. $\gcd(C, C_s) > |B_X - B_Y| \bmod C_s$ if $(b - (\delta f_0 + c' - d')) = 0$

Our algorithm finds appropriate values of C and $|B_X - B_Y|$ that satisfy all four conditions. Since cache size C_s is a power of two, the GCDs in all the conditions are also powers of two. We consider $C = 2^x t_1$ and $|B_X - B_Y| = 2^y t_2$ where t_1, t_2 are nonzero odd positive integers. The following constraints follow from the four conditions:

$$\text{From Condition 1}: x > \lg(\max|b - (\delta f_0 + c' - d')|)$$
$$\text{From Condition 2}: x < \lg(C_s / (\max|\delta f + c - d|))$$

```
Algorithm Find_ColumnSize_and_BaseAddresses
Input: CMEs of the loop nest
Output: (i) The column size C of the arrays
        (ii) Base address B_X for every array X accessed in the loop nest
{
For each reference /* Say the array accessed by this reference is X */
   For each reuse vector
      For each replacement CME
         Lb = lg(max|b − (δf_0 + c' − d')|);
         Ub = lg(C_s / max|δf + c − d|);

         If (Type 1 equation)
            update lower_bound(x) with Lb
            update upper_bound(x) with Ub
         Else
         /* Say the other array involved in this CME is Y */
            update lower_bound(x) with Lb
            update lower_bound(y(XY)) with Lb
            add constraint (lg(|B_X − B_Y| mod C_s) < x)
Get_paddings from above ranges and constraints
}
```

Fig. 11. Algorithm for padding arrays and setting base addresses to reduce cache interferences.

$$\text{From Condition}\,3: x, y > \lg(\max|b - (\delta f_0 + c' - d')|)$$
$$\text{From Condition}\,4: \lg(|B_X - B_Y| \bmod C_s) < x$$

Once x is known, the compiler can easily choose any value of t_1 such that C is at least equal to the original column size. Once y is known, it can choose any value of t_2 such that: (i) $|B_X - B_Y|$ is at least equal to the size of the arrays lying between B_X and B_Y, and (ii) $|B_X - B_Y|$ satisfies Condition 4.

Based on these constraints, we have developed the algorithm sketched out as pseudo-code in Figure 11. The core of the algorithm finds the x and y values. For every pair of conflicting arrays X and Y we need to find $y(XY)$, but we need only one x since all the array column sizes are assumed to be same. The algorithm iterates through each equation and updates the bounds of x and y's according to the constraints. The max values are easily evaluated from the ranges of the intervening points $\vec{j}$; these depend on the reuse vector. Finally, the minimum possible C and inter-array paddings are evaluated satisfying the constraints on x and y's in *Get_paddings*. This algorithm guarantees a solution if there exist x, y's that satisfy all the conditions. In practice, however, most cases satisfy these conditions.

As discussed in Section 4, our CME methods let us choose precision by choosing how many reuse vectors to consider. We have implemented the described algorithm considering only the nearest reuse vector for every reference.[2] The algorithm is

[2]For even more precise results, one could increase the number of reuse vectors considered. Also conditions 2 and 4 are rare cases and hence can be ignored if needed.

Table III. Impact of our padding algorithm: Data cache misses in the original and optimized code. Both replacement miss and total miss counts are shown. ('Repl.' in the table stands for 'Replacement'.)

Loop Nest	#Data accesses	#Data cache misses				%Reduction in cache misses	
		Original		Optimized			
		Repl.	Total	Repl.	Total	Repl.	Total
mmult	67108864	7017760	7042336	3454304	3478880	50.8	50.6
gauss	16744320	1974689	1998466	883473	901048	55.3	54.9
sor	387096	0	8192	0	8192	-	0
adi	587520	367104	391680	0	24576	100.0	93.7
trans	262144	57344	73456	57344	73456	0.0	0.0
alv	183150	4880	14090	0	9240	100.0	34.4
tom	387096	225552	258064	0	32512	100.0	87.4

quite fast—quadratic on the number of references, which is a small number in all practical loop nests.

Table III shows the results of applying this padding algorithm to our benchmark suite. Of the six programs with non-zero replacement misses, our padding algorithm dramatically reduces replacement misses in all but one. In particular, both the precision and generality of the CME approach allow our algorithm to eliminate more conflict misses than the padding methods recently described by Rivera and Tseng [1998]. For example, their methods cannot decrease any conflict misses for the *mmult* loop (Figure 1(a)), because they do not address inter-array padding for the Y(j,k) and Z(j,i) references that are not uniformly generated. The Rivera-Tseng method also lacks sufficient generality to handle replacement misses between references of the form A(i,j) and B(i,j) as in *alv* (Figure 10(a)); this is because it does not attempt intra-array padding to reduce cross-interferences. In contrast, our algorithm decreased conflict misses in these loops by 50.8% and 100% respectively.

Figure 10(b) shows the sensitivity of cache misses in the *alv* loop to different choices of column sizes and base addresses. Such an irregular pattern makes it difficult to find effective padding choices through a heuristic, iterative framework. Manipulating CMEs allows our algorithm to directly and precisely identify the padding values that will eliminate all replacement misses in this case. The generality of the CME framework also allows our algorithm to simultaneously consider (and eliminate) both self- and cross-interferences.

Thus, we have shown how effective compiler optimizations can be derived directly from the solution properties of linear Diophantine equations. Our padding algorithm only needs the CMEs, not their explicit solutions. Furthermore, the precision and generality of the CME framework allow our algorithm to reduce more conflict misses than prior approaches.

5.2 Example 2: Selecting Block Size for Loop-Tiling

This case study deals with a familiar loop optimization for scientific code: *array blocking* (or *tiling*). This technique tries to eliminate capacity misses by reordering accesses so that accesses to reused data are closer together in the iteration space. There has been significant research on how to restructure loop nests for tiling [Carr

```
DO kk = 1, N, T_k
  DO jj = 1, N, T_j
     DO i = 1, N
        DO k = kk, min(kk+T_k-1, N)
           r = X(k, i);
           DO j = jj, min(jj+T_j-1, N)
              Z(j, i) += r * Y(j, k);
```

Fig. 12. Blocked matrix multiplication loop nest.

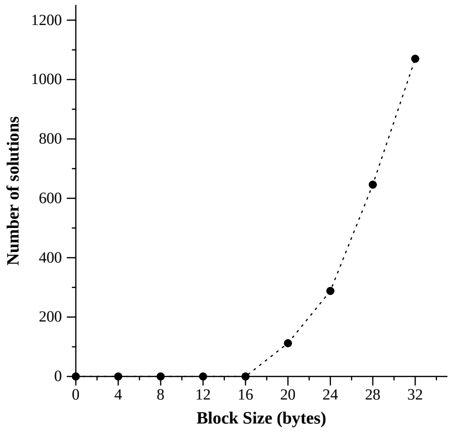


Fig. 13. Solutions to self-interference equation of $Y(j, k)$.

and Kennedy 1992; Wolf and Lam 1991; Wolfe 1989]. This prior work typically ignores the effects of conflict misses arising due to the low associativity of real caches. More recent work noted that cache conflicts can have significant impact on performance and are highly sensitive to the problem size and block size [Lam et al. 1991]. This led to recent research on choosing appropriate tile sizes or block sizes that would reduce conflict misses as well [Coleman and McKinley 1995; Lam et al. 1991]. These papers concentrate on eliminating the self-interference misses that are found to dominate conflict misses in tiled code. They develop specific algorithms for choosing a blocking factor based on program and cache parameters. In the first subsection, we will show how to use CMEs, a more general framework, to find the block size that will eliminate all the self-interference misses in a tiled code. In fact, our framework can be used to handle both self and cross-interferences as discussed in the second subsection. (The analysis described here can be translated to an automated algorithm as we did for the padding example.)

5.2.1 **Block Size Selection for Blocked Matrix Multiply**. In this example, we start with an already-blocked loop nest. We explain our method for the blocked matrix multiplication loop nest given in Figure 12. Our analysis could be easily generalized for all other tiled loops handled in previous work. In Figure 12, T_k by T_j is the block size which this example works to maximize without incurring any self-interference misses.

In order to roughly match the analysis given by Lam et al. [1991], we consider a 4 KB (512 element) direct-mapped cache with 8 Byte (1 element) lines. We assume matrices of size 295 by 295 double-word elements. The predominant source of misses in the tiled code is self-interference misses in *Y(j, k)*. In fact, after a certain block size, these self-interferences outweigh the performance gain expected from increasing block size [Lam et al. 1991]. The self-interference equation of *Y(j, k)* for each execution of the blocked code in Figure 12 is given by Equation 22.

$$295k + j \; = \; 295k^{'} + j^{'} + nC_s + b \tag{22}$$

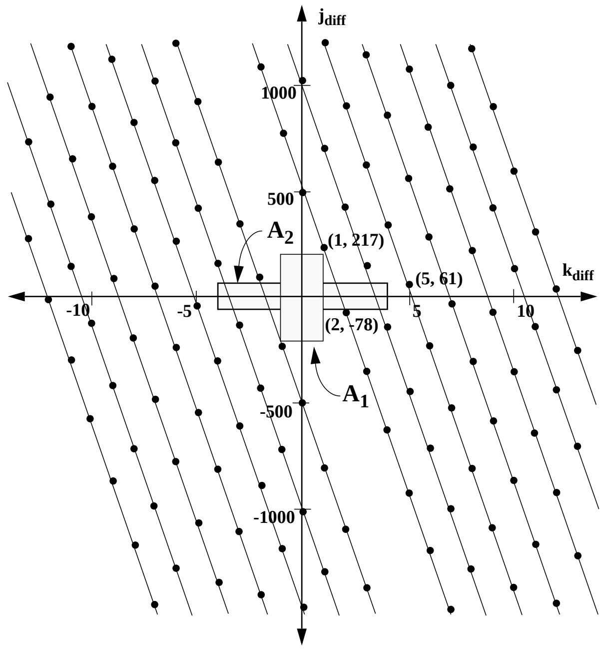


Fig. 14. Solution plot of the self-interference equation of $Y(j, k)$. The parallel lines correspond to sets of solutions for different values of n. Dots along each line represent the integral solution points (interference misses). Shaded regions correspond to possible blocks with no self-interference.

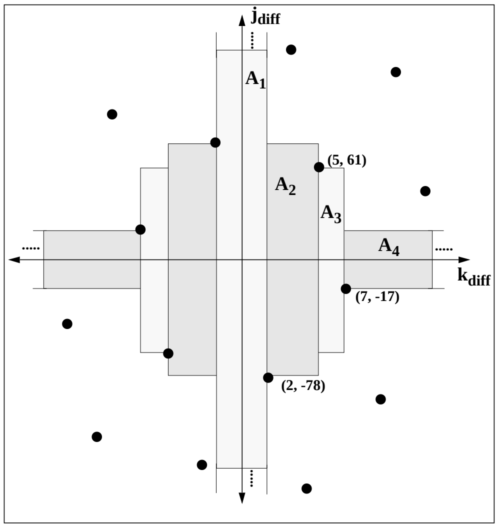


Fig. 15. Finding regions without integral solution points.

where $(k,j) \in [(1,1),(T\_k,T\_j)]$, $(k',j') \in [(1,1),(k,j-1)]$ or $[(k,j+1),(T\_k,T\_j)]$, $b \in [-0,0]$, $C_s = 512$ and n is any integer except 0. As before, C_s is the cache size, and the absolute value of the bounds of b is cache line size minus one. Equation 23 is a simplified version and includes all the solutions of Equation 22.

$$295k_{diff} + j_{diff} \; = \; 512n \tag{23}$$

where, $|k_{diff}| = |k - k'| < T\_k$ and $|j_{diff}| = |j - j'| < T\_j$.

Figure 13 plots the number of solutions of Equation 22 for different square block sizes. These data are consistent with the self-interference misses of blocked matrix multiplication presented by Lam et al. [1991]. There are no self-interferences of *Y(j, k)* until a block size of 16 by 16. Thereafter, it increases drastically with increasing block size. Our goal is to use the CMEs to identify the largest block size with no self-interference. That means we need to find the largest value of T_k by T_j such that Equation 23 has no solution. Lam et al. considered only square blocks, but Coleman and McKinley [1995] showed that we can sometimes get better performance by considering more general rectangular blocks. CMEs can guide us to choose the best block, square or rectangular.

Figure 14 illustrates the block-selection problem. The lines in the figure are plots of Equation 23, but without any constraints on k_{diff} or j_{diff}, for different values of n. The bold dots show the self-interference instances, that is the solution points for integral values of k_{diff} and j_{diff}. Since $|k_{diff}| < T\_k$ and $|j_{diff}| < T\_j$, Equation 23 will have no solution if there are no integral solution points strictly within the region defined by $k_{diff} = T\_k, k_{diff} = -T\_k$ and $j_{diff} = T\_j, j_{diff} = -T\_j$. Thus, such an *empty region* corresponds to the selection of a block of size T_k by T_j which would eliminate all self-interferences. For example, the shaded region A_1 in Figure 14 is one such empty region. Our aim is to find the empty region with the largest area.

For our explanation, we only concentrate on the right-half of Figure 14 (*i.e.* $k_{diff} \geq 0$) as the left half is just the reflection of that through the origin. In order to explain our method, consider the zone around the origin in the figure. We start with the tallest, thinnest empty region A_1 where $T\_k = 1$ and $T\_j = 217$ since $(k_{diff}, j_{diff}) = (1,217)$, an integral solution point, has to be excluded. We then try to expand that empty region along the k_{diff} axis until we hit an integral solution point. Here, since $(2,-78)$ is an integral solution point, A_1 cannot be expanded beyond $k_{diff} = 2$. (Note that, we are only interested in expanding at integral steps as the dimensions of a block can only be integers.) Now the region is shrunk along the j_{diff} axis to exclude that solution point; we then expand again along k_{diff} as much as possible without including any integral solution point. So, here we shrink A_1 along the j_{diff} axis such that $j_{diff} = 78$ in order to exclude $(2,-78)$. Subsequently, the region is expanded along k_{diff} axis until $k_{diff} = 5$. It cannot be expanded beyond $k_{diff} = 5$ as the integral solution point $(5,61)$ has to be excluded. As a result, we obtain the region A_2. We repeat the above process of finding the empty regions by shrinking along the j_{diff} axis and expanding along the k_{diff} axis to find the *maximal empty regions* which are defined as the empty regions whose area cannot be increased without decreasing the height. The process is continued until a maximal empty region is generated whose area exceeds the cache size or is equal to zero. Table IV lists the maximal empty regions A_1 through A_7 found by the

Table IV. Blocks with no self-interference (from our algorithm).

Region	T_k	T_j	Block size (T_k * T_j)
A_1	2	217	434
A_2	5	78	390
A_3	7	61	427
A_4	26	17	442
A_5	33	10	330
A_6	59	7	413
A_7	151	3	453

above algorithm. Figure 15 depicts the formation of the first four maximal empty regions. We define a *limiting point* to be an integral solution point that prevents any further expansion of a maximal empty region along the k_{diff} axis. For instance, $(5, 61)$ is the limiting point for the maximal empty region A_2. Each of the empty regions define a block size with no self-interference. For example, region A_4 defines the block size of T_k by $T\_j = 26$ by 17 which clearly includes the square block solution of 16 by 16 found experimentally from Figure 13. The empty region with the largest area is clearly A_7. For the largest block size with even dimensions, we choose the region with the maximum area that have even dimensions which, in this case, is the block of size 26 by 16. Our algorithm does not require finding integral solution points for all n, j, k. Rather, we need only identify the limiting points as we stretch the rectangles.

5.2.2 **Selecting Block Size to Eliminate both Self and Cross Interferences.** Here we present a novel method, using CMEs, where we integrate block size selection and padding in order to reduce both self and cross interferences. We briefly describe the process for a blocked matrix multiply loop nest, determining a block size of T_k by T_j. Say, we want to reduce self-interferences of $Y(j, k)$ and also its cross-interferences with $Z(j, i)$. Hence, the equations we analyze are $Y(j, k)$'s self-interference equation and the cross-interference equation with $Z(j, i)$. For the blocked code they are:

$$Ck_{diff} - nC_s = b - j_{diff}, \quad \text{where} \quad k_{diff} < T\_k, \quad j_{diff} < T\_j \tag{24}$$

$$(B_Y - B_Z) + C(k - i') - nC_s = b - j_{diff}, \quad \text{where} \quad k \in [0, N-1], \quad i' \in [0, N-1], \quad j_{diff} < T\_j \tag{25}$$

The variables to be optimized here are T_k, T_j, B_Y, and B_Z. There are a lot of ways one can follow here. We have developed an algorithm where we first find T_k, T_j from Eqn. 24, and then optimize B_Y, B_Z from Equation 25 by an algorithm similar to Figure 11. The block size selection algorithm conceptually finds all combinations of (T_k, T_j) that ensure no solution to Eqn. 24 for a direct-mapped cache. For a k-way set-associative cache, it finds (T_k, T_j)'s that allow at most (k-1) solutions to Eqn. 24. As our algorithm combines padding along with selecting block sizes, it would be interesting to compare this algorithm with the block size selection algorithm presented by Coleman and McKinley [1995].

6. COMPUTATIONAL COMPLEXITY

Here we discuss the computational requirements of generating and using CMEs. The first step in generating CMEs is calculating reuse vectors. If the number of array references in a loop nest of depth n is n_{ref} and d_{max} is the maximum number of dimensions of any of those arrays, the worst case complexity of calculating all the reuse vectors is $O(n_{ref}^2 \times (max(n, d_{max}))^3)$. Each reuse vector can be calculated by simple manipulations on a matrix of size $max(n, d_{max})$ which takes $O((max(n, d_{max}))^3)$ time in the worst case. Moreover, in the worst case, we need to do n_{ref}^2 of such matrix manipulations, corresponding to the group reuse vector calculation between every pair of references in the loop nest.

Once the reuse vectors are calculated, the time taken to generate all the CMEs of the loop nest is given by $O(n \times d_{max} \times n_{eqn})$, where $n_{eqn} = \#\text{Equations} = n_{rv} \times n_{ref}$, n_{rv} = Total #reuse vectors of all the references.

We have implemented our CME generator in the SUIF compiler system [Wilson et al. 1994] and have tested our system on SPECfp and other benchmarks. In these experiences, we have found the CME generator to be quite fast. In fact, for the SPECfp benchmarks, CME generation always executes in less than 10s per program on an SGI/INDY with a 133MHz MIPS R4600 CPU.

Once the CMEs are generated, further computational requirements depend on the methodology used to develop an optimization using CMEs. For example, the algorithm presented in Section 5.1.2 simply computes GCDs, and is a linear algorithm in the number of loop indices.

The equations generated in Section 3 represent a set of linear equalities or inequalities. Fast methods to solve these kind of equations for most practical loops have been demonstrated in Banerjee [1993] and Pugh [1992; 1994]. Taking unions and intersections to find the cache misses takes polynomial time in the number of elements of the solution sets. However, in most optimization applications, we do not need to find the cache miss instances explicitly.

Many loop optimizations and estimations need the total number of cache misses rather than the iteration points where the cache misses occur. This involves finding the number of integer solutions to the unions and intersections of the CMEs. The method to find the number of integer points in unions and intersections of closed convex polyhedrons defined by most practical scientific loops, as given in Clauss [1996], could be used. For the closed convex polyhedrons defined by the CMEs, the method takes polynomial time for fixed loop bounds. It can also handle parametric loop bounds for practical loop nests.

7. DISCUSSION: FUTURE EXTENSIONS AND APPLICATIONS

In order to make our analysis framework more general, it needs to handle the effects of multiple loop nests in the program. Also, to make the framework more effective, we should be able to automatically solve or at least find the number of solutions to the CMEs in parametric form. Finally, if we can automate the analysis of the equations for cache optimizations, it can be effectively incorporated within a compiler for better program performance. In the following sections we discuss the above issues in further detail and also investigate the potentially broad application scope of our CME-analysis framework.

7.1 Inter-nest Analysis

The methods presented here can be followed to generate equations taking inter-nest effects into account, once efficient methods are developed to calculate reuse vectors across loop nests. Fortunately, most inter-nest misses occur between adjacent nests [McKinley and Temam 1996]. So it may be enough to find reuse vectors only between adjacent nests for most practical purposes.

7.2 Automatic Solving and Analysis of the CMEs

In order to help code optimizations, as described in Section 5, we try to find values of parameters that would eliminate or reduce the number of solutions to the CMEs. As one possible way to automate that analysis, we hope to use the extension of Pugh's *Omega test* [Pugh 1992] to project our constraints on the parameters of interest and find the possible ranges of those parameters that would eliminate or reduce the number of solutions. Another possible way is to calculate the parametric number of solutions as given by Clauss [1996] and then find the values of the parameters that would reduce that number. The Pugh and Clauss methods are reported to be fast for linear constraints derived from most practical loops. The varying ability of these methods to handle parameters would also help us to analyze and optimize loops with parametric loop bounds.

7.3 Applications

As described in Section 5, the CMEs can be used to reduce the number of cache misses due to their precise characterization of the cache misses. Also, the CMEs provide users with a general framework from which different kinds of optimizations to improve memory performance can be applied. We have shown two different optimizations in this article and are studying the techniques to use them for guiding various other optimizations including different kinds of loop transformations. The idea is to use the CMEs to choose the appropriate loop transformation that would yield the best memory performance for the loop nest considered. We plan to automate the analysis of the equations as much as possible, enabling the compiler to harness the preciseness of our framework, for applying effective memory optimizations from a common platform.

Many optimizing transformations are decided on a comparison of the execution times of the original and the transformed code. Therefore, it is essential to estimate, as accurately as possible, the execution time of programs at various granularity levels as program, function, loop, basic block, statement, and expression. Tighter bounds on the worst-case execution time of programs are also necessary for designing efficient embedded systems. One of the biggest problems in closely predicting the execution time is to be able to estimate the data cache performance. The CMEs can be used to calculate quite precisely the number of data cache misses in a loop nest by counting the number of solutions to them. Besides providing an estimate of the number of cache misses, the CMEs can also provide indicators for an optimizer to the origin of most of the cache misses.

Finally, we also hope to use the equation framework to improve the performance of cache simulation tools. Essentially, CMEs can accelerate simulations by identifying cache misses and summarizing much of a program's memory behavior at compile-time, before cache simulation is run.

8. RELATED WORK

Realizing the importance of cache behavior in numerical codes, extensive research has focussed on improving the cache performance of numerical programs. Most of the previous work explores the techniques to reduce capacity misses in scientific loops [Gannon et al. 1988; Irigoin and Triolet 1988; Wolfe 1989; Eisenbeis et al. 1990; Wolf and Lam 1991; Carr and Kennedy 1992; Li and Pingali 1992; Banerjee 1993; McKinley et al. 1996; Carr and Lehoucq 1995]. For example, several researchers have described the popular technique of loop tiling to reduce capacity misses [Wolfe 1989; Carr and Kennedy 1992; Carr and Lehoucq 1995; Irigoin and Triolet 1988; Eisenbeis et al. 1990]. There are also several case studies that report the severe adverse effects of cache interferences or conflicts on cache performance [Lam et al. 1991; Sugumar and Abraham 1993; Temam et al. 1994; McKinley and Temam 1996]. For instance, recently McKinley and Temam [1996] performed a study of the locality characteristics of numerical loop nests and found that conflict misses comprise half of all cache misses and are the most significant sources of intra-nest misses. In addition, some work shows that cache interferences can vary wildly with slight variations in problem size and base addresses [Lam et al. 1991; Bacon et al. 1994; Navarro et al. 1994]. However, cache conflicts are difficult to predict and estimate as it requires a detailed analysis of the data mapping in cache and the data referencing patterns. In fact, there are relatively few studies on analyzing and reducing interference misses. More generally, there has been no work on precisely and analytically representing the cache misses of a loop nest as presented in this paper. Our work shows how such a precise, analytical representation could guide optimizations for reducing cache misses including obscure interference misses in a methodical way.

The methods presented by Gannon et al. [1988] and Gallivan et al. [1988] can be used to estimate the amount of local memory needed by array references in a loop nest. Their focus was on optimizing for a software-controlled local memory rather than for a hardware-controlled cache memory. Porterfield [1989] described a technique for estimating the hit rate of a fully-associative LRU cache in a uniprocessor system. None of these papers, however, consider the effect of cache conflicts. Later, Ferrante et al. [1991] provided closed-form formulae that bound the number of distinct accesses and distinct lines for an array reference in a loop nest. These bounds provide an estimate of the number of cache misses in a loop nest. They have also included some suggestions on how to consider interferences. All of these projects, however, only provide rough estimates of the cache misses, and it is not possible to find the appropriate padding amounts, base addresses, and tile sizes as shown in the optimizations presented in this paper. The CMEs, however, can also be used to estimate the number of cache misses in a loop nest more accurately than presented in previous work. In fact, the CMEs can be used to get separate, precise estimates of cache misses of each type and/or from every source, which in turn helps to determine the cache-performance bottlenecks.

Temam et al. [1993;1994] provide a more accurate analysis than Ferrante et al. [1991] for predicting and estimating cache interferences. However, the methods described by Temam et al. [1994] handle a much smaller subset of array references than we do. For example, their technique does not consider the reference $A[i_1 + i_2]$

where i_1 and i_2 are loop indices. Then, due to several approximations, their process is not as precise as ours. Finally, as they only estimate the number of cache misses, it is not clear how the expressions presented in the paper could be used directly to guide optimizations as discussed in this paper.

Some of the optimizations described in this paper have been addressed in isolation in previous work. Bailey [1992] analyzes the behavior of a direct-mapped cache with strided data access to estimate the cache efficiency. Subsequently, cache efficiency is used by the compiler to detect unfavorable strides and determine automatically the necessary padding for array dimensions, thereby reducing cache set conflicts. In the parallel domain, Torrellas et al. [1990] have suggested alignment of arrays to cache line boundaries to reduce false sharing. Later, Bacon et al. [1994] developed a padding algorithm for selecting efficient padding amounts, which takes into account both cache and TLB (Translation Lookaside Buffer) effects collectively within a single framework. The goal of the algorithm is to avoid set conflicts in the cache and TLB for a given loop nest. Recently, Rivera and Tseng [1998] presented various heuristic techniques for inter- and intra-array padding optimizations. CMEs, on the other hand, allow us to apply padding optimizations in a more general framework along with other kinds of optimizations. Moreover, due to the precision inherent in CMEs, CME-based padding algorithms can sometimes eliminate more conflict misses than earlier work.

There are also some papers on choosing problem-size dependent tile sizes that eliminate self-interference and capacity misses in a tiled loop nest. Lam et al. [1991] present cache performance data for tiled matrix multiply and describe a model for evaluating cache interference. They provide an algorithm which selects the largest square tile size that avoids self-interference. However, the algorithm presented by Coleman and McKinley [1995] can find more effective data-dependent rectangular tiles and improves execution times over Lam et al. [1991]. We have shown how our general framework of CMEs can also be used to choose such efficient rectangular tile sizes, given a particular tiled loop nest.

Finally, there has been some work on automatic analysis and counting the number of solutions to a set of linear equalities and inequalities. This complementary work would help to automate the analysis of the CMEs. Pugh [1992] describes the Omega test, which determines whether there is an integer solution to an arbitrary set of linear equalities and inequalities. One of the important extensions to Omega test, relevant to the analysis of CMEs, deals with symbolic projection of a set of constraints onto the variables of interest. The Omega test has been further extended in Pugh [1994] by considering a set of constraints as Presburger formulas. Recently, Clauss [1996] has provided an exact method for counting number of integer points in polytopes, based on the determination of *Ehrhart Polynomials*. While our work is mainly concerned with the compiler and architectural implications of caching behavior by deriving the CMEs, their work focuses on the mathematical aspects of solving such equations.

9. CONCLUSIONS

This paper demonstrates the use of Cache Miss Equations as a means of precisely characterizing both cold and replacement misses within a loop nest. Our method involves extending traditional reuse analysis in order to generate a set of linear

Diophantine equations whose solutions comprise potential cache misses for the loop nest. We then describe algorithms for identifying cache misses and provide experimental results on the accuracy of our method for loop nests taken from SPECfp benchmarks. Finally, we present examples of CME applications to show the effectiveness of the precision and generality of our CME framework.

The CME framework provides an excellent compile-time platform for a wide variety of applications ranging from performance estimation to code optimizations within a single, unified and precise framework. By automating CME analysis within the compiler, one can guide compiler memory optimizations. Overall, Cache Miss Equations provide an unique, systematic framework for accurately assessing the frequency and causes of cache misses in loop-oriented code; this general and precise foundation will serve as an enabling technology for effective cache optimizations in the future.

APPENDIX

A. REPRESENTING SET OF POTENTIALLY-INTERFERING POINTS

The set of potentially-interfering points are represented by the ranges given in Equation 7. Each of these ranges are again represented by sets of equalities and/or inequalities. Here we describe such a representation for the range $[\vec{p}, \vec{i})$ of Equation 7; the other ranges are represented similarly.

Before presenting the most general form, the basic idea is explained with the help of an example range from the matrix multiply loop nest of Figure 1(a). The region defined by the range $\vec{j} = [(2,18,26),(16,12,24))$ is divided into five convex regions, namely R_1, R_2, R_3, R_4, and R_5 as shown in Figure 16.The convex regions are formed by sequentially considering all the iterations of the loop nest from the starting iteration $\vec{p} = (2,18,26)$ to the last iteration $\vec{i} = (16,12,24)$. A new convex region is formed when the current set of points cannot be expanded any further without violating the convexity. Each of these convex regions is represented with a set of constraints defining a single range for each of the loop indices as shown in Equation 26.

$$\begin{aligned}
\vec{j} \;&\in\; [(2,18,26),\ (16,12,24)) = \{R_1, R_2, R_3, R_4, R_5\} \\
R_1 : \quad & \Big[(i,k) = (2,18);\ \ 26 \ \le j \le\ 31\Big] \\
R_2 : \quad & \Big[i = 2;\ \ 19 \ \le k \le\ 31;\ 0 \ \le j \le\ 31\Big] \\
R_3 : \quad & \Big[3 \ \le i \le\ 15;\ 0 \ \le k \le\ 31;\ 0 \ \le j \le\ 31\Big] \\
R_4 : \quad & \Big[i =\ 16;\ 0 \ \le k \le\ 11;\ 0 \ \le j \le\ 31\Big] \\
R_5 : \quad & \Big[(i,k) = (16,12);\ \ 0 \ \le j <\ 24\Big]
\end{aligned} \tag{26}$$

Equation 27 presents the most general form of the division of the range $\vec{j} = [\vec{p}, \vec{i})$ in an n-deep loop nest. In Equation 27, $\vec{j} = (j_1, j_2, \cdots, j_n)$, $\vec{p} = (p_1, p_2, \cdots, p_n)$, and $\vec{i} = (i_1, i_2, \cdots, i_n)$. Since $\vec{p} \prec \vec{i}$, we assume, without loss of generality, that the k^{th} loop is the outermost loop such that $p_k < i_k$ and so $(p_1, p_2, \cdots, p_{k-1}) = (i_1, i_2, \cdots, i_{k-1})$. In all the convex regions, provided in Equation 26, our assumption

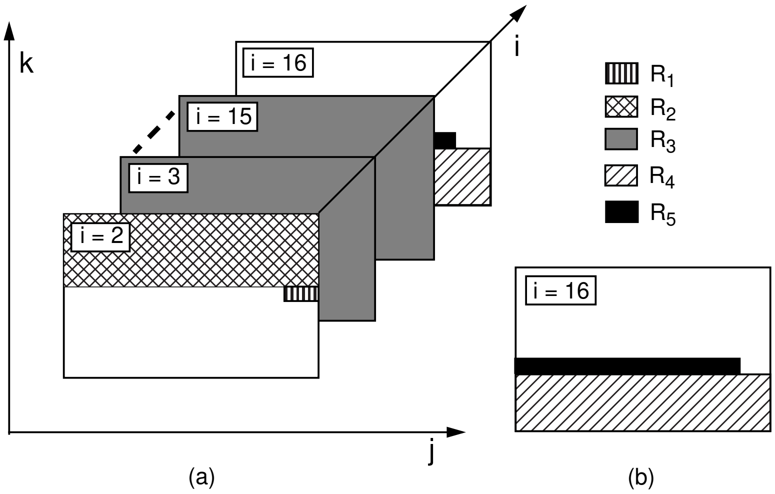


Fig. 16. Division of the range $[(2,18,26),(16,12,24))$ into the convex regions R_1, R_2, R_3, R_4, and R_5. (a) shows the regions in the iteration space of the loop nest of Figure 1(a). (b) shows the regions in the iteration space when $i = 16$ to clarify the area hidden in (a).

of $(p_1, p_2, \cdots, p_{k-1}) = (i_1, i_2, \cdots, i_{k-1})$ holds and so we have avoided restating these equalities. We have also assumed that the lower bounds of the loop indices are $l_1, l_2, \cdots, l_n$ and the corresponding upper bounds are $u_1, u_2, \cdots, u_n$. With these assumptions, Equation 27 represents the range of $\vec{j}$ as the union of $(2k-1)$ convex regions where each convex region is defined by a conjunction of a set of constraints.

$$
\begin{aligned}
& \vec{j} \in [\vec{p},\ \vec{i}) = \{R_1, R_2, \cdots, R_{k-1}, R_k, R_{k+1}, \cdots, R_{2k-1}\} \\
R_1 : \quad & \Big[(j_k, j_{k+1}, \cdots, j_{n-1}) = (p_k, p_{k+1}, \cdots, p_{n-1}); \\
& p_n \ \le j_n \le \ u_n\Big] \\
R_2 : \quad & \Big[(j_k, j_{k+1}, \cdots, j_{n-2}) = (p_k, p_{k+1}, \cdots, p_{n-2}); \\
& p_{n-1} + 1 \ \le j_{n-1} \le \ u_{n-1}; \\
& l_n \ \le j_n \le \ u_n\Big] \\
R_{k-1} : \quad & \Big[j_k = p_k; \\
& p_{k+1} + 1 \ \le j_{k+1} \le \ u_{k+1}; \\
& l_{k+2} \ \le j_{k+2} \le \ u_{k+2}; \\
& \quad \vdots \\
& l_n \ \le j_n \le \ u_n\Big] \\
R_k : \quad & \Big[p_k + 1 \ \le j_k \le \ i_k - 1; \\
& l_{k+1} \ \le j_{k+1} \le \ u_{k+1}; \\
& \quad \vdots
\end{aligned}
$$

$$
\begin{array}{ll}
 & l_{n-1} \le j_{n-1} \le u_{n-1}; \\
 & l_n \le j_n \le u_n \Big] \\
R_{k+1}: & \Big[j_k = i_k; \\
 & l_{k+1} \le j_{k+1} \le i_{k+1} - 1; \\
 & l_{k+2} \le j_{k+2} \le u_{k+2}; \\
 & \vdots \\
 & l_n \le j_n \le u_n \Big] \\
R_{2k-1}: & \Big[(j_k, j_{k+1}, \cdots, j_{n-1}) = (i_k, i_{k+1}, \cdots, i_{n-1}); \\
 & l_n \le j_n < i_n \Big]
\end{array}
\tag{27}
$$

B. THEORY BEHIND THE CACHE-MISS-FINDING ALGORITHM

The methods described in Section 3 generate cold and replacement CMEs along a single reuse vector. Eventually, the effects of all the reuse vectors are combined in the algorithm presented in Figure 7 to find the set of all cache miss instances of a loop nest from the solutions of all of its CMEs. This algorithm is developed based on two theorems presented and proved here. Before presenting the theorems, we first present some lemmas that we use to prove these theorems.

The following lemma shows the relationship between a solution of a cold CME and a cold miss along a reuse vector.

Lemma 1. If a reference R suffers a cold miss at an iteration point $\vec{i}$ along a reuse vector $\vec{r}$ of R, $\vec{i}$ satisfies at least one of the cold CMEs of $\vec{r}$.

PROOF. According to the methods described in Section 3.1.1 for the cold misses along a spatial reuse vector, two equations are generated—one for each of the two cases described there. However, only one equation need to be generated, as in Section 3.1.2, for the cold misses along a temporal reuse vector. As we consider all possible cases for the occurrence of cold misses along a reuse vector, we can claim that every cold miss is captured by at least one of the equations generated in Sections 3.1.1 and 3.1.2. □

In fact, the converse of the above lemma, as stated below, holds as well.

Lemma 2. If an iteration point $\vec{i}$ satisfies any of the cold CMEs for a reuse vector $\vec{r}$ of a reference R, then R suffers a cold miss at $\vec{i}$ along $\vec{r}$.

PROOF. Consider any iteration point $\vec{i}$ that satisfies the cold miss equation from case 1 of Section 3.1.1 if $\vec{r}$ is a spatial reuse vector; otherwise, if $\vec{r}$ is a temporal reuse vector, assume that $\vec{i}$ satisfies the equation generated in Section 3.1.2. Also, assume that $\vec{i}$ does not satisfy the case 2 equation when $\vec{r}$ is a spatial reuse vector. Now, according to the methods used to generate the equations, reference R definitely accesses a new memory line at $\vec{i}$ along $\vec{r}$. Hence, R is guaranteed to suffer a cold miss at $\vec{i}$ along $\vec{r}$ as claimed in Lemma 2.

Now, consider $\vec{r}$ is a spatial reuse vector and $\vec{i}$ satisfies the equation from case 2 and not the case 1 equation in Section 3.1.1. $\vec{i}$ not satisfying case 1 implies that

$\vec{i}$ is not the first iteration point along that vector. But, $\vec{i}$ satisfying case 2 implies that a new memory line is brought in at $\vec{i}$. Hence, there is cold miss at $\vec{i}$ along $\vec{r}$.

Finally, consider the situation when $\vec{r}$ is a spatial reuse vector and $\vec{i}$ satisfies both the case 1 and case 2 equations in Section 3.1.1. An iteration point satisfying the case 1 equation, however, is guaranteed to be the instance of a first access to a memory line by R along $\vec{r}$. Hence, there is a cold miss at $\vec{i}$ along $\vec{r}$. (The method used to generate the case 2 equation does not take any separate action for the iteration points that satisfy the case 1 equation. That is why in this case $\vec{i}$ wrongly satisfies the case 2 equation. So, whenever $\vec{i}$ is not a candidate for case 2, but still satisfies case 2 equation wrongly, $\vec{i}$ is, in fact, a candidate for case 1.) □

Next we provide two more lemmas that depict the relationship between a solution of a replacement CME and a replacement miss along a reuse vector. Before that, we define the terminology of *distinct solutions* of replacement CMEs used hereafter.

Definition 6. A set of solutions to replacement CMEs of a reference R are defined to be *distinct at an iteration point* $\vec{i}$ if they correspond to the interferences of the memory line accessed by R at $\vec{i}$ with other distinct memory lines.

Every solution to the replacement CMEs of a reference R is a vector of the form $(\vec{i}, \vec{j}, n)$ where $\vec{i}$ is the iteration where R might suffer a miss due to the contention with another memory line for the same cache set at $\vec{j}$, and the contending memory lines are separated by (n/k) cache sizes. From the simplification in Section 3.3, we can verify that for an iteration point $\vec{i}$ two solutions with the same value of n correspond to the interference between the same two memory lines. So, mathematically, two solutions with $\vec{i}$ as the first component are distinct if they have different values of n.

Lemma 3. If there is a replacement miss of a reference R at the iteration point $\vec{i}$ along the reuse vector $\vec{r}$, then there exist at least k distinct solutions at $\vec{i}$ in the union of all the solution sets of the replacement CMEs of $\vec{r}$.

PROOF. As the associativity of the cache is k and an LRU replacement policy is used, at least k distinct interferences must be encountered by R between $\vec{i} - \vec{r}$ and $\vec{i}$ before it suffers a replacement miss at $\vec{i}$ along $\vec{r}$. In the method of generating the replacement CMEs described in Section 3.2, we consider every possible interference encountered by R that can prevent the reuse along $\vec{r}$ from being realized at $\vec{i}$. In fact, we generate one equation for each of the potentially interfering references (including R itself). Hence, all the k or more distinct interferences must correspond to k or more distinct solutions to the set of all replacement CMEs. □

The exact converse of the above lemma does not hold. However, when the logic of Lemma 3 is reversed, the following lemma results:

Lemma 4. If an iteration point $\vec{i}$ is not a cold miss point of a reference R along a reuse vector $\vec{r}$ and there exist at least k distinct solution points at $\vec{i}$ in the union of all the solution sets of the replacement CMEs of $\vec{r}$, then R suffers a replacement miss at $\vec{i}$ along $\vec{r}$.

PROOF. The replacement CMEs of $\vec{r}$ do not take any special action for the iteration points at which R suffers a cold miss along $\vec{r}$. But if there is a cold miss at $\vec{i}$

along $\vec{r}$, the reuse $\vec{r}$, in fact, does not exist at $\vec{i}$ and so the replacement CMEs at $\vec{i}$ become meaningless. Since $\vec{i}$ is not a cold miss point of R along $\vec{r}$, there is potential for realizing the reuse represented by $\vec{r}$ at $\vec{i}$. According to the assumption in this lemma, there also exist at least k distinct solution points at $\vec{i}$. It follows from the method given in Section 3.2 that R encounters at least k distinct interfering accesses before the reuse of $\vec{r}$ can be realized at $\vec{i}$. All these interfering accesses occur after the previous access of R along $\vec{r}$ at $\vec{i}-\vec{r}$. Hence, due to the LRU replacement policy, the memory line accessed by R at $\vec{i}-\vec{r}$ will be evicted from the cache before its reuse at $\vec{i}$. Note that, we ignore the presence of any other access of R in between $\vec{i}-\vec{r}$ and $\vec{i}$ to the same memory line, if any, as it implies the presence of another reuse vector at $\vec{i}$. Here we consider misses along $\vec{r}$ and so ignore the presence of other reuse vectors. □

The solution set—the *set of all miss instances*[3]—can be generated with the help of the following theorems which are discussed shortly.

Let us first consider the example iteration space shown in Figure 17(a). For a particular reference, assume that one of its reuse vectors is $\vec{r_1}$, which indicates the reference reuses the same memory line at the iteration points $\vec{i_1}$ and $\vec{i_2}$. Each solution of all the equations corresponds to either a cold miss along $\vec{r_1}$ (if it is a cold miss equation) or a replacement miss (if it is a replacement miss equation), preventing the reuse to be realized at $\vec{i_1}$. Moreover, for a replacement miss, it takes at least k different conflicting memory line accesses between $\vec{i_2}$ and $\vec{i_1}$ to cause the reference to miss at $\vec{i_1}$ along $\vec{r_1}$. This indicates that a non-cold miss point will be a miss point along $\vec{r_1}$ if there are at least k distinct solutions to the replacement CMEs at that point. This is formally stated in the following theorem.

THEOREM 1. *Within a given set of iteration points, the set of cold miss points of a reference R along a reuse vector $\vec{r}$ is given by the union of all the solution sets of the cold CMEs of $\vec{r}$. Again, the set of all replacement miss points of R along $\vec{r}$ is given by the set of points that are not cold miss points along $\vec{r}$ but at each point there exists at least k distinct solutions in the union of all the solution sets of the replacement CMEs of $\vec{r}$. (k is the associativity of the cache.)*

PROOF. Let us denote the equations corresponding to the reuse vector $\vec{r}$ by $eq_1, eq_2, \cdots, eq_n$. eq_1 represents a single cold miss equation if $\vec{r}$ is a temporal reuse vector; otherwise, it represents two cold miss equations, eq_{1a} and eq_{1b}, when $\vec{r}$ is a spatial reuse vector. $\{eq_m : 2 \leq m \leq n\}$ represent the replacement miss equations of the reference. Let the solution set of eq_m be given by S_m. If $\vec{r}$ is a spatial reuse vector, we consider S_1 as the union of the solution sets of the equations eq_{1a} and eq_{1b}. If C_r denote the set of all cold miss instances of R along $\vec{r}$, we first need to show $C_r = S_1$.

$$
\begin{aligned}
\text{Let } \vec{i} \in C_r &\Rightarrow \vec{i} \in S_1 \text{ (follows from Lemma 1)} \\
&\Rightarrow C_r \subseteq S_1 \\
\text{Let } \vec{i} \in S_1 &\Rightarrow \vec{i} \in C_r \text{ (follows from Lemma 2)} \\
&\Rightarrow S_1 \subseteq C_r
\end{aligned}
$$

[3]The *set of all miss instances* of a reference is comprised of all the iteration points at which that reference suffers a cache miss.

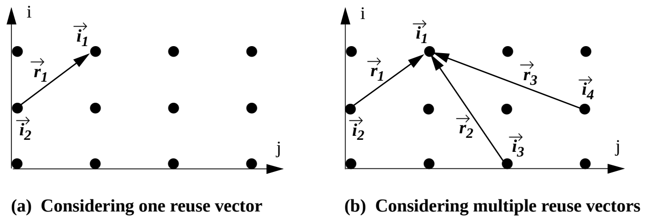


Fig. 17. Illustration of (a) Theorem 1 and (b) Theorem 2 in the iteration space of a 2D loop nest.

$$\text{Hence,} \quad C_r = S_1$$

Let X_r be the set of all replacement miss points of R along $\vec{r}$. Say, S is the set of points where at each point there exist at least k distinct solutions from $\cup_{m=2}^{n} S_m$. Eliminating the cold miss points from S we get the set $S_r = S - C_r$. So, we need to show $X_r = S_r$.

$$\begin{aligned} \text{Let } \vec{i} \in X_r &\Rightarrow \vec{i} \in S_r \text{ (follows from Lemma 3)} \\ &\Rightarrow X_r \subseteq S_r \\ \text{Let } \vec{i} \in S_r &\Rightarrow \vec{i} \in X_r \text{ (follows from Lemma 4)} \\ &\Rightarrow S_r \subseteq X_r \\ \text{Hence,} \quad & X_r = S_r \end{aligned}$$

□

Now, let us consider the example iteration space in Fig 17(b). We assume that the reference has the reuse vectors $\vec{r_1}$, $\vec{r_2}$, and $\vec{r_3}$, which means it accesses the same memory line at the iteration points $\vec{i_1}, \vec{i_2}, \vec{i_3}$, and $\vec{i_4}$. The reference can reuse data at $\vec{i_1}$ if the data remains in cache after being accessed at either $\vec{i_2}$ or $\vec{i_3}$ or $\vec{i_4}$. Intuitively, we can argue that if the reuse $\vec{r_3}$ is not realized at $\vec{i_1}$ there is definitely a miss at $\vec{i_1}$ irrespective of the presence of the other two reuse vectors. Similarly, if $\vec{r_3}$ is realized at $\vec{i_1}$ there is definitely a hit at $\vec{i_1}$. This is because the memory line accessed by R at $\vec{i_1}$ was accessed most recently at $\vec{i_4}$ and it suffices to check whether the memory line will remain in cache or not after that access. Based on this observation, we present Theorem 2. Theorem 2 provides a mechanism to combine the effects of all the reuse vectors in order to determine the definite cache misses of a reference within a loop nest. Before presenting Theorem 2, we define *lexicographical ordering* for the reuse vectors.

Definition 7. Reuse vector $\vec{r_1} = \vec{i} - \vec{i_1}$ is defined to be *lexicographically* shorter than $\vec{r_2} = \vec{i} - \vec{i_2}$, if and only if $\vec{i_1} \succ \vec{i_2}$

THEOREM 2. *Consider a reference R. If there exists no reuse vector of R shorter than $\vec{r}$ for a given set of iteration points I the following holds:*

(*1*) *The set of replacement miss points of R in I along the reuse vector $\vec{r}$ are the definite replacement miss points of R.*

(*2*) *All the other non-cold miss points in I are definite hit points.*

(*3*) *No absolute decision can be taken for the cold miss points in I along $\vec{r}$ unless there exists no reuse vector longer than $\vec{r}$, in which case, these cold miss points are the definite cold miss points of R.*

PROOF. (1) We prove this statement by contradiction. Say, the statement is not true. Assume $\vec{i}$ in I is a replacement miss point of R along the reuse vector $\vec{r}$. So, the memory line accessed by R at $\vec{i}-\vec{r}$ is evicted from the cache before it is accessed again at $\vec{i}$. As the statement is not true, assume $\vec{i}$ is a hit point. So, there must be another iteration point, say $\vec{i_1}$, in between $\vec{i}-\vec{r}$ and $\vec{i}$, where the same memory line is accessed but is not evicted before R accesses it again at $\vec{i}$. This implies that there exists a reuse vector $\vec{i}-\vec{i_1}$ which is clearly shorter than $\vec{r}$. This contradicts the assumption that there exists no reuse vector of R in I shorter than $\vec{r}$. Hence, the statement is true.

(2) Say, the iteration point $\vec{i}$ in I is neither a cold miss point nor a replacement miss point of R along $\vec{r}$. Since $\vec{i}$ is not a cold miss point along $\vec{r}$, $\vec{i}-\vec{r}$ exists within I and the same memory line is accessed by R at $\vec{i}-\vec{r}$ and $\vec{i}$. Hence, there is potential for realizing the reuse $\vec{r}$ at $\vec{i}$. But, the fact that $\vec{i}$ is also not a replacement miss point along $\vec{r}$ implies that the memory line accessed by R at $\vec{i}-\vec{r}$ is not evicted from the cache before it is accessed again by R at $\vec{i}$. Hence, the access of R at $\vec{i}$ enjoys a definite cache hit.

(3) Say, the iteration point $\vec{i}$ in I is a cold miss point along $\vec{r}$. Hence, either $\vec{i}-\vec{r}$ is outside the iteration space or the memory line accessed by R at $\vec{i}$ is different from the one accessed at $\vec{i}-\vec{r}$. This implies that the reuse $\vec{r}$ does not exist here. If there exist other reuse vectors in I (which has to be longer than $\vec{r}$ due to the assumption made in the theorem), the fate of the access of R at $\vec{i}$ will be determined by them and hence no conclusive remarks can be made just based on $\vec{r}$. However, if there exist no other longer reuse vectors, $\vec{r}$ is the only reuse vector present in I as it is already assumed to be the shortest one in I. Hence, $\vec{i}$, being a cold miss point along $\vec{r}$, becomes a definite cold miss point also. □

The algorithm in Figure 7 uses Theorem 2 to consider reuse vectors one at a time starting from the lexicographically shortest one. We can develop a different algorithm that considers all the reuse vectors simultaneously using Theorem 3 presented in Appendix C.

C. ALTERNATIVE TO THEOREM 2

Let us again consider the example iteration space in Fig 17(b). If at least one of the reuses are realized, the reference will find the data in cache at $\vec{i_1}$. Hence, the reference suffers a miss at $\vec{i_1}$ if and only if $\vec{i_1}$ is a miss along *all* the reuse vectors $\vec{r_1}$, $\vec{r_2}$, and $\vec{r_3}$. This is generalized and stated formally in the following theorem:

THEOREM 3. *The set of all miss instances of a reference is given by the intersection of all the miss-instance sets along the reuse vectors.*

PROOF. Let us assume that a reference R has m reuse vectors. We denote these reuse vectors as $\vec{r_1}, \vec{r_2}, \cdots, \vec{r_m}$ and the set of all miss instances along them (as evaluated by Theorem 1) as $M_1, M_2, \cdots, M_m$ respectively. Also let M denote the set of all miss instances or the iteration points at which the reference suffers a cache miss. Hence we are required to prove that $M = \cap_{k=1}^{m} M_k$.

Let $\vec{i} \in M$

$$\Rightarrow \textit{R suffers a cache miss at } \vec{i}$$

Now, we claim that $\vec{i} \in M_k$, *for every* $k \in [1, m]$. This can be proven by contradiction. Assume there exists a reuse vector $\vec{r}$ of the reference such that the iteration point $\vec{i}$ is not an element of M_r, the set of miss instances of $\vec{r}$. Now, if $\vec{i}$ is not an element of M_r, this implies that the reuse represented by the vector $\vec{r}$ is realized at $\vec{i}$. In other words, R does not suffer a cold or a replacement miss at $\vec{i}$ along $\vec{r}$. The fact that R does not suffer a cold miss at $\vec{i}$ along $\vec{r}$ guarantees that there exists a reuse of R at $\vec{i}$. Thus, there cannot be a definite cold miss at $\vec{i}$ irrespective of the presence of other reuse vectors. Also, as there is no replacement miss of R at $\vec{i}$ along $\vec{r}$, the memory line accessed by R at $\vec{i}$ is not evicted from the cache before it is reused at $\vec{i}$. Now, while finding cache misses along a particular reuse vector (by Theorem 1) we consider every possible interfering accesses. (We might, however, ignore the presence of other accesses to the same memory line as discussed in Lemma 4.) Hence, there is a definite hit at $\vec{i}$. But, this is contrary to our assumption that the reference suffers a cache miss at $\vec{i}$. Thus, it follows that

$$\begin{aligned} & \vec{i} \in M_k, \textit{ for every } k \in [1, m] \\ \Rightarrow\ & \vec{i} \in \cap_{k=1}^{m} M_k \\ \Rightarrow\ & M \subseteq \cap_{k=1}^{m} M_k \end{aligned}$$

Let $\vec{i} \in \cap_{k=1}^{m} M_k$

$$\begin{aligned} \Rightarrow\ & \vec{i} \in M_k, \textit{ for every } k \in [1, m] \\ \Rightarrow\ & \textit{There is no reuse which is realised at } \vec{i} \\ \Rightarrow\ & \textit{The reference suffers a miss at } \vec{i} \\ \Rightarrow\ & \vec{i} \in M \\ \Rightarrow\ & \cap_{k=1}^{m} M_k \subseteq M \end{aligned}$$

Hence, $\quad M = \cap_{k=1}^{m} M_k$ □

REFERENCES

ADLER, A. AND COURY, J. E. 1995. *The theory of numbers: A text and source book of problems.* Jones and Bartlett Publishers, Boston, MA.

ALLEN, R. AND KENNEDY, K. 1987. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Program. Lang. Syst. 9,* 4, 491–542.

BACON, D. F. ET AL. 1994. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proc. CASCON'94 conf.* (Nov.).

BAILEY, D. 1992. Unfavorable strides in cache memory systems. Tech. Rep. RNR-92-015, NASA Ames Research Center, CA.

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

CARR, S. AND KENNEDY, K. 1992. Compiler blockability of numerical algorithms. In *Proc. Supercomputing '92* (Nov.).

CARR, S. AND LEHOUCQ, R. B. 1995. A compiler-blockable algorithm for QR decomposition. In *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing* (Feb.).

CLAUSS, P. 1996. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *Proc. Int'l Conf. on Supercomputing* (May).

COLEMAN, S. AND MCKINLEY, K. S. 1995. Tile size selection using cache organization and data layout. In *Proc. SIGPLAN '95 Conf. on Programming Language Design and Implementation* (June).

EISENBEIS, C., JALBY, W., WINDHEISER, D., AND BODIN, F. 1990. A strategy for array management in local memory. In *Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing.*

FERRANTE, J., SARKAR, V., AND THRASH, W. 1991. On estimating and enhancing cache effectiveness (extended abstract). In *Proc. 4th Int'l Workshop on Languages and Compilers for Parallel Computing* (Aug.).

GALLIVAN, K., JALBY, W., AND GANNON, D. 1988. On the problem of optimizing data transfers for complex memory systems. In *Proc. Int'l Conf. on Supercomputing* (July).

GANNON, D., JALBY, W., AND K. GALLIVAN. 1988. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing 5,* 5 (Oct.), 587–616.

HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA.

HILL, M. D. 1987. Aspects of cache memory and instruction buffer performance. Ph.D. thesis, Computer Science Dept., University of California, Berkeley.

HILL, M. D. AND SMITH, A. J. 1989. Evaluating associativity in CPU caches. *IEEE Trans. Comput. 38,* 12 (Dec.), 1612–1630.

IRIGOIN, F. AND TRIOLET, R. 1988. Supernode partitioning. In *Proc. 15th Annual ACM Symposium on the Principles of Programming Languages* (Jan.).

LAM, M., ROTHBERG, E. E., AND WOLF, M. E. 1991. The cache performance of blocked algorithms. In *Proc. 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (Apr.).

LEBECK, A. R. AND WOOD, D. A. 1994. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer* (Oct), 15–26.

LI, W. AND PINGALI, K. 1992. Access normalization: Loop restructuring for NUMA compilers. In *Proc. 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (Oct).

MARTONOSI, M., GUPTA, A., AND ANDERSON, T. 1992. MemSpy: Analyzing memory system bottlenecks in programs. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems* (June). 1–12.

MCKINLEY, K. S., CARR, S., AND TSENG, C. W. 1996. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst. 18,* 4 (July), 424–453.

MCKINLEY, K. S. AND TEMAM, O. 1996. A quantitative analysis of loop nest locality. In *Proc. 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (Oct.).

MOWRY, T. C., LAM, M. S., AND GUPTA, A. 1992. Design and evaluation of a compiler algorithm for prefetching. In *Proc. 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (Oct.).

NAVARRO, J. J., JUAN, T., AND LANG, T. 1994. Mob forms: A class of multilevel block algorithms for dense linear algebra operations. In *Proc. Int'l Conf. on Supercomputing* (June).

PORTERFIELD, A. K. 1989. Software methods for improvement of cache performance on supercomputer applications. Ph.D. thesis, Rice University.

PUGH, W. 1992. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Commun. ACM 35,* 8 (Aug.), 102–114.

PUGH, W. 1994. Counting solutions to Presburger formulas: How and Why. In *Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation* (June). 121–134.

RIVERA, G. AND TSENG, C. W. 1998. Data transformations for eliminating conflict misses. In *Proc. ACM SIGPLAN'98 Conference on Programming Language Design and Implementation* (July).

SUGUMAR, R. A. AND ABRAHAM, S. G. 1993. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proc. ACM SIGMETRICS Conference on Measurement & Modeling of Computer Systems* (May).

Temam, O., Fricker, C., and Jalby, W. 1994. Cache interference phenomena. In *Proc. ACM SIGMETRICS Conf. on Measurement & Modeling of Computer Systems* (May).

Temam, O., Granston, E., and Jalby, W. 1993. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proc. Supercomputing'93* (Nov.).

Torrellas, J., Lam, M. S., and Hennessey, J. L. 1990. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *Proc. Int'l Conf. on Parallel Processing* (Aug.).

Wilson, R. P. et al. 1994. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices 29,* 12 (Dec.).

Wolf, M. E. and Lam, M. S. 1991. A data locality optimization algorithm. In *Proc. SIGPLAN '91 Conf. on Programming Language Design and Implementation* (June).

Wolfe, M. J. 1989. More iteration space tiling. In *Proc. Supercomputing '89* (Nov).