Cache Miss Equations: An Analytical Representation of Cache Misses

Somnath Ghosh Margaret Martonosi Sharad Malik

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

Abstract

With the widening performance gap between processors and main memory, efficient memory accessing behavior is necessary for good program performance. Both hand-tuning and compiler optimization techniques are often used to transform codes to improve memory performance. 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 that give a detailed representation of the cache misses in loop-oriented scientific code. Implemented within the SUIF compiler framework, our approach extends on traditional compiler reuse analysis to generate linear Diophantine equations that summarize each loop's memory behavior. Mathematical techniques for manipulating Diophantine equations allow us to compute the number of possible solutions, where each solution corresponds to a potential cache miss. These equations provide a general framework to guide code optimizations for improving cache performance. 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 is more precise than traditional memory behavior heuristics, and is also potentially faster than simulation.

1 *Introduction*

Over the past two decades, improvements in DRAM speeds have not kept pace with increases in processor speeds. As a result, data caches are now widely used for hiding memory latency. Although caches generally work well, some programs fail to use them effectively. 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, why, and how many cache misses occur. Prior approaches for analysing 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 an analysis technique that is more precise than many existing compiler heuristics and that is faster than simulation.

There has been extensive research on improving the cache performance of numerical programs [4, 7, 9, 16, 17]. 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, interchanging, skewing and various combinations of them 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 [7, 9, 12, 14], thereby precluding effective cache utilization. Conflict misses can be significant in caches with low associativity. In such situations programmers often rely on time-consuming cache profiling and performance tuning [10, 11]. There has also been compiler work in tailoring code to reduce conflict misses [1, 6, 9]. Unfortunately, conflict misses are highly sensitive to slight variations in problem size and base addresses [1, 9] 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 *ad hoc* cost models to guide loop transformations [4, 16] or are targeted towards some specific optimization [1, 9]. There has also been some initial work on estimating the number of cache misses in numerical code [7, 14]. 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*)—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.

The CM equations provide a general framework that can be used to: (i) guide a programmer on efficient tuning of the code, (ii) help a compiler in performing code transformations to improve cache usage, (iii) improve the simulation speeds of tools that simulate caches, (iv) tighten bounds on program performance estimates and even (iv) help in better instruction scheduling in super-scalar processors. This paper focuses on the first two of these applications; we discuss how our equations can guide programmers or compilers towards memory optimizations without going through time-consuming cache-profiling. 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 CM Equations within the SUIF compiler system [15]. It successfully generates the equations for numerical loop nests including matrix-multiply, Gaussian elimination, successive over-relaxation (SOR), and many loops from the SPECfp benchmarks. The statistics generated using the equations help focus attention on the badly behaving loop nests and the causes of their poor performance.

In Section 2 the algorithm to generate CM equations is given. Section 3 presents the algorithm to generate the cache misses of a loop nest from all its CM equations. Section 4 shows how these equations can be used to choose data padding/offset amounts or to choose a blocking factor in tiled code. Section 5 describes the future extensions to this work and Section 6 contains our conclusions.

2 Generating the CM Equations

This section describes the algorithm for generating miss equations. For our algorithm we need the *reuse vectors* obtained from relatively standard reuse analysis [16]. We will explain the information needed from reuse analysis as we describe our algorithm.

2.1 Program and Architecture Model

We consider only perfectly nested loops such that all array references are contained within the innermost loops. The algorithm can be extended to handle even some imperfectly nested loops if they have only a single basic block in between the loops in a nest. The loop nest is assumed to consist entirely of *for* loops or *DO* loops. We assume subscript expressions of array references and the bounds of a loop index to be affine combinations of the enclosing loop indices. These restrictions are not too stringent in practice as most array references and loop bounds satisfy this. We have also assumed that the loops contain no conditional expressions. We consider loops with only constant step values as is true for virtually all loops found in practice. Finally, we have considered each loop nest separately ignoring any inter-nest effects. We plan to do inter-nest analysis in the future.

The basic architecture considered is a uniprocessor model with a memory hierarchy. We assume a direct-mapped cache, but we believe the work could be extended to other degrees of set associativity as described briefly in Section 5.

In Table 1, we report 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 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. The "variable bound" entry shows the number of loops which have variables in their loop bounds that cannot be determined at compile time. 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 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 last category 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 analyse, absolutely or parametrically, a significant number of loops appearing in the SPECfp benchmarks ($\approx$ 72% of the total number of loops found).

```
for (i=0; i<32; i++)
  for (k=0; k<32; k++)
    for (j=0; j<32; j++)
      Z[i, j] += X[i, k] * Y[k, j];
```

Figure 1: Matrix multiply loop nest.

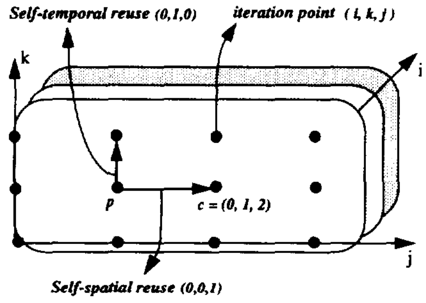

Self-temporal reuse (0,1,0)
iteration point (i, k, j)
k
i
p
c = (0, 1, 2)
j
Self-spatial reuse (0,0,1)

Figure 2: Iteration space of the matrix multiply loop nest. Bold arrows denote reuses of *Z[i,j]*.

2.2 Equation Forming Algorithm

This subsection describes the steps to generate the CM equations. We use the matrix multiplication example given in Figure 1 to illustrate our algorithm.

In order to describe our analysis steps in a concise mathematical form 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 [5, 16]. 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. $\vec{c}$ is an iteration point and corresponds to the iteration $i = 0, k = 1$, and $j = 2$. 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, $\vec{c} \succ \vec{p}$.

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.[1] If it is reusing a previously accessed line we need to know when it

[1] 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 the memory, while a *cache line* refers to the actual cache block to which a memory line is mapped to.

Program	Total	Fcn	Ret	Non affine	Non constant	Non perfect	Variable	Analyzable		Parametrically Analyzable		Non Analyzable
	#L	call		bound	step	loops	bound	#L	#N	#L	#N	#L
TOMCATV	18	0	0	0	0	2	13	5	4	11	8	2
DNASA7	140	12	0	7	2	27	63	44	26	48	37	48
MDLJDP2	31	4	1	2	0	8	13	12	12	5	5	14
SWIM	24	0	0	0	0	0	24	0	0	24	16	0
HYDRO2D	159	3	0	3	2	33	131	20	18	102	63	37
FPPP	33	10	0	0	0	7	25	5	4	11	10	17
ALVINN	23	10	0	0	0	1	0	12	8	0	0	11
SU2COR	127	40	0	0	0	24	104	12	11	51	47	64
MGRID	53	11	1	0	0	9	48	1	1	31	18	21
DODUC	227	29	0	3	0	32	104	163	163	50	50	14
TURB3D	68	8	0	0	0	18	62	2	2	40	30	26
Total	903	127	2	15	4	161	587	276	249	373	284	254

Table 1: 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.)

was last accessed and the reference accessing 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*. The central idea behind our CM equations is to find the loop instances at which reuse does not result in cache hits.

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* [16]. For now, let us assume only one reuse vector of a reference is present. We denote all the misses represented by the equations for a reuse vector as ***misses along that reuse vector.***[2] In Section 3 we show how all the reuse vectors interact to decide the cache misses for a reference.

The CM equations[3] generated here are of two types, namely, the *cold miss equations* and the *replacement miss equations*. The cold miss equations represent the *cold* or *compulsory misses*—misses that occur on the first reference to a memory line. The replacement miss equations represent all other misses including both *capacity* and *conflict misses* [8]. Figure 3 summarizes the process to generate the CM equations.

To generalize our discussion, we will consider generating the CM equations for an arbitrary array reference R_A accessing $A[f_{d_A-1}, \cdots, f_1, f_0]$, where d_A is the number of dimensions of the array A. For any dimension k of the array, the index function is written as f_k and is of the form $f(i_1, i_2, \cdots, i_n)$ where f is a linear function of the loop indices $i_1, i_2, \cdots, i_n$. For the sake of uniformity in the analysis presented here, we assume that all arrays are laid out in memory in a row-major order as in C. Here we have also assumed that all the load/ store references inside a nest correspond to only the array references. This algorithm can be easily extended to handle scalars resulting in a load/store, by considering scalars as a special case of 1-D arrays. While generating the equations, we repeatedly need to find the cache line of an array reference of the form given above. Considering addresses in terms of array element size, the cache line of the reference R_A is given by the following expression:

[2] A miss is said to occur along a reuse vector at those iteration points where, ignoring other reuse vectors, the data reuse is not realized.

[3] The term *equation* has been used loosely to represent a set of simultaneous equalities or inequalities.

```
Algorithm GENERATE:
  Generate the CM equations for a loop nest
 Input:
    Information about the loop nest, array references in
    the nest, reuse vectors and the sequence in which
    those references appear in the generated code.
 Output: A set of CM equations 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 2.2.1).
         2. For each reference in the loop nest
              Generate a replacement miss equation
              (described in Section 2.2.2).
```

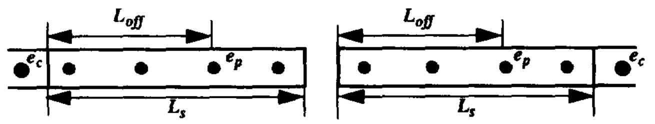
Figure 3: Algorithm to generate all the CM equations.

$$Memory\_Address\_of\_A[f_{d_A-1}, \cdots, f_1, f_0] = Mem_{R_A}$$
$$= offset + f_0 + \sum_{k=1}^{d_A-1} (dim\_size_{k-1} \times f_k)$$
$$Cache\_Line_{R_A} = \lfloor Mem_{R_A} / L_s \rfloor \bmod N_l \qquad (1)$$

where $offset$ = Base address of the array A; f_k = Value of the index expression along the dimension k, evaluated with the current values of the loop indices; dim_size_k = Size of A below the k^{th} dimension; L_s = Cache line size; N_l = Number of cache lines.

2.2.1 Cold Miss Equations

Cold miss equations 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. For each reference, we form a cold miss equation which captures all the cold misses along each reuse vector. In our analysis below $\vec{i} = (i_1, i_2, \cdots, i_n)$ is the current iteration point.



(a) **Negative Stride** $e_p = f_0(p);\ e_c = f_0(p) - L_{off} - 1$

(b) **Positive Stride** $e_p = f_0(p);\ e_c = f_0(p) + (L_s - L_{off})$

Figure 4: Cold miss examples along an array row aligned on a cache line boundary. If e_p is the row element accessed in the previous iteration, the access of row element e_c in the current iteration will result in a cold miss.

• **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 2, $Z[i,j]$ has spatial locality along the vector $(0,0,1)$. So the access of $Z[i,j]$ in the iteration $(0,1,0)$ is a cold miss along that reuse vector, because the previous iteration point along this vector $(0,1,-1)$ is outside the iteration space. If the reuse vector is given by $\vec{r} = (r_1, r_2, \cdots, r_n)$, an access is a cold miss if the corresponding iteration point satisfies any of the following inequalities:

$$i_1 - r_1 < l_1,\ i_1 - r_1 > u_1, \cdots, i_n - r_n < l_n,\ i_n - r_n > u_n \quad (2)$$

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 also be a cold miss along a spatial reuse vector when a new cache line is accessed along that vector. This means that the cache line accessed in the present iteration point $\vec{i}$ by the reference R_A is different from the cache line accessed in the previous iteration point $\vec{p}$ along that vector by the same reference (if self reuse) or a different reference (if group reuse). Hence, there is a cold miss along this reuse vector in the iteration points $\vec{i}$ which satisfy the following relation:

$$\textit{Cache line accessed at } \vec{i} \neq \textit{Cache line accessed at } \vec{p} \quad (3)$$

But, if we assume that the beginning of each row (or the least significant dimension) of arrays are aligned to the cache line boundary, we can further simplify the above equation. Now we can have spatial reuse only if the same row of the same array is accessed at $\vec{i}$ and $\vec{p}$. When we access the same row, all dimensions, except the least significant one, remain the same. We can express this as the two inequalities in Equation 4. This is illustrated in Figure 4.

$$\begin{array}{l} f_0(\vec{i}) < f_0(\vec{p}) - L_{off} \ \ if \ f_0(\vec{i}) < f_0(\vec{p}) \ \ (\textbf{\textit{neg. stride}}) \\ f_0(\vec{i}) > f_0(\vec{p}) + (L_s - 1 - L_{off}) \ \ if \ f_0(\vec{i}) > f_0(\vec{p}) \ \ (\textbf{\textit{pos. stride}}) \\ \quad where \ L_{off} = f_0(\vec{p}) \ mod \ L_s,\ L_s = Cache \ Line \ Size \end{array} \quad (4)$$

When the array is accessed with unit stride we can further simplify Equation 4 to the more familiar form: $f_0(\vec{p})$ $mod\ L_s = 0$ since $f_0(\vec{i}) - f_0(\vec{p}) = 1$ for unit stride. For example, in Figure 2 the cold misses of $Z[i,j]$ along the spatial reuse vector $(0,0,1)$ can be represented as $(j \ mod \ 4) = 0$, assuming the cache line size is 4 array elements.

• **Temporal reuse (self or group):** Cold misses along a temporal reuse vector occur only for the iteration points which lie first along a temporal reuse vector. This is similar to the first case given for spatial reuse vectors and so all the cold misses along this vector are given by the same equations as in Equation 2.

2.2.2 Replacement Miss Equations

All replacement miss equations are formed by a single method, irrespective of whether it is due to self or cross interference and whether the reuse vector is temporal or spatial.

Let us say we want to find the replacement miss equations for the reference R_A accessing $A[f_{d_A - 1}, \cdots, f_1, f_0]$ along the reuse vector $\vec{r} = (r_1, r_2, \cdots, r_n)$. Here we show how to form the replacement miss equation representing the interferences with an arbitrary reference R_B accessing $B[g_{d_B - 1}, \cdots, g_1, g_0]$. ($d_A$ and d_B denote the number of dimensions of the arrays A and B respectively.) For the 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 $\vec{p} = \vec{i} - \vec{r} = (i_1 - r_1, i_2 - r_2, \cdots, i_n - r_n)$. The iteration points at which we can have an interference with R_B, preventing R_A from realizing the reuse at $\vec{i}$, are all the points lying between $\vec{p}$ and $\vec{i}$ which are considered as the set of potential 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 interfering points as $\vec{j} = (j_1, j_2, \cdots, j_n)$, where

$$\begin{array}{ll} \vec{j} & \in \ [\vec{p}, \vec{i})\ \textit{if access of } R_A \textit{ is before } R_B, \\ & \in \ (\vec{p}, \vec{i}]\ \textit{otherwise} \end{array}$$

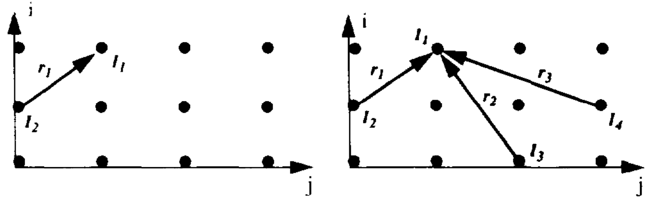
There is an interference if the cache line accessed by R_A in $\vec{i}$ (which is the same as accessed in $\vec{p}$ due to the reuse) is the same as any of the cache lines accessed by R_B at every $\vec{j}$. Equating the appropriate cache lines accessed gives the condition for an interference miss along $\vec{r}$: $Cache\_Line_{R_A}(\vec{i}) = Cache\_Line_{R_B}(\vec{j})$. Substituting in the expressions from Equation 1, we can simplify the resulting equation to the linear Diophantine equation shown in Equation 5.

$$Mem_{R_A}(\vec{i}) = Mem_{R_B}(\vec{j}) + nC_s + b \quad (5)$$

where C_s is the cache size and n is any 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.

For the matrix multiply example shown in Figure 1, consider generating replacement miss equations for $Z[i,j]$ 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 cache with 256 cache lines and 4 array elements per cache line, Equation 6 shows the replacement miss equation for the interferences with $X[i,k]$ along $\vec{r}$. (Here the access of $Z[i,j]$ is after $X[i,k]$ in each loop nest iteration.)

$$\begin{array}{l} Cache\_Line\_Z[i,j] = Cache\_Lines\_X[i',k'] \\ \quad where \ \ (i',k',j') \in ((i,k,j-1),\ (i,k,j)] \\ \Rightarrow \lfloor (4192 + 32i + j)/4 \rfloor \ \ mod \ \ 256 \\ \quad = \ \ \lfloor (2136 + 32i + k)/4 \rfloor \ \ mod \ \ 256 \\ \Rightarrow 4192 + 32i + j = 2136 + 32i + k + 1024n + b \end{array} \quad (6)$$



(a) Considering one reuse vector (b) Considering multiple reuse vectors

Figure 5: Illustration of (a) Theorem 1 and (b) Theorem 2 in the iteration space of a 2D loop nest.

where $n > 0$, $(i, j) \in [(0,0),(31,31)]$, $b \in [-3,3]$. 1024 is the cache size. 4192 and 2136 are the base addresses of the arrays Z and X respectively, and 32 is the row size of both the arrays (All numbers are in units of data element size.)

3 Finding Cache Misses from CM Equations

CM equations are useful because they allow a systematic way to generate all the cache misses of a loop nest from them. This section describes the algorithm to generate all the cache misses of a loop nest from its CM equations. This helps us to understand how the solutions to the CM equations are related to the cache miss instances of a loop nest.

As described in Section 2, for every reference we generate a set of equations for each of its reuse vectors. Every reuse vector has one equation for the cold misses and other equations that represent the conflicts with that reference along that reuse vector. The solution set—the set of all miss instances—can be generated with the help of two theorems which are discussed below. (Formal proofs are not given here due to space constraints.)

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

As an example consider the iteration space shown in Figure 5(a). For a particular reference, assume that one of its reuse vectors is $\vec{r_1}$, which means the reference reuses the same cache-line at the iteration points I_1 and 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 an interference, preventing the reuse to be realized at I_1. However, it only takes one conflicting cache line access between I_2 and I_1 to cause the reference to miss along $\vec{r_1}$ at I_1. This means that we have to take the *union* of the solution sets of all the equations for a particular reuse vector, $\vec{r_1}$, to get all the misses along that vector.

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

As an example consider the iteration space in Fig 5(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 I_1, I_2, I_3, and I_4. The reference can reuse data at I_1 if the data remains in cache after being accessed at either I_2 or I_3 or I_4. If at least one of the reuses are realized, the reference will find the data in cache at I_1. Hence, the reference suffers a miss at I_1, if and only if, I_1 is a miss along *all* the reuse vectors $\vec{r_1}$, $\vec{r_2}$, and $\vec{r_3}$.

For a reference R, say there are m reuse vectors and the number of equations corresponding to the k^{th} reuse vector is given by n_k. From the above two theorems, the set of miss instances of R can be generated from the following expression:

$$\text{Miss Instances of } R = \cap_{k=1}^{m} \left[\cup_{j=1}^{n_k} (\textit{Solution Set of } CM\_eqn_{kj})\right]$$
$$CM\_eqn_{kj} = j^{th} \textit{ CM equation of the } k^{th} \textit{ reuse vector} \quad (7)$$

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.

The equations generated here represent a set of linear equalities or inequalities. Fast methods to solve these kind of equations for most practical loops can be found in [2, 13]. Taking unions and intersections to find the cache misses takes polynomial time in the number of elements of the solution sets. 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 CM equations. 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 [5], could be used. For the closed convex polyhedrons defined by the CM equations, the method takes polynomial time for fixed loop bounds. It is also efficient for parametric loop bounds of practical loop nests.

4 Using CM Equations to Guide Memory Optimizations

We have implemented our algorithm to generate the CM Equations for all the analyzable loops in a program that is integrated with the SUIF compiler system [15]. All necessary information regarding analyzable loops is gathered by a pass integrated into the SUIF analysis. Besides the reuse vectors, our algorithm also needs the exact reference sequence within a loop nest. We extract this from the code generator after register allocation is done.

In this section we describe how these equations can be used to guide different memory optimizations through two examples. The first example shows how the CM equations can be used to reduce cache misses by changing the base addresses and the row sizes (*i.e.* padding) of the arrays referenced in a loop nest which is taken from the SPECfp benchmarks. The second example shows how the equations can be used for efficient block size selection for the blocked matrix multiply loop nest. While these optimizations have been addressed in isolation by past work [1, 6, 9], these examples illustrate how CM equations provide an accurate, unifying framework to drive optimizations.

4.1 Example 1: Padding and Changing Base Address

The code in this example is from the *alvinn* program of the SPECfp benchmarks. When we run our equation generator on the loop nests, it generates a collection of CM equations summarizing the memory behavior of each nest. We focus here on one of the analyzable loop nests (shown in Figure 6) which suffers significantly from replacement misses. In this loop, roughly 187000 out of 306000 misses are replacement misses. We use the CM equations to eliminate these misses.

Equation 8 and Equation 9 are generated as the replacement miss equations for the reference to *i_h_weights[hu][iu]*. Equation 8 represents the self-interferences while Equation 9 gives the cross-interferences with the reference *i_h_w_ch_sum_*

```
for (iu=0; iu<1221; iu++)
  for (hu=0; hu<30; hu++) {
    i_h_weights[hu][iu] +=
          i_h_w_ch_sum_array[hu][iu]* i_h_lrc;
    i_h_w_ch_sum_array[hu][iu] *= ALPHA;
  }
```

Figure 6: Example loop nest from the *alvinn* benchmark.

array[hu][iu].

$$1221hu + iu = 1221hu' + iu' + nC_s + b \quad (8)$$

$$82110 + 1221hu + iu = 45480 + 1221hu' + iu' + nC_s + b \quad (9)$$

where $(iu', hu') \in [(iu-1, hu+1), (iu, hu-1)]$ in Equation 8, and $(iu', hu') \in [(iu-1, hu), (iu, hu-1)]$ in Equation 9. For both the equations, $hu \in [0, 29]$, $iu \in [0, 1220]$, $b \in [-3, 3]$, and $n \in [0, \infty)$. For this example, we assume a cache size, C_s, of 1024 and a line size of 4 elements. In Equation 9, 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 row 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.)

Equations 8 and 9 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 offsets and the row sizes (*i.e. padding* the arrays). For this reason, we will replace all terms related to the base addresses and the row sizes with parameters. Equations 10 and 11 are derived from the Equations 8 and 9 respectively using standard algebraic techniques. The parameter k is related to the base addresses while P is related to the row sizes. Our goal is to see which values of k and P will result in the fewest interference misses.

$$Phu_{diff} - 1024n' = (b - iu_{diff}),\ n' \in [-29, \infty) \quad (10)$$

$$k + Phu_{diff} - 1024n' = (b - iu_{diff}),\ n' \in [-64, \infty) \quad (11)$$

In Equation 10, $hu_{diff} = (hu - hu') \in [-29, -1]$ if $iu_{diff} = (iu - iu') = 1$ and $hu_{diff} \in [1, 29]$ if $iu_{diff} = 0$. In Equation 11, $hu_{diff} \in [-29, 0]$ 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. Though Equations 10 and 11 are independent equations, they are connected through the parameter P. Changing P in one equation will affect the other. We consider Equation 10 first to determine which values of P would eliminate its solutions. We then use one of these P's in Equation 11 to find the values of k that eliminate its solutions.

From basic number theory, we see 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 10 has no solution [2]. 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 10 will have no solution. As $4 < |GCD(P, 1024)| < 36$, we can write $P = 8t, 16t$, or $32t$ where t is any odd positive integer.

```
for (kk=0; kk<N; kk+=B_k)
  for (jj=0; jj<N; jj+=B_j)
    for (i=0; i<N; i++)
      for (k=kk; k<min(kk+B_k-1, N); k++)
        r = X[i,k];
        for (j=jj; j<min(jj+B_j-1, N); j++)
          Z[i,j] += r * Y[k,j];
```

Figure 7: Blocked matrix multiplication loop nest.

By similar reasoning as above, we can say that Equation 11 will have no solution if $GCD(k, P, 1024)$ does not divide $(b - iu_{diff})$ for $(b - iu_{diff}) \neq 0$. For $(b - iu_{diff}) = 0$, rewriting the equation as $Phu_{diff} - 1024n' = -k$, the equation will have no solution if $GCD(P, 1024)$ does not divide k. If we choose $k = 8$ and $P = 16t$ we can satisfy all the above criteria and make both Equation 10 and 11 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.)

Tracing back the origin of k, it can be seen that setting $k = 8$ corresponds to changing the offset of the array *i_h_weights* from 82110 to 81328. On the other hand, setting $P = 208$ corresponds to changing the size of each row of the arrays from 1221 to 1232. These simple changes in the array layout eliminated all the interference misses in the loop nest of Figure 6. (The equations for the reference *i_h_w_ch_sum_array[hu][iu]* are similar and required similar changes to eliminate misses.) Overall, this example has shown how expressing a loop nest's cache misses in the form of linear Diophantine equations allows us to methodically identify ways of padding and aligning data in order to avoid interference misses.

4.2 Example 2: Selecting Block Size for Loop-Tiling

This case study deals with a familiar loop optimization for scientific code: *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 [3, 16, 17]. This 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 [9]. This led to recent research on choosing appropriate tile sizes that would reduce conflict misses as well [6, 9]. 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. Here we will show how to use our CM equations, 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, but here we focus on the former.

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 7. Our analysis could be easily generalized for all other tiled loops handled in previous work. In Figure 7, *B_k* by *B_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

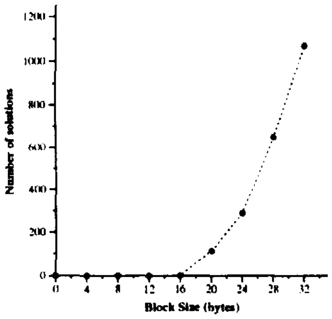


Figure 8: Solutions to self-interference equation of $Y[k,j]$.

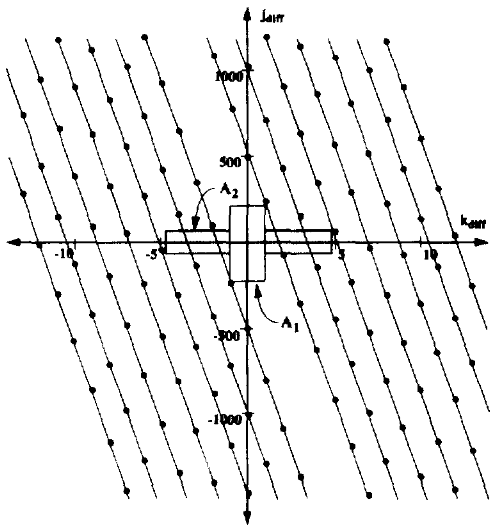


Figure 9: Solution plot of the self-interference equation of $Y[k,j]$. The parallel lines correspond to sets of solutions for different values of n. Dots along each line represent the integral solution points (interference misses). Shaded regions correspond to possible blocks with no self-interference.

et al. [9], 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[k,j]*. In fact, after a certain block size, these self-interferences outweigh the performance gain expected from increasing block size [9]. The self-interference equation of *Y[k,j]* for each execution of the blocked code in Figure 7 is given by Equation 12.

$$295k + j \quad = \quad 295k' + j' + nC_s + b \qquad (12)$$

where $(k,j) \in [(0,0),(B\_k-1, B\_j-1)]$, $(k',j') \in [(0,0),(k,j-1)]$ or $[(k,j+1),(B\_k-1, B\_j-1)]$, $b \in [-0,0],$, $C_s = 512$ and n is any integer except 0. As before, C_s is the cache size, and b is the cache line size minus one. Equation 13 is a simplified version and includes all the solutions of Equation 12.

$$295k_{diff} + j_{diff} \quad = \quad 512n \qquad (13)$$

where, $|k_{diff}| = |k - k'| < B\_k$ and $|j_{diff}| = |j - j'| < B\_j$.

Figure 8 plots the number of solutions of Equation 12 for different square block sizes. These data are consistent with the self-interference misses of blocked matrix multiplication presented in [9]. There are no self-interferences of

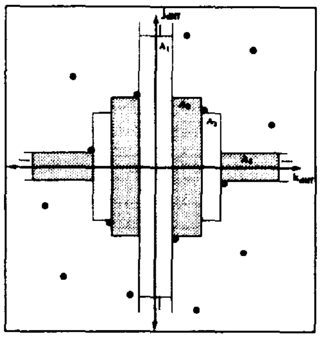


Figure 10: Finding regions without integral solution points.

Y[k,j] until a block size of 16 by 16. Thereafter, it increases drastically with increasing block size. Our goal is to use the CM equations to identify the largest block size with no self-interference. That means we need to find the largest value of B_k by B_j such that Equation 13 has no solution. Lam *et al.* considered only square blocks, but Coleman *et al.* [6] showed that we can get better performance with rectangular blocks without restricting ourselves to square blocks. We will show that CM equations can guide us to choose the best block, square or rectangular.

Figure 9 illustrates the block-selection problem. The lines in the figure are plots of Equation 13 for different values of n. The bold dots show the self-interference instances, that is the solution points for integral values of x and y. Since $|k_{diff}| < B\_k$ and $|j_{diff}| < B\_j$, Equation 13 will have no solution if there are no integral solution points in the region defined by $k_{diff} = B\_k, k_{diff} = -B\_k$ and $j_{diff} = B\_j, j_{diff} = -B\_j$. Thus, such an ***empty region*** corresponds to the selection of a block of size Bk by Bj which would eliminate all self-interferences. For example, the shaded region A_1 in Figure 9 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 9 (*i.e.* $x \geq 0$) 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 $B\_k = 1$ and $B\_j = 216$ since $(x,y) = (1,217)$, 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. Now the region is shrunk along the j_{diff} axis to just exclude that solution point and expanded again along k_{diff} as much as possible without including any integral solution point. As a result of the above operation we obtain the region A_2. We repeat the above process to find the ***maximal empty regions***[4] until the region's area exceeds the cache size or reduces to zero. Table 2 lists the maximal empty regions A_1 through A_7 found by the above algorithm. Figure 10 depicts the formation of the first four regions. Now, each of these regions define a block size with no self-interference. For example, region A_4 defines the block size of B_k by B_j = 25 by 16 which clearly includes the square block solution of 16 by 16 found experimentally from Figure 8. For the largest block size, we choose the region with the maximum area which, in this case, is the block A_4. Our algorithm does not require finding integral solution points for all n, j, k. Rather, we need only identify those close to the axes as we stretch the rectangles.

[4] Regions whose area cannot be increased without decreasing the height.

Region	B_k	B_j	Block size (B_k * B_j)
A_1	1	216	216
A_2	4	77	308
A_3	6	60	360
A_4	25	16	400
A_5	32	9	320
A_6	58	6	348
A_7	150	2	300

Table 2: Blocks with no self-interference (from our algorithm).

5 Future Work

One of the important extensions to our analysis is to handle *set-associative caches*. As associativity does not affect the cold misses of a program, our methods for generating cold miss equations hold for any degree of associativity. For replacement misses we need to modify the constraints given in Equation 5. For a k-way set-associative cache with LRU replacement policy there must be at least k interfering accesses to the same cache line before a data can be replaced from its cache line. Hence, the constraints need to be modified such that an iteration point $\vec{i}$ will be a solution if and only if there are k different $\vec{j}$'s satisfying Equation 5 for that $\vec{i}$.

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 [12]. So it should be enough to find reuse vectors only between adjacent nests for most practical purposes.

In order to help code optimisations, as described in Section 4, we try to find values of parameters that would eliminate or reduce the number of solutions to the CM equations. As one possible way to automate that analysis, we hope to use the extension of Pugh's *Omega test* [13] 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 [5] 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 analyse and optimise loops with parametric loop bounds.

6 Conclusions

This paper has demonstrated the use of CM equations as a means of precisely characterizing the cache 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 described algorithms for identifying cache misses without simulation, and we used examples from SPECfp benchmarks to demonstrate practical applications of this information.

There are numerous applications of this work that range from performance estimation to code optimisations. By automating CM equation analysis within the compiler, one can guide compiler memory optimizations. In addition to loop optimizations, CM equations may also be useful in instruction scheduling to avoid stalls due to long cache miss penalties. We also hope to use the equation framework to improve the performance of cache simulation tools. Essentially, CM equations can accelerate simulations by identifying cache misses and summarizing much of a program's memory behavior at compile-time, before the cache simulation is run. Overall, Cache Miss Equations provide an unique, systematic framework for accurately assessing the frequency and causes of cache misses in loop-oriented code; this accurate framework will support a range of future uses.

References

[1] 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.

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

[3] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proc. Supercomputing '92*, Nov. 1992.

[4] S. Carr, K. S. McKinley, and C-W. Tseng. Compiler optimizations for improving data locality. In *Proc. Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.

[5] 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.

[6] 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.

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

[8] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec. 1989.

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

[10] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, Oct. 1994.

[11] 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.

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

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

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

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

[16] 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.

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