Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity

Somnath Ghosh Margaret Martonosi Sharad Malik

Department of Electrical Engineering
Princeton University
Princeton, NJ 08544-5263
{sghosh,martonosi,sharad}@ee.princeton.edu

Abstract

Analyzing and optimizing program memory performance is a pressing problem in high-performance computer architectures. Currently, software solutions addressing the processor-memory performance gap include compiler- or programmer-applied optimizations like data structure padding, matrix blocking, and other program transformations. Compiler optimization can be effective, but the lack of *precise* analysis and optimization frameworks makes it impossible to confidently make optimal, rather than heuristic-based, program transformations. Imprecision is most problematic in situations where hard-to-predict cache conflicts foil heuristic approaches. Furthermore, the lack of a *general* framework for compiler memory performance analysis makes it impossible to understand the combined effects of several program transformations.

The Cache Miss Equation (CME) framework discussed in this paper addresses these issues. We express memory reference and cache conflict behavior in terms of sets of equations. 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. Unlike our prior work, this framework applies to caches of arbitrary associativity. This paper also demonstrates the utility of CMEs by presenting precise algorithms for intra-variable padding, inter-variable padding, and selecting tile sizes. Our experiences with CMEs implemented in the SUIF system show that they are a unifying mathematical framework offering the generality and precision imperative for compiler optimizations on current high-performance architectures.

1 Introduction

As the disparity between processor cycle times and main memory access times grows, data cache performance becomes increasingly important. Although caches generally work well, some programs have access patterns that fail to use the cache effectively. Ideally, automatic compiler transformations should improve the memory behavior of such code, thereby reducing the programmer's need to hand-tune. While compiler transformations are often effective, the analysis and optimization techniques they employ are sometimes insufficient for memory optimizations on particular programs or machines. These shortcomings are often due to a lack of precision or generality. In response, this paper describes Cache Miss Equations (CMEs), a precise mathematical framework for guiding a range of compiler optimizations.

There are two main approaches to automatically improving the data locality of loop-oriented programs—*loop nest restructuring* and *data layout optimizations*. Restructuring loop optimizations (e.g., permutation, tiling, and fusion) are mechanisms widely used to reorder the access pattern in a loop nest for better temporal and spatial locality [8, 12, 15, 23, 24]. The key issues here are determining appropriate analyses and policies for determining when to apply these optimizations. In the past, such analyses have primarily considered capacity misses, but loops can also suffer heavily from conflict misses, particularly in caches with low associativity [11, 12, 16, 21]. Moreover, conflict misses are highly sensitive to slight variations in problem size and base addresses [3, 12]. As a result, previous compiler techniques, using simpler cost models to guide loop transformations, have disappointing results when there are unaccounted-for conflict misses. Researchers have also considered transformations on the underlying data layout. They have developed specialized algorithms for selecting tile sizes [7, 12] and for specific data layout optimizations [3, 20] to reduce conflict misses.

While compiler transformations have been effective in optimizing some programs, they are frequently based on limited or heuristic analyses whose imprecision leads to poor results for some programs and cache organizations. Furthermore, these transformations are often based on individual isolated insights, with little common base, perhaps a consequence of the lack of a unified analytical model for program cache behavior. It is therefore difficult to reason about the combined effect of several transformations applied to the same code.

To address these problems, we have introduced *Cache Miss Equations* (CMEs), a mathematical framework that precisely represents the cache misses of a loop nest. CMEs are unique in unifying both the loop structure and the data layout within a simple, equations-based analytical model representing the cache misses. We can count cache misses in a code segment by analyzing the number of solutions present for a system of CMEs, where each solution corresponds to a potential cache miss. CMEs are a system of linear Diophantine equations. The precision of this framework allows

ASPLOS VIII 10/98 CA, USA

us to find optimal solutions to padding or blocking optimizations by employing mathematical analysis to determine when the equations will have zero solutions. The generality of the framework allows us to concurrently reason about the impact of multiple optimizations on the solution count; this gives us a mechanism for analyzing several optimizations applied at once. We have implemented our algorithm to automatically generate CMEs within SUIF [22]. We have also tested our system by automatically generating the equations for many numerical loop nests from the SPECfp benchmarks and subsequently using them for optimizations.

This work makes several significant contributions. Our prior work described CMEs and some initial manipulations thereof [9], but that work was applicable only to direct-mapped caches. The new CME framework presented in this paper applies to caches of arbitrary associativity. Furthermore, our prior work employed a solution method based on difficult aggregate set operations on the complete set of a reference's reuse vectors. The work we describe in this paper "re-invents" CMEs in a way that is both more broadly useful and more conveniently manipulated. We have implemented an entirely new algorithm for finding cache misses from the CMEs; this algorithm enables one to trade off precision versus analysis complexity for practical loop nests. The final contribution of this paper is to give new compiler optimization algorithms that are concrete examples of the sophisticated code optimization methodologies that CMEs enable. While space limits preclude a comprehensive demonstration of CME's broad utility, these examples and experimental data demonstrate how the precision and generality of our framework make CME-based optimizations more effective at reducing cache misses than other non-CME-based optimizations previously described by the compiler community.

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 includes examples of using CMEs on a range of code optimizations. Finally, Sections 6, 7, and 8 present related work, future work, and conclusions respectively.

2 CMEs: 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 [2]. 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. Extending this work to handle conditionals, by focusing on the frequently-taken program paths is part of our future research. Our previous work evaluated these restrictions for the SPECfp benchmarks and found that the majority of loops satisfied them [9]. In this paper, we consider each loop nest separately; inter-nest analysis is part of our ongoing research. 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.

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 optimizing value needs to be determined for some cache optimization is kept as a parameter in the CMEs. Compiler optimizations will then try to use these CMEs based on available information. However, in order to find the exact number of cache misses using CMEs for a precise evaluation, we will 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 [18], locality optimizations [23], or prefetching algorithms [17]. We build on these approaches and adopt more precise mechanisms for using them.

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

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

Figure 1: Matrix multiply loop nest.

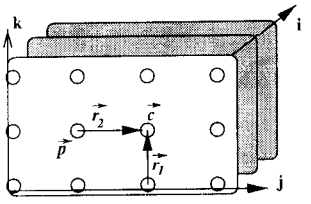

Figure 2: Iteration space of the matrix multiply loop nest. The iteration points are shown by the hollow dots. $\vec{r_1} = (0, 1, 0)$ and $\vec{r_2} = (0, 0, 1)$ are two reuse vectors of $Z(j, i)$ shown at the iteration point $\vec{c} = (1, 2, 3)$. $\vec{r_1}$ is a self-temporal reuse vector and $\vec{r_2}$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

of the n-dimensional iteration space Z^n, bounded by the loop bounds [6, 23]. 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 represented by the first dimension. Figure 2 shows the iteration space of the matrix-multiply loop nest in Figure 1. $\vec{c}$ 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 2, $\vec{c} \succ \vec{p}$.

Memory terminology: We refer to a static read or write in the program as a *reference*, while a particular execution of that read or write at runtime is a *memory access.* A *memory line* refers to a cache-line-sized block in memory, while a *cache line* refers to the actual cache block a memory line maps 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 $C_s = N_s \times k \times L_s$. With all elements in units of data element size, the cache set accessed by a reference R_A at the iteration point $\vec{i}$ is given by the following expression:

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

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 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.

Reuse Vector: Reuse vectors provide a mechanism for summarizing repeated memory access patterns in loop-oriented code [23]. 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 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 2.

If the reference 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. If a reuse results in a cache hit we say that the *reuse is realized* and the reference that enjoys the cache hit has *locality.*[1] 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 [23] 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 2, 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.

Miss Along a Reuse Vector: Consider a miss of a reference R at an iteration point $\vec{i}$. We define the **miss 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, that CME represents all the *misses* of R *along the reuse vector* $\vec{r}$. Section 4 shows how all the reuse vectors interact to decide the cache misses for a reference.

3 Generating the Cache Miss Equations

Our approach generates two types of CMEs: *cold miss equations* (or *cold CMEs*) and *replacement miss equations* (or *replacement CMEs*). Solutions to the cold miss equations represent potential *cold* or *compulsory misses*—misses that occur on the first reference to a memory line. Solutions to the replacement miss equations represent all other misses including both *capacity* and *conflict misses.*

Figure 3 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.

[1] We use *reuse* and *locality* as defined in Wolf and Lam [23]. *Reuse* is considered to be an intrinsic property of the references that occurs whenever a reference accesses a memory line that was already accessed before and *locality* is used to indicate that the reuse has resulted in a cache hit. McKinley and Temam [16], however, follows a different interpretation where the meanings of reuse and locality are interchanged with respect to our interpretation.

Algorithm Generate
Generate the CMEs for a loop nest
Input:
Information about the loop nest, array references in the nest, and the sequence in which those references appear in the generated code.
Output: A set of CMEs for each reference.
Algorithm:
Generate the reuse vectors
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).

Figure 3: Algorithm to generate all the CMEs.

3.1 Forming Cold Miss Equations

Cold CMEs mathematically summarize the situations when a memory line is brought into the cache for the first time. Each loop nest is treated in isolation, so we assume none of the data accessed in a loop nest is already present in the cache before it starts execution. The simplest type of cold CME is already fairly familiar. Namely, a relationship like $(i \bmod 4) = 0$ 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 cold CMEs that capture all the cold misses along each reuse vector. We use reuse vectors to help us determine when a cold access occurs; cold memory accesses are encountered at the iteration points that contain (1) either the first access along the direction of the vector or (2) accesses that have just crossed a memory line boundary along the direction of the vector. As a result, the cold misses are dependent on the reuse vectors, iteration space, and the line size. Since cold misses do not change with cache associativity, the methods to generate the cold CMEs remain as described in [9].

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:

$$Memory\_Address\_of\_R_A = Memory\_Address\_of_{R_B} + n \times Cache\_Size + Line\_Size\_Range \quad (2)$$

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 2 is a linear Diophantine equation.

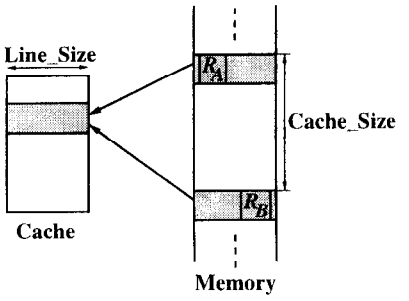

Figure 4: An example of two memory addresses that map to the same cache line. The addresses are marked by the references that access them, namely R_A and R_B.

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 that map to the same cache set. Each of these conflicts satisfy an equation similar to Equation 2. 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.[2] 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 last iteration point where R_A accessed the same memory line is $\vec{p} = \vec{i} - \vec{r}$. Interferences between R_A and R_B can occur at all iteration points lying between $\vec{p}$ and $\vec{i}$. Depending on the relative access order of R_A and R_B, either $\vec{p}$ or $\vec{i}$ will also be included in the set. (Our implementation extracts access order information automatically from the code generation phase.) An example of these potentially-interfering points is shown in Figure 5.

[2]Note the contrast to other research that may use self-interference more broadly to refer to any cache interferences of a data structure with itself.

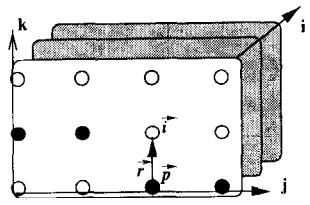

Figure 5: An example of potentially-interfering points in a 3D loop nest. The current iteration point is $\vec{i} = (1, 2, 4)$, the reuse vector is $\vec{r} = (0, 1, 0)$, and $\vec{p} = (1, 1, 4)$. The potentially-interfering points are shown by the filled dots.

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 $\vec{r}$:

$$Cache\_Set_{R_A}(\vec{i}) = Cache\_Set_{R_B}(\vec{j}) \quad (3)$$

Substituting in the expressions from Equation 1, we can simplify the resulting equation to the linear Diophantine equation shown in Equation 4.

$$Mem_{R_A}(\vec{i}) = Mem_{R_B}(\vec{j}) + nC_s/k + b \quad (4)$$

where C_s is the cache size and n is any non-zero integer. The variable b can take on values in the range $-L_{off} \leq b \leq L_s - 1 - L_{off}$ where $L_{off} = Mem_{R_B}(\vec{j})\ mod\ L_s$. 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. Since the loop indices are bounded, the equality holds for a bounded region.

Every solution of Equation 4 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 cache sizes. So, if n is 0 the memory lines are identical and there is no conflict at all. That is why we disallow all solutions with $n = 0$. In a k-way set associative cache, there are k cache lines in every cache set, so k distinct contentions are needed before a cache miss will occur along the reuse vector $\vec{r}$.

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 5 shows the replacement CME 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.)

$$\begin{aligned} &Cache\_Set\_Z(j, i) = Cache\_Set\_X(k', i') \\ &\quad \text{where } (i', k', j') \in ((i, k, j-1), (i, k, j))] \\ &\Rightarrow \lfloor (4192 + 32i + j - 1)/4 \rfloor \mod 128 \\ &\quad = \lfloor (2136 + 32i + k - 1)/4 \rfloor \mod 128 \\ &\Rightarrow 4192 + 32i + j = 2136 + 32i + k + 512n + b \end{aligned} \quad (5)$$

where $n > 0$, $(i, j) \in [(0, 0), (31, 31)]$, $b \in [-3, 3]$. 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.

4 Finding Cache Misses from CMEs

This section describes the algorithm for finding all cache misses in a loop nest by composing the effects of multiple CMEs. This algorithm 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 we describe 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 [9], as well as replacement CMEs representing the self- and cross-interferences of this reference with itself and 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 6 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.

In our previous work, determining cache miss points from solution points relied on computing unions and intersections of solution sets, while considering all the reference's reuse vectors at once [9]. Our current methodology represents a significant departure from this method; we instead consider reuse vectors one-by-one in lexicographic order. While both methods are of the same computational complexity, our current method can be computed more quickly for practical loop nests commonly occurring in real programs. In addition, this method applies more directly to caches of arbitrary associativity, while our earlier approach was only for direct-mapped caches.

Here, we will provide an intuitive explanation of the algorithm shown in Figure 6 with the help of an illustrative example. Formal proofs can be found in a more mathematical report [10]. We will consider the iteration space shown in Figure 7 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, 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

```
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
{
1. for each reference
     /* Say the reference is X */
2.   Sort the reuse vectors lexicographically from
           the shortest to the longest one;
3.   M_X = φ (null set);
4.   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 */
5.   for each reuse vector of the reference X
     /* Say the reuse vector is r⃗ */
6.     if (|C| ≤ ε) 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 */
7.     C' = union of the solutions of cold CMEs of r⃗;
8.     R' = union of the solutions of replacement CMEs of r⃗;
9.     R'' = φ;
10.    for each (i⃗, j⃗, n) ∈ R'
11.      R'' = R'' ∪ {(i⃗, n)};
         /* R'' stores the distinct (i⃗, n) tuples */
12.    I = φ;
13.    for each (i⃗, n) ∈ R''
14.      I = I ∪ {i⃗}; |i⃗| += 1;
         /* |i⃗| keeps track of #occurrences of i⃗ */
         /* So, |i⃗| counts the distinct cache set contentions */
         /* All |i⃗|'s are initialized to 0 before this loop */
15.    I = I ∩ C; C = C ∩ C'; /* Search inside C only */
16.    I = I − C;
       /* eliminate cold CME solution points from I */
17.    for each i⃗ ∈ I
18.      if |i⃗| ≥ k /* k = associativity of the cache */
           /* i⃗ is a replacement miss point along r⃗ */
19.        M_X = M_X ∪ {i⃗};
     /* At this point C = cold misses of the reference */
     /* and M_X = replacement misses of the reference */
20.  M_X = M_X ∪ C;
}
```

Figure 6: Algorithm to find the cache miss points of a loop nest from its CME solutions.

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 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 6) immediately after it is found. In Figure 6, 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.

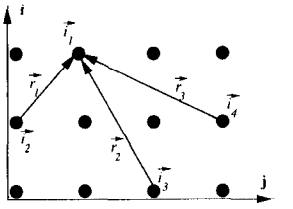

Figure 7: Illustration of the algorithm to find cache misses for a 2D loop nest.

Figure 8 depicts the progress of the algorithm for the load reference of $Z(j,i)$ in a 256×256 matrix multiply loop nest (Figure 1). 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 8 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.

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 6.) 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 reference (the "perpetrator") occurs. The third element of the triple, n, denotes how many "cache wraparounds" there

	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

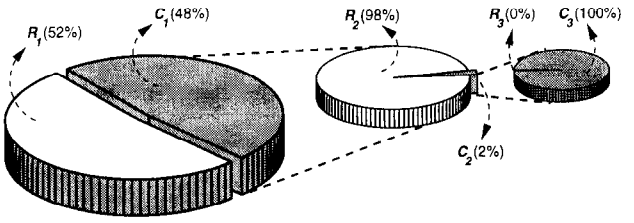

Figure 8: Using the CME-based algorithm from Figure 4 to find cache misses for the load of $Z(j,i)$ in the matrix multiply loop nest (Figure 1). The diagram and the table shows the progress of the algorithm as reuse vectors are considered one by one, each time zeroing in on the previously indeterminate 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.

are between the memory addresses of the two potentially-conflicting references. 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 6).

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 6). 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 [5, 18]. Taking the unions and intersections shown in Figure 6 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 1 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 for Automated and Interactive Analysis

CMEs form a mathematical underpinning for analyzing many different cache optimizations. To highlight their generality, this section describes four distinct algorithm styles for cache optimizations using CMEs. These cover a range of automatic and semi-automatic techniques. Most importantly, none of these techniques explicitly require us to solve the CMEs; instead, mathematical properties of CMEs directly facilitate the optimizations. As shown in Figure 9, we broadly classify methods into the following categories:

- Automated CME Analysis
  - Exploiting Special Cases
  - Using a Solution Counting Engine
  - Using Parametric Solutions
- Interactive CME Analysis

Space limits preclude detailed examples for each of the four usages. Instead, we focus on the first, and give brief sketches of example use for the rest.

5.1 Automated CME Analysis

5.1.1 Exploiting Special Cases

The general strategy for an algorithm to exploit mathematical special cases is to form the CMEs and study them to determine a set of well-defined conditions that eliminate or reduce their solution count, i.e. minimize cache misses. This analysis is then codified into an algorithm that automatically finds optimizing parameter values to reduce or eliminate CME solutions.

Loop Nest	#Arrays.	Max. #refs to an array	#Data accesses	#Data cache misses		%Error	#Refs.	Max. #RVs used per ref.	Distribution of RVs used	
				DineroIII	CME				SUIF-RV	Ext-RV
mmult	3	2	67108864	7042336	7042336	0.0	4	3	7	3
gauss	1	5	16744320	1998466	2019682	1.0	5	2	4	-
sor	1	6	387096	8192	8192	0.0	6	4	11	-
adi	3	3	587520	391680	391680	0.0	9	1	9	-
trans	1	4	262144	73456	73732	0.4	4	2	6	-
alv	2	2	183150	14090	14090	0.0	5	1	5	-
tom	4	2	387096	258064	258064	0.0	6	1	6	-

Table 1: #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.)

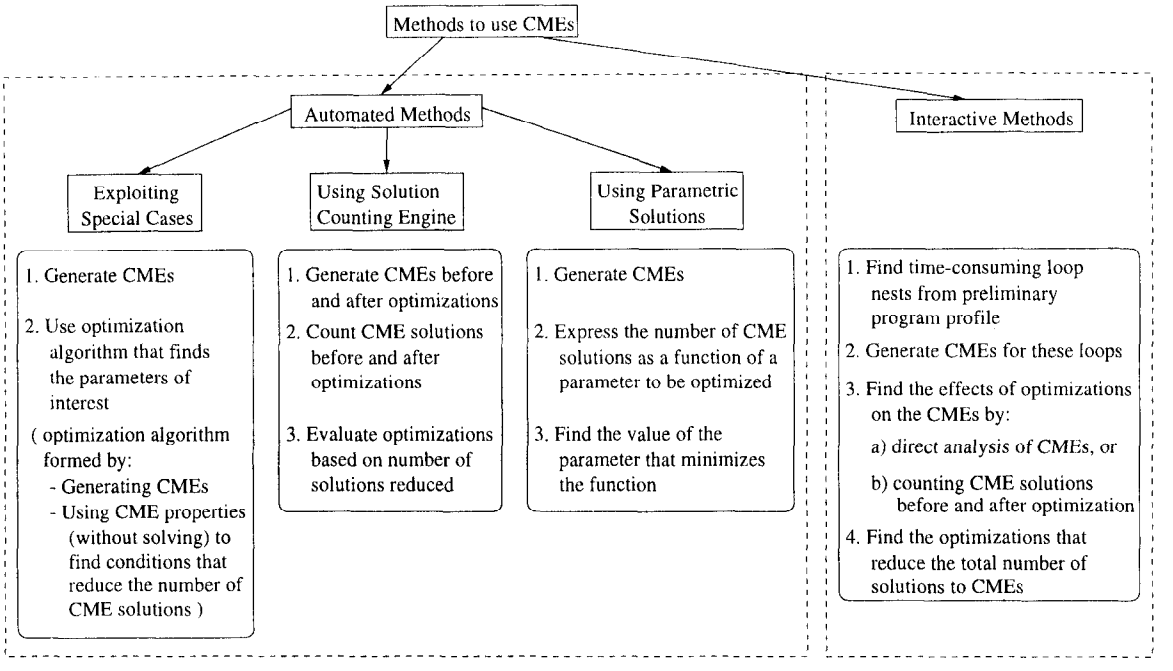

Figure 9: Overview of the methods using CMEs for optimizations.

Example: Padding In the first of the automated methodologies we give an example of a padding algorithm that uses mathematical properties available through CMEs. Our algorithm finds 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.

In the padding example, the parameters of interest 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 references that access the same array are of the following type (called Type 1 in Fig. 10):

$$C(\delta f + c - d) - nC_s = b - (\delta f_0 + c' - d') \quad (6)$$

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 [1, 5], 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):

$$(B_X - B_Y) + C(\delta f + c - d) - nC_s = b - (\delta f_0 + c' - d') \quad (7)$$

Again from number theory, Equation 7 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. $\gcd(C, C_s) > |B_X - B_Y|$ 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:

From Condition 1 : $x > \lg(\max|b - (\delta f_0 + c' - d')|)$

From Condition 2 : $x < \lg(C_s/(\max|\delta f + c - d|))$

From Condition 3 : $x, y > \lg(\max|b - (\delta f_0 + c' - d')|)$

From Condition 4 : $x > y$

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. Similarly, once y is known, it can choose any value of t_2 such that $|B_X - B_Y|$ is at least equal to the size of the arrays lying between B_X and B_Y.

Based on these constraints, we have developed the algorithm sketched out as pseudo-code in Figure 10. 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 exists 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 trade-off precision and compute time by choosing how many reuse vectors to consider. We have implemented the described algorithm considering only the nearest reuse vector for every reference.[3] The algorithm is quite fast—quadratic on the number of references, which is a small number in all practical loop nests.

Table 2 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, namely the *trans* loop nest. There exists no padding solution for our algorithm to reduce the replacement misses in the *trans* loop nest. We believe no other padding algorithm can find effective solutions for this loop nest. In fact, 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 [20]. For example, their methods cannot decrease any conflict misses for the *mmult* loop (Figure 1), because they do not address inter-array padding for the Y(j,k) and Z(j,i) references that are not uniformly generated. Rivera and Tseng's 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 11); 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.

[3] For even more precise results, one could increase the number of reuse vectors considered.

```
Algorithm Find_ColumnSize_and_BaseAddresses
Input: CMEs of the loop nest
Output: C, and B_X for every reference X
{
For each reference
   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 arrays are X and Y */
            update lower_bound(x) with Lb
            update lower_bound(y(XY)) with Lb
            add constraint (x > y(XY))
Get_padding from above ranges and constraints
}
```

Figure 10: Algorithm for padding arrays and setting base addresses to reduce cache interferences.

Figure 12 shows the sensitivity of cache misses in the *alv* loop to different choices of row 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.

Example: Selecting Tile Size to Eliminate both Self and Cross Interferences In this example we use CMEs to find effective tile sizes given a tiled loop nest. Cache conflicts are highly sensitive to the problem size and the tile size [12], motivating researchers to find tile sizes based on program and cache parameters [7, 12]. Our approach here is novel in that we integrate tile size selection and padding in order to reduce both self and cross interferences. We briefly describe the process for a tiled matrix multiply loop nest, determining a tile 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 also its cross-interference equation with $Z(j,i)$. For the tiled code they are:

$$C\delta k - nC_s = b - \delta j, \quad \text{where} \quad \delta k < T_k, \quad \delta j < T_j \quad (8)$$

$$(B_Y - B_Z) + C(k - i') - nC_s = b - \delta j, \quad \text{where} \quad k \in [0, N-1], \quad i' \in [0, N-1], \quad \delta j < B_j \quad (9)$$

The forms of these equations are similar to those used for padding. The variables to be optimized here are T_k, T_j, B_Y, and B_Z. There are a lot of ways one can proceed here. We have developed an algorithm where we first find T_k, T_j from Equation 8, and then optimize B_Y, B_Z from Equation 9 by an algorithm similar to Figure 10. The tile size selection algorithm conceptually finds all combinations of (T_k, T_j)

Loop Nest	#Data accesses	#Data cache misses				%Reduction in cache misses	
		Original		Optimized			
		Replacement	Total	Replacement	Total	Replacement	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

Table 2: Impact of our padding algorithm: Data cache misses in the original and optimized code. Both replacement miss and total miss counts are shown.

```
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 ;
```

Figure 11: Loop nest from *alvinn* benchmark.

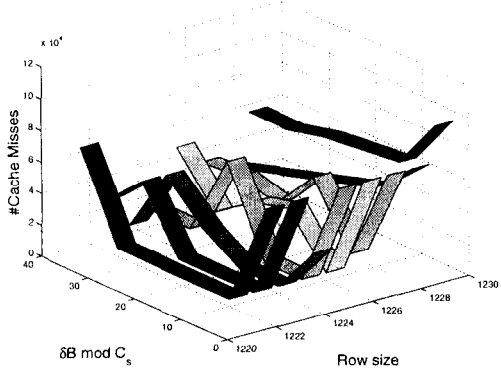

Figure 12: Surface plot of #cache misses for different row sizes and base address positioning of the two arrays accessed in the *alv* loop of Figure 11. (δB is the difference in base addresses of the two arrays.)

that ensure no solution to Equation 8 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 Equation 8. As our algorithm combines padding along with selecting tile sizes, it would be interesting to compare this algorithm with the tile size selection algorithm presented by Coleman and McKinley [7].

5.1.2 Using Solution Counting Engines

As with the previous subsection, the methodology described here does not require *generating* CME solutions. Rather, the example presented here relies on being able to count the number of CME solutions, which is potentially much faster. Counting cache misses for a set of CMEs is equivalent to counting the number of lattice points in some projection of a union of polytopes. This problem has received attention recently in the context of parallel compilers, and several researchers have presented solutions for this that work reasonably well in practice [6, 19]. The method presented in this section uses these lattice point counting engines to directly compute the number of solutions of CMEs and uses this to drive optimizations. The general strategy is sketched out in Figure 9. We illustrate its use here in determining when to apply loop fusion.

Example: Loop fusion Consider the example loop nest shown in Figure 13(a). The example is similar to a loop nest found in the ADI kernel used by McKinley and Temam [16]. We consider a 4-byte array element size and an 8KB direct-mapped cache with 32-byte lines. The base addresses of the arrays A, B, X are $0x10000110, 0x10004130, 0x10008150$ respectively. Figure 13(b) shows the transformed code after loop fusion. We use our automated CME generator to generate the CMEs before and after applying loop fusion to this loop nest. Then, we count the cache misses in both cases by counting the number of solutions to the CMEs using a solution counting engine. Before the transformation, there were roughly 21K cache misses. After loop fusion, the CMEs indicate a drop to roughly 15K cache misses. Thus, CMEs can be effectively used with solution point counters to determine when to apply particular optimizations like loop fusion. The precision of CMEs allows us to consider a particular cache organization when making this decision.

5.1.3 Using Parametric Solutions

In this third category of automated CME optimization, compiler-writers determine the functional relationship between the number of CME solutions (i.e., cache misses) and a set of input parameters related to the desired optimization. Thereafter, they use function optimization techniques to find the parameter values that optimizes the CME solution function. This technique again uses a lattice point counting engine, except that instead of finding a numeric value for the number of misses, it determines the number of misses as a function of some parameter. (Such parameterized optimizations are possible with lattice point counting techniques presented in the literature [6, 19].)

For example, we have used this approach as an alternative technique for implementing the padding optimization. In this case, the value the compiler controls (such as the inter-variable spacing) appears as a parameter in the CMEs. To determine the number of solutions as a function of this parameter, we generate *Ehrhart Pseudo-Polynomials* (*EP*s) [6] for the above CMEs. Finally, one can analyze the specific *EP*s generated to find the parameter value that minimizes the *EP*s.

For a specific optimization, this methodology is completely automatic. When optimizations can be expressed

(a) Input code (from ADI Kernel)

```
DO i = 2, 64
   DO k = 1, 64
      X(i, k) -= X(i-1, k) * A(i, k) / B(i-1, k)
DO i = 2, 64
   DO k = 1, 64
      B(i, k) -= A(i, k) * A(i, k) / B(i-1, k)
```

(b) Transformed code after loop fusion

```
DO i = 2, 64
   DO k = 1, 64
      X(i, k) -= X(i-1, k) * A(i, k) / B(i-1, k)
      B(i, k) -= A(i, k) * A(i, k) / B(i-1, k)
```

Figure 13: ADI loop used for evaluating loop fusion by CME solution counting.

parametrically, determining the optimal parameter value *without enumerating all possible values* can be computationally advantageous compared to brute-force solution point counting. While solution counting provides a precise count, it requires an iterative search-and-evaluate process through possible parameter values. Parametric approaches, when they apply, can zero in on the functionally-optimal choice.

5.2 Interactive CME Analysis

Automated optimization techniques are preferable, but often programmers supplement them with hand-tuned memory optimizations as well. To the astute user, CMEs may also be useful for interactively exploring the possible impact of different cache optimizations. As summarized in Figure 9, interactive methods share characteristics with the automated methodologies, but interactive analysis allows more complex decision-making that is hard to automate. CME manipulations can often be helpful for reasoning about the impact of cache optimizations which would otherwise be possible only through elaborate cache simulations [13, 14]. In addition to the direct benefit of helping programmers in hand-tuning code, interactive approaches are also the first step in discovering optimizations, automating them, and including them in compilers. We have used interactive analysis extensively to develop our optimization algorithms, and also to study the combined effects of various cache optimizations on loop nests taken from the SPECfp benchmarks.

5.3 Computational Requirements

Here we discuss the computational requirements of the different usage scenarios. 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 $O(n_{ref}^2 \times (max(n, d_{max}))^3)$.

Once the reuse vectors are calculated, the time taken to generate all the CM equations of the loop nest is given by $O(n \times d_{max} \times n_{eqn})$, where n_{eqn} = #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 [22] 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. The "special case" approach of Section 5.1.1 simply computes GCDs, and is a linear algorithm in the number of loop indices.

Section 5.1.2 discussed methods that require counting the number of CME solutions. CMEs and their related inequalities are similar in form to other equation systems previously discussed for dependence analysis in parallelizing compilers. Methods for counting solutions to such systems of equations are, worst-case, exponential-time algorithms. For practical loop nests, however, solution counting methods have been given in [6, 19].

When using parametric methods to manipulate CMEs, as in Section 5.1.3, the computation time includes first generating a parametric function, and then second, determining the parameter value that optimizes the function. As with solution counting, the former task is exponential in the worst-case, but remains computationally tractable for practical loop nests [6]. The latter task, function optimization, can be simply passed to a mathematical software package.

Interactive methods clearly depend on how the user manipulates the equations, but are likely to include one or more of the steps described above.

6 Related Work

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 [15, 23, 24]. For example, most of these explore the popular technique of loop tiling to reduce capacity misses. There are also several case studies that report the severe adverse effects of cache interference or conflicts on cache performance [12, 16, 21]. Due to the difficulty in predicting and estimating cache conflicts, however, there are relatively few studies on analyzing and reducing interference misses.

Methods for predicting and estimating cache misses in the presence of cache interferences have been considered by Ferrante et al. [8] and Temam et al. [21]. Due to several approximations in their cost model, these methods are not as precise as ours. More importantly, these methods only estimate the *number* of cache misses, while our approaches can give insights as to their cause.

Some of the optimizations described in this paper have been addressed in isolation in previous work. Algorithms to select efficient padding amounts have been proposed [3, 4, 20]. There are also some papers on choosing problem-size-dependent tile sizes that eliminate self-interference and capacity misses in a tiled loop nest [7, 12]. We have shown how our general framework of CMEs can also be used for guiding these optimizations, sometimes more precisely. Moreover, CMEs provide a mechanism for analyzing several of these optimizations applied at once.

Finally, there has been some work on automatic analysis and counting the number of solutions to a set of linear equalities and inequalities [6, 18, 19]. This complementary work would help to automate the analysis of the CMEs.

7 Future Work

In order to make our analysis framework more general, it needs to handle the effects of multiple loop nests in the program. This needs efficient methods to calculate reuse vectors across loop nests. Fortunately, most inter-nest misses occur between adjacent nests [16] and so it may be enough to find reuse vectors only between adjacent nests for most practical purposes. In order to automate our parametric analysis for cache optimizations, we hope to extend the methods presented by Pugh [18] and Clauss [6]. Finally, we would like to use CMEs for developing an automatic algorithmic framework for diagnosing poor cache behavior and selecting appropriate transformations.

8 Conclusions

The widening processor-memory performance gap makes a program's memory referencing behavior increasingly important. Compiler optimizations for cache accesses are frequently effective, but often a lack of precision or generality can cause them to provide disappointing performance for some programs or cache organizations. Our work has developed *Cache Miss Equations*, a detailed, analytic, compile-time framework for representing the caching behavior of loop-oriented programs. CMEs unify the effects of both loop structure and data layout, and thus can be used as the foundation for a range of control and data optimizations. CMEs are a general framework with the mathematical precision needed to accurately predict cache behavior at compile-time.

This paper demonstrates how to generate a program's CMEs for caches of arbitrary associativity and how to determine cache misses from the equations formed. We have implemented this method within the SUIF compiler framework. We also describe a variety of automatic and interactive program transformations based on different CME usages. These examples serve two purposes. First, in some cases, the algorithms are improvements on current compiler optimizations for padding and blocking. Second, these algorithms serve as examples of how CMEs facilitate precise optimization techniques. Furthermore, their generality provides a framework for reasoning about the combined effects of optimizations applied in concert. In summary, CMEs represent a general and precise foundation that will serve as an enabling technology for effective cache optimizations in the future.

References

[1] A. Adler and J. E. Coury. *The theory of numbers: A text and source book of problems*. Jones and Bartlett Publishers, Boston, MA, 1995.

[2] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Prog. Lang. Syst.*, 9(4):491–542, 1987.

[3] D. F. Bacon et al. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proc. CASCON'94 conf.*, Nov. 1994.

[4] D. Bailey. Unfavorable strides in cache memory systems. Technical Report RNR-92-015, NASA Ames Research Center, CA, 1992.

[5] U. Banerjee. *Loop transformations for restructuring compilers*. Kluwer Academic Publishers, Norwell, MA, 1993.

[6] P. Clauss. 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 1996.

[7] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proc. SIGPLAN '95 Conf. on Programming Language Design and Implementation*, June 1995.

[8] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness (extended abstract). In *Proc. 4th Int'l Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.

[9] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proc. Int'l Conf. on Supercomputing*, July 1997.

[10] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: An analytical representation of cache misses. Technical report, Dept. of Electrical Engineering, Princeton University, 1998.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA., 1996.

[12] M. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance of blocked algorithms. In *Proc. 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Apr. 1991.

[13] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, pages 15–26, Oct 1994.

[14] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: Analyzing memory system bottlenecks in programs. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 1–12, June 1992.

[15] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Trans. Prog. Lang. Syst.*, 18(4):424–453, 1996.

[16] K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proc. 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[17] T. C. Mowry, M. S. Lam, and A. Gupta. 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. 1992.

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

[19] W. Pugh. Counting solutions to Presburger formulas: How and Why. In *Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 121–134, June 1994.

[20] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proc. ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, July 1998.

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

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

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

[24] M. J. Wolfe. More iteration space tiling. In *Proc. Supercomputing '89*, Nov 1989.