Analysing Cache Memory Behaviour for Programs with IF Statements

Xavier Vera*
Institutionen för Datateknik
Mälardalens Högskola
Västerås, Sweden
xavier.vera@mdh.se

Jingling Xue
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia
jxue@cse.unsw.edu.au

UNSW-CSE-TR0107

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney, NSW 2052, Australia

---

[*] This work was done during the 1st author's visit to the 2nd author.

Abstract

Cache memories are widely used to bridge the increasing performance gap between processors and main memories. However, cache memories are effective only when the program exhibits good cache locality. Analytical methods such as the Cache Miss Equations (CMEs) use mathematical formulas to provide a precise characterisation of the number and causes of cache misses in loop-oriented programs. The information gathered can be used to guide locality enhancement compiler optimisations. Unfortunately, all existing analytical methods are limited to special forms of perfectly nested loops, which, for example, must be free of IF statements.

This paper presents an analytical method for analysing the cache behaviour of perfectly nested loops containing IF statements with compile-time-analysable conditionals. We demonstrate that our method, together with the compiler technique loop sinking, can be used to analyse a large number of imperfect loop nests. By analysing the loop nests in SPECfp95, Perfect Suite, Livermore kernels, Linpack and Lapack, we find that our method enables 17% more loop nests to be analysed than previously. This represents an important step towards analysing complex program constructs in real programs.

1 Introduction

Data caches are widely used to bridge the increasing performance gap between processors and main memories. However, caches are effective only when programs exhibit sufficient data locality in their memory access patterns. Both programmers and compiler transformations often restructure a program to improve its memory behaviour. In both cases, it is necessary to have detailed knowledge about the number and causes of cache misses in the program.

Several approaches for analysing cache behaviour can be identified. Cache simulation techniques are accurate and can report a rich source of information about a program's cache behaviour. Based usually on trace-driven simulation [22], they are both time- and space-consuming and do not provide insights about the causes of cache misses. Hardware counters [1], although fast and accurate, are architecture-dependent and do not usually provide information about the causes of cache misses. Analytical methods such as the Cache Miss Equations (CMEs) [11] attempt to set up mathematical formulas to provide a precise characterisation of the number and causes of cache misses in a program. These formulas can be potentially exploited to guide a range of memory optimisations and improve the simulation times of tools like cache simulators and profilers.

The CMEs [11] represent an analytical method for analysing the cache behaviour of loop-oriented programs. These programs typically spend a considerable amount of time operating on arrays in loop nests. The CMEs describe the relationships among loop indices, array sizes, base addresses and the cache parameters for cache misses in a loop nest using a set of Diophantine equations (which consists of actually both equalities and inequalities). This characterisation makes it possible to understand the causes behind cache misses and helps reduce these misses in a systematic manner. However, computing the *exact* number of cache misses from the CMEs is computationally expensive. Some statistics-based methods have been reported to produce an accurate estimate of such misses [2, 12, 24]. In certain compiler transformations, it is possible to reduce the number of cache misses by reasoning about the causes of some cache misses expressed in the CMEs without requiring the CMEs to be solved. Two classic applications are tiling and padding [11].

Unfortunately, the CMEs are limited to perfectly nested loops, which must be free of several language constructs such as IF statements, subroutine calls and return statements [11]. As a result, only portions of a program can be analysed. In this paper, we overcome one of these limitations by tackling the problem of analysing programs with IF statements.

This paper makes the following contributions. First, we present an analytical method for analysing the cache behaviour of perfectly nested loops containing IF statements with compile-time-analysable conditionals. These conditionals can contain common ABS, MOD, MIN and MAX operators. In particular, we discuss the derivation of reuse vectors in the presence of IF statements. Second, we discuss how our method can be used to analyse those imperfectly nested loops that are sinkable by the compiler technique loop sinking. Third, we present our experimental results in a collection of programs from SPECfp95, Perfect Suite, Livermore kernels, Linpack and Lapack. By analysing the loop nests in these benchmarks, we find that our method enables 17% more loop nests to be analysed than previously. This represents an important step towards a mechanical analysis of complex language constructs.

The rest of this paper is organised as follows. Section 2 presents our analytical method,

```
PROGRAM COND
PARAMETER (N = 512, M = 512)
REAL*8 a(N+1,M+1), b(N+1,M+1), z(N+1,M+1)
REAL*8 vnew(N+1,M+1), unew(N+1,M+1)
DO I1 = 1,N
  DO I2 = 1,M
    a(I1+1,I2) = b(I1+1,I2)+ z(I1+1,I2+1)  ≜ Ref1
    IF (I1+I2.GE.200) THEN
      vnew(I1,I2+1) = 1+ z(I1,I2+1)  ≜ Ref2
    ENDIF
    IF (I1.LE.100) THEN
      unew(I1,I2) = b(I1,I2)+ z(I1,I2)  ≜ Ref3
    ENDIF
  ENDDO
ENDDO
END
```

Figure 1: A running example.

which works for any k-way set associative caches. Section 3 applies our method to analyse imperfectly nested loops. Section 4 presents some experimental results. Section 5 summarises the related work. Section 6 concludes the paper and discusses some future work.

2 Analysing Cache Behaviour

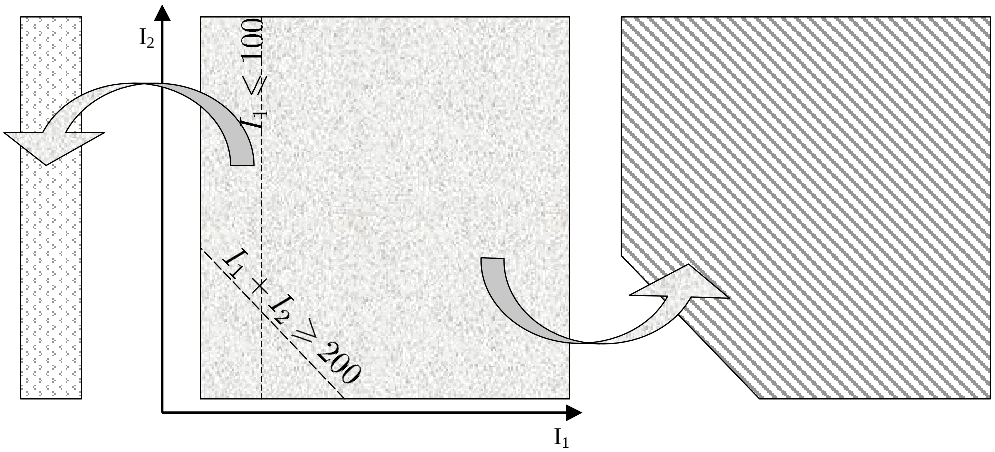
We represent a perfect loop nest of depth n with affine loop bounds as an n-dimensional convex polyhedron in $\mathcal{Z}^n$ called the *iteration space* of the loop nest. Every point in the iteration space is known as an *iteration (point)* and is identified by its index vector $\vec{\imath} = (i_1, i_2, \ldots, i_n)$, where i_k is the index of the k-th loop (counting from the outermost to the innermost). We write $\prec$ to denote the lexicographic "less than" operator so that if $\vec{\imath} \prec \vec{\jmath}$, then $\vec{\imath}$ executes before $\vec{\jmath}$. In a sequential loop nest, all its iterations are executed in their lexicographic order $\prec$.

We assume a uniprocessor with a k-way set associative data cache using a least-recently-used (LRU) replacement policy. In the case of write misses, we assume a fetch-on-write policy so that both reads and writes are modelled identically. A *memory line* refers to a cache-line-sized block in the main memory while a *cache line* refers to the actual cache block to which a memory line is mapped. In this paper, $Mem\_Line_R(\vec{\imath})$ ($Cache\_Set_R(\vec{\imath})$) denotes the memory line (cache set) to which the memory address accessed by reference R at iteration $\vec{\imath}$ is mapped. In a k-way set associative cache, a cache set contains k distinct cache lines.

Let $Mem\_Addr_R(\vec{\imath})$ be the memory address of the reference R at iteration $\vec{\imath}$. We have:

$$\begin{aligned} Mem\_Line_R(\vec{\imath}) &= \lfloor Mem\_Addr_R(\vec{\imath})/\mathcal{L} \rfloor \\ Cache\_Set_R(\vec{\imath}) &= Mem\_Line_R(\vec{\imath}) \bmod \mathcal{N} \end{aligned}$$

where $\mathcal{L}$ is the cache line size (in bytes) and $\mathcal{N} = \mathcal{C}/k$ is the number of cache sets.



(a) Ref_3 (b) Ref_1 (c) Ref_2

Figure 2: RISs of the three z references in Figure 1.

Like the CMEs [11], the analytical method proposed in this paper describes all cache misses in a loop nest using a set of equalities and inequalities (collectively referred to as *miss equations*). These equations describe the relationships among loop indices, array sizes, base addresses and the cache parameters for a loop nest. They can be further manipulated to find when and why cache misses occur in the loop nest. The information obtained can be used to guide automatic compiler optimisations, which is beyond the scope of this paper.

When analysing IF statements, the major complication comes from the fact that different references may be accessed in different parts of the iteration space, which may or may not overlap. In this section, we describe various concepts and steps involved in formulating and solving the miss equations for a loop nest. Our running example is given in Figure 1. The three highlighted z references will be used later for illustrations. Ref_1 is not guarded while Ref_2 and Ref_3 are guarded by conditionals that are affine expressions of loop indices.

The rest of this section is organised as follows. Section 2.1 introduces the concept of reference iteration space. Section 2.2 discusses the derivation of reuse vectors and some complications that arise in the presence of IF statements. Section 2.3 describes the miss equations used for representing all cache misses in a loop nest. Section 2.4 gives two algorithms for computing the number of cache misses from these miss equations.

2.1 Reference Iteration Spaces

We define the *reference iteration space (RIS)* of a reference as the set of iteration points where the reference is accessed. If a reference is not guarded by a conditional, its RIS is the entire iteration space of the loop nest. Otherwise, the RIS can be a subspace of the iteration space. Figure 2 displays the RISs for the three z references highlighted in Figure 1.

Our analytical method can deal with any IF conditionals involving loop indices and compile-time constants. These are the conditionals that can be analysed at compile-time without relying on any runtime information about the conditionals involved. However, data-dependent conditional expressions such as *a(i,j).EQ.0* are beyond our current method and their analysis is part of our future work. In loop-oriented programs with regular compu-

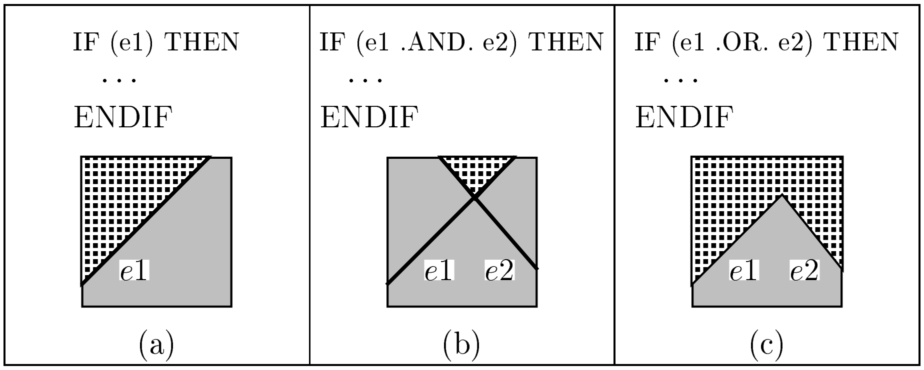


Figure 3: Some commonly occurring RISs (in dotted areas).

tations, almost all data-independent conditionals are affine expressions of loop indices and compile-time constants. In all programs analysed from SPECfp95, Perfect Suite, Livermore Kernels, Linpack and Lapack, we have not found a single case that is not affine.

If a reference is guarded by affine conditionals (containing possibly .AND., .OR. or .NOT. operators), the corresponding RIS can always be expressed as a finite union of convex polytopes in $\mathcal{Z}^n$. Such a RIS can be manipulated by the Omega library [18] and its volume computed using methods [5, 13, 18] for various purposes. Figure 3 depicts three commonly occurring cases. In Section 2.4, we discuss a simple yet efficient technique used for computing the volume of a RIS required in our statistics-based algorithm for solving miss equations.

2.2 Computing Reuse Vectors

Reuse vectors are a mathematical representation of data reuse in a loop nest [27]. They determine the direction and distance of the reuse between uniformly generated references. A set of references, $\{a_1(f_1(\vec{\imath})), a_2(f_2(\vec{\imath})), \ldots, a_m(f_m(\vec{\imath}))\}$, is *uniformly generated* if (a) all references are to the same array variable and (b) $f_1(\vec{\imath}), f_2(\vec{\imath}), \ldots, f_m(\vec{\imath})$ are all affine functions of loop indices and compile-time constants sharing the same linear part, i.e., $f_k(\vec{\imath}) = H\vec{\imath} + \vec{h}_k$ for all k. For example, $\{a(i), a(i+1)\}$ is uniformly generated. So is $\{b(i,j), b(i+1,j), b(i,j-2)\}$. But $\{a(i), a(2i)\}$ and $\{b(i,j), b(i,i), b(i,2j)\}$ are not. Scalars are considered either register-allocated or as one-dimensional arrays of single elements each. Non-affine references like $a(i^2, j)$, which yields little reuse [27] and are uncommon anyway, are ignored.

Let R_1 and R_2 be two references to the same array variable. If R_2 at iteration $\vec{\imath_2}$ accesses the same memory line as the reference R_1 at iteration $\vec{\imath_1}$, i.e., $Mem\_Line_{R_1}(\vec{\imath_1}) = Mem\_Line_{R_2}(\vec{\imath_2})$, where $\vec{\imath_1} \prec \vec{\imath_2}$, we say *there exists reuse from R_1 to R_2 (or R_2 reuses R_1) along direction $\vec{r} = \vec{\imath_2} - \vec{\imath_1}$, and $\vec{r}$ is called a reuse vector.* The reuse is *temporal* if $Mem\_Addr_{R_1}(\vec{\imath_1}) = Mem\_Addr_{R_2}(\vec{\imath_2})$ and *spatial* otherwise. (Thus, a temporal reuse will not also be classified as a spatial reuse in this paper.) In addition, the reuse is said to be a *self-reuse* if R_1 and R_2 are identical and a *group-reuse* otherwise. Thus, there are four kinds of reuse: self-temporal, group-temporal, self-spatial and group-spatial. We shall speak of self-reuse vectors (for self-reuse), self-spatial reuse vectors (for self-spatial reuse), and so on.

A reuse between two iterations does not imply that the reuse can be *realised* in the

Reusing Reference	Reused Reference		Reuse Vectors
z(I_1+1, I_2+1)	Self Spatial		(1,0)
	z(I_1, I_2+1)	Group-Spatial	(1,0)
z(I_1, I_2+1)	Self-Spatial		(1,0)
	z(I_1+1, I_2+1)	Group-Spatial	(0,0)
		Group-Temporal	(1,0)
z(I_1, I_2)	Self-Spatial		(1,0)
	z(I_1+1, I_2+1)	Group-Spatial	(0,1)
		Group-Temporal	(1,1)
	z(I_1, I_2+1)	Group-Temporal	(0,1)

Table 1: Reuse vectors for the z references in Figure 1.

cache. The memory line brought into the cache at the earlier iteration may have been evicted by other memory accesses between the two iterations before it gets reused. The basic idea behind the miss equations is to identify the iterations in which reuse results in cache misses. This requires all reuse vectors of a reference to be computed if all cache misses are to be characterised in the miss equations. Ignoring a reuse vector may cause a slight over-estimation of cache misses. The reuse vectors of a reference are computed by using Wolf and Lam's reuse framework [27] to provide basic reuse vectors and some extensions described in [23] to provide additional reuse vectors specific to the shape of the iteration space and the cache parameters used (the very information ignored in Wolf and Lam's framework). The reuse vectors are computed only for sets of uniformly generated references [27, 29].

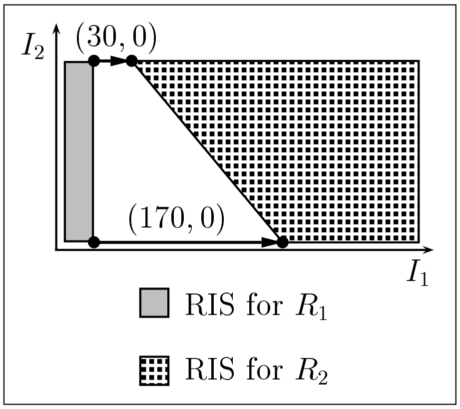
When a loop nest contains IF statements, different references can be executed in different RISs. However, we will compute the reuse vectors for all references by ignoring the conditionals present in the loop nest. Our justification for doing so is presented in the following paragraphs. Table 1 lists all reuse vectors used for the three z references highlighted in Figure 1. For this example, the reuse vectors calculated using Wolf and Lam's reuse framework are sufficient. In FORTRAN, all arrays are stored in the column major order. Thus, all three references are associated with the self-spatial reuse vector $(1, 0)$. The reference $z(I_1 + 1, I_2 + 1)$ may reuse the same cache line that $z(I_1, I_2 + 1)$ accessed one iteration earlier of the outer loop. Hence, there is a group-spatial reuse vector $(1, 0)$ between the two references. The other reuse vectors can be understood similarly.

The justification for ignoring all conditionals in the derivation of reuse vectors is as follows. The self-reuse vectors are calculated for a single reference in its own RIS. Whether the RIS of the reference is the entire iteration space or its strict subset is immaterial.

In the case of group-reuse vectors, the two references involved can have different RISs. Some complications can arise at the boundaries of the RIS of the reusing reference being analysed. For illustration purposes, these complications are illustrated by two extreme examples below. Figure 4 illustrates some complications in the derivation of group-temporal reuse vectors. R_2 (the reusing reference) at every point (I_1, I_2) on the left boundary of its RIS may reuse R_1 (the reused reference) at the point $(30, I_2)$ on the right boundary of R_1's RIS along the group-temporal reuse vector $(I_1 - 30, 0)$. If we ignore the two conditionals to analyse the reuse between the two references, the group-temporal reuse vector $\vec{r} = (0, 0)$ will

```
DO I1 = 1,400
  DO I2 = 1,141
    IF (I1.LE.30) THEN
      R1: A(I2)
    ENDIF
    IF (I1 + I2.GE.201) THEN
      R2: A(I2)
    ENDIF
  ENDDO
ENDDO
```

(a) Code



(b) RISs

Figure 4: Derivation of group-temporal reuse vectors.

describe correctly the reuse from R_1 to R_2. When the miss equations for R_2 are formulated, the two conditionals must be taken into account (as discussed in Section 2.3). Then this reuse vector will be ignored since the two RISs do not overlap. As a result, the number of cache misses for R_2 on the left boundary of its RIS may be over-estimated. For practical applications, such an over-estimation is negligible because (a) the over-estimation occurs only on a facet of a RIS (e.g., the left boundary of R_2's RIS) and (b) the underlying reference may reuse on the facet via other reuse vectors. In the example, R_2 may reuse from itself along the self-spatial reuse vector $(1, -1)$. Thus, only a small fraction of these boundary points are mis-predicted.

Figure 5 illustrates some complications in the derivation of group-spatial reuse vectors. R_2 at every point (I_1, I_2) on the line segment $I_1 - I_2 = 10$ that is confined in the iteration space reuses from R_1 along the group-spatial reuse vector $(I_2, 1)$, where $2 \leqslant I_2 \leqslant 90$. When the two conditionals are ignored, the amount of group-spatial reuse between the two references will be approximated by $(0, 1)$ and some other extended reuse vectors. When the miss equations for R_2 are formulated, the two conditionals must be taken into account (as discussed in Section 2.3). Then these reuse vectors will be ignored since they does not actually describe any group-spatial reuse. An over-estimation of cache misses in this case is negligible for the same reasons given above when the group-temporal reuse vectors are discussed. Note that programs like the one illustrated in Figure 5 rarely occur in practice.

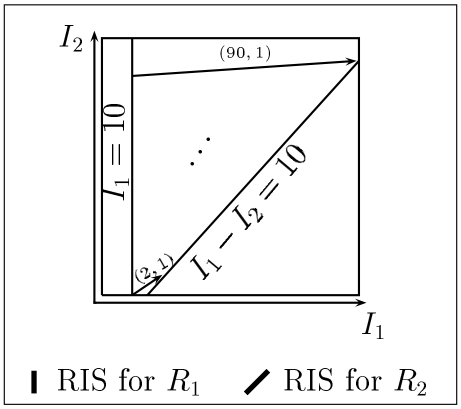
We have done extensive experiments using a collection of benchmark programs. The number of cache misses obtained from our method are always close to the actual number of cache misses obtained by simulation. For practical loop nests with regular data accesses, deriving reuse vectors while ignoring conditionals is a feasible approach.

2.3 Forming the Miss Equations

A reference R at an iteration $\vec{\imath}$ suffers from a *compulsory* or *cold* miss if $Mem\_Line_R(\vec{\imath})$ is being accessed for the very first time and a *replacement miss* if $Mem\_Line_R(\vec{\imath})$ was accessed before and evicted later so that it is no longer in the cache when $Mem\_Addr_R(\vec{\imath})$ is accessed.

```
DO I1 = 1,100
  DO I2 = 1,100
    IF (I1.EQ.10) THEN
      R1: A(I2, I2 + 1)
    ENDIF
    IF (I1 − I2.EQ.10) THEN
      R2: A(I2, I2)
    ENDIF
  ENDDO
ENDDO
```

(a) Code



(b) RISs

Figure 5: Derivation of group-spatial reuse vectors.

Replacement misses encompass both capacity and conflict misses.

There are two types of miss equations: *compulsory or cold miss equations* and *replacement miss equations*. These equations are formulated for a single generic reuse vector of a fixed but arbitrary reference. If the reference has only that reuse vector, the solutions to the cold miss equations represent precisely the cold misses of the reference, and the solutions to the replacement equations represent precisely the replacement misses of the reference. If the reference has other reuse vectors, the solutions to the two types of equations represent only potential cache misses. How to find cache misses in the presence of multiple reuse vectors is discussed in Section 2.4.

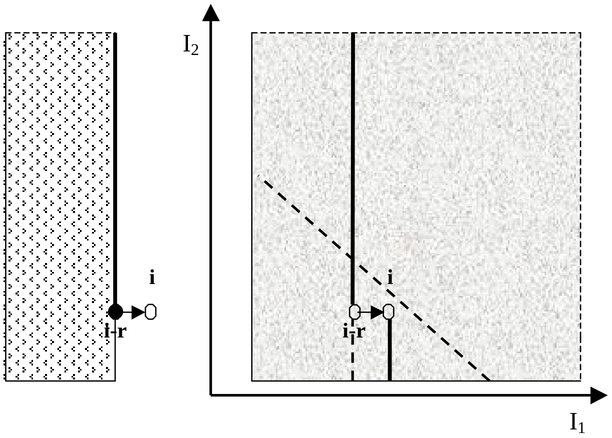
In this section, we describe the miss equations for a single reference R_c along a single reuse vector $\vec{r}$. Let R_p be the reference such that R_c reuses from R_p along $\vec{r}$. Let R_i be an intervening reference that may prevent such a reuse from being realised. Here, the subscripts c, p and i denote mnemonically "consuming", "producing" and "intervening" references, respectively. Let RIS_{R_c}, RIS_{R_p} RIS_{R_i} be the RISs for R_c, R_p and R_i, respectively. It is important to note that some or all of the three references can be identical.

2.3.1 Cold Miss Equations

The cold miss equations for R_c along $\vec{r}$ are to investigate if the memory line $Mem\_Line_{R_c}(\vec{\imath})$ accessed by R_c at iteration $\vec{\imath}$ is accessed for the first time. It then follows that R_c suffers a compulsory or cold miss at iteration $\vec{\imath}$ along $\vec{r}$ if $\vec{\imath}$ is a solution to the following equations:

$$\left.\begin{array}{c} \vec{\imath} \in RIS_{R_c} \\ \text{and} \\ \big(\vec{\imath} - \vec{r} \notin RIS_{R_p} \\ \text{or} \\ Mem\_Line_{R_c}(\vec{\imath}) \neq Mem\_Line_{R_p}(\vec{\imath} - \vec{r})\big) \end{array}\right\} \tag{1}$$

If $\vec{r}$ is temporal, the second equation, which always evaluates to false (due to the temporal reuse), is redundant. Then the cold miss equations simplify to:



(a) $R_i = Ref_3$ (b) $R_i = Ref_1$ (c) $R_i = Ref_2$

Figure 6: The interference sets with the three z references when $R_c = R_p = Ref_1$ along $\vec{r} = (1, 0)$ for the running example. For illustration purposes, the point $\vec{\imath} \in RIS_{Ref_1}$ being analysed is chosen such that $\vec{\imath} \notin RIS_{Ref_2}$ and $\vec{\imath} \notin RIS_{Ref_3}$. In each case, the interference set consists of the solid line(s) and $\vec{\imath}$ or $\vec{\imath} - \vec{r}$ if the corresponding point is a fat point.

$$\begin{array}{c} \vec{\imath} \in RIS_{R_c} \\ \vec{\imath} - \vec{r} \notin RIS_{R_p} \end{array}$$

2.3.2 Replacement Miss Equations

The replacement miss equations for R_c along $\vec{r}$ are to investigate if R_c at iteration $\vec{\imath}$ can reuse the memory line that R_p accessed at iteration $\vec{\imath} - \vec{r}$ subject to the interferences of the memory accesses from R_i at all points executed between $\vec{\imath} - \vec{r}$ and $\vec{\imath}$. These interferences are known as *self-interferences* if R_c and R_i are identical and *cross-interferences* otherwise.

The iteration points at which an interference may occur are the points that are located between $\vec{\imath} - \vec{r}$ and $\vec{\imath}$ and that are contained in RIS_{R_i}. All these points belong to a so-called *interference set*, denoted J_{R_i}. Whether the two end points $\vec{\imath}$ and $\vec{\imath} - \vec{r}$ are included depends on whether some or all three references are identical or not and the relative lexical order of these references. In all cases, the reference set for R_i is defined as follows:

$$J_{R_i} \;=\; \{\vec{\jmath} \in RIS_{R_i} \mid \vec{\jmath} \in \ll \vec{\imath} - \vec{r}, \vec{\imath} \gg\}$$

where '$\ll$' is '[' if R_i is lexically after R_p and '(' otherwise and '$\gg$' is '[' if R_i is lexically before R_c and '(' otherwise. A reference is neither lexically before nor lexically after itself.

Figure 6, a zoomed-in version of Figure 2 at its bottom-left corner, shows the interference sets with the three z references when Ref_1 is analysed along $\vec{r} = (1, 0)$.

There is potentially a *cache set contention* if the cache set accessed by R_c at $\vec{\imath}$ (which is the same as accessed by R_p at $\vec{\imath} - \vec{r}$ due to the reuse) is the same as any of the cache sets accessed by R_i at every $\vec{\jmath} \in J_{R_i}$. The replacement miss equations for an interference at $\vec{\imath}$

along $\vec{r}$ are:

$$\left.\begin{array}{c} Mem\_Line_{R_c}(\vec{\imath}) = Mem\_Line_{R_p}(\vec{\imath} - \vec{r}) \\ \vec{\imath} \in RIS_{R_c} \\ \vec{\imath} - \vec{r} \in RIS_{R_p} \\ Cache\_Set_{R_c}(\vec{\imath}) = Cache\_Set_{R_i}(\vec{\jmath}) \\ \vec{\jmath} \in J_{R_i} \end{array}\right\} \qquad (2)$$

where the first three lines dictate the reuse of a memory line from R_p to R_c along $\vec{r}$ and the last two lines define all possible interferences of R_c caused by R_i.

In a k-way set associative cache with a LRU replacement policy, it takes at least k different cache set contentions to cause the least-recently-used cache line to be evicted from the cache set. However, the existence of k distinct solutions $\vec{\jmath_1}, \vec{\jmath_2}, \ldots, \vec{\jmath_k}$ to the replacement equations (2) does not mean the existence of k distinct cache set contentions to the cache set $Cache\_Set_{R_c}(\vec{\imath})$. It is possible that $Mem\_Line_{R_c}(\vec{\imath}) = Mem\_Line_{R_i}(\vec{\jmath_{k'}})$, where $1 \leqslant k' \leqslant k$.

We use the technique presented in [11] to solve these equations to find the replacement miss points for a k-way set associative cache with a capacity of $\mathcal{C}$ bytes and a cache line size of $\mathcal{L}$ bytes. The basic idea is to replace the fourth line of (2) by:

$$Mem\_Addr_{R_c}(\vec{\imath}) \quad = \quad Mem\_Addr_{R_i}(\vec{\jmath}) + n\mathcal{C}/k + b$$

where n is any nonzero integer and $L_{off} \leqslant b \leqslant \mathcal{L} - 1 - L_{off}$ such that $L_{off} = Mem\_Addr_{R_i}(\vec{\jmath}) \bmod \mathcal{L}$. Let S be set of solutions of the form $(\vec{\imath}, \vec{\jmath}, n)$ to the replacement miss equations of R_c along $\vec{r}$. Let $S' = \{(\vec{\imath}, n) \mid (\vec{\imath}, \vec{\jmath}, n) \in S\}$. Then, R_c suffers a replacement miss at $\vec{\imath}$ along $\vec{r}$ if S' contains at least k distinct $(\vec{\imath}, n_1), (\vec{\imath}, n_2), \ldots, (\vec{\imath}, n_k)$, which represent k distinct memory accesses via R_i to k distinct memory lines all mapped to the same cache set.

2.4 Finding Cache Misses from the Miss Equations

One advantage of our miss equations is that the cache misses for different references can be analysed independently and the cache misses for different iteration points of the same RIS can also be analysed independently. In Section 2.3, we have presented the miss equations for a single reuse vector of a reference. To find precisely the cache misses of a reference, its multiple reuse vectors must be considered at once. We have employed two algorithms (given in Figure 7) in our experiments in finding the cache misses from the miss equations. *FindMisses* analyses all points in all RISs and is practical only for loop nests of small problem sizes. *EstimateMisses* analyses a sample for every RIS and is capable of analysing any program with a good degree of accuracy.

FindMisses finds the cache misses of a reference by considering its reuse vectors in lexicographically increasing order $\prec$. The solutions to the cold miss equations of R along the present reuse vector $\vec{r}$ are indeterminate and need to be examined further using the other reuse vectors of the reference. All the other points can be classified into either hits and misses using the replacement miss equations of R along $\vec{r}$. Once all reuse vectors are exhausted, the points that remain indeterminate are cold misses for the reference R being analysed. The miss ratio for a reference and that for the loop nest are calculated in the normal manner.

```
0   Algorithm MissAnalyser
1   for each reference R (in no particular order)
2       Sort its reuse vectors in lexicographically increasing order $\prec$
3       $H_R = \emptyset$          // set of hits for R
4       $RM_R = \emptyset$         // set of replacement misses for R
5       $CM_R = \mathcal{S}(R)$ // set of cold misses for R initially
6       for each reuse vector $\vec{r}$ of R in the sorted list (given in line 2)
7           $CM'_R$ = set of solutions of R's cold miss equations along $\vec{r}$
                    i.e., set of solutions of to (1) with $R_c = R$ and
                    $RIS_{R_c} = CM_R$ and $R_p$ uniquely determined by $R_c$ and $\vec{r}$
8           for each $\vec{\imath} \in (CM_R - CM'_R)$
9               if $\vec{\imath}$ is a hit according to R's replacement miss equations along $\vec{r}$
10                  $H_R = H_R \cup \{\vec{\imath}\}$
11              else
12                  $RM_R = RM_R \cup \{\vec{\imath}\}$
13          $CM_R = CM'_R$
14      $Miss\_Ratio(R) = \frac{|CM_R|+|RM_R|}{|\mathcal{S}(R)|}$
15  $Loop\_Nest\_Miss\_Ratio = \frac{\sum_R |RIS_R| \times Miss\_Ratio(R)}{\sum_R |RIS_R|}$

16  Algorithm FindMisses
17  for each reference R (in no particular order)
18      $\mathcal{S}(R) = RIS_R$ (i.e., R's RIS) // analyse all points
19  MissAnalyser

20  Algorithm EstimateMisses
21  c is the confidence percentage from the user
22  w is the confidence interval from the user
23  for each reference R (in no particular order)
24      compute the volume of $RIS_R$
25      if $RIS_R$ is too small to achieve $(c, w)$
26          if $RIS_R$ is large enough to achieve the default $(c', w') = (90\%, 0.15)$
27              $\mathcal{S}(R)$ = a sample of $RIS_R$ according to $(c', w')$
28          else
29              $\mathcal{S}(R) = RIS_R$ // analyse all points
30      else
31          $\mathcal{S}(R)$ = a sample of $RIS_R$ according to $(c, w)$
32  MissAnalyser
```

Figure 7: Two algorithms for computing the cache misses from miss equations.

Since all points in a RIS are analysed, *FindMisses* works as long as all IF conditionals can be evaluated at every iteration point at compile time. These *compile-time-analysable conditionals* include all expressions involving loop indices and compile-time constants only.

In lines 9 – 12 of *MissAnalyser*, every point examined is not a solution to the cold miss equations (1). Thus, the replacement miss equations (2) can be simplified to:

$$\begin{gathered} Cache\_Set_{R_c}(\vec{\imath}) = Cache\_Set_{R_i}(\vec{\jmath}) \\ \vec{\jmath} \in J_{R_i} \end{gathered}$$

EstimateMisses operates in exactly the same way as *FindMisses* except that a sample from every RIS is analysed. This allows us to analyse programs of large problem sizes effectively and efficiently. The technical details for the statistical sampling technique used in this work can be found in [24]. However, we have made some modifications to cope with references with different RISs (Figure 2) and references with non-convex RISs (Figure 3).

EstimateMisses expects the user to enter values to the two parameters: the *confidence percentage* c and the *confidence width* w*, where* $0\% < c \leqslant 100\%$ *and* $0 < w < 1$ [24]. The two input values determine the size of the sample taken from RIS_R and also impose a lower bound on $|RIS_R|$. If a RIS is too small to achieve (c, w), we either use the default values $(c', w') = (90\%, 0.15)$ (which requires a sample size of 72 points and $|RIS_R| \geqslant 1440$ [24]) or analyse all points in RIS_S (when $|RIS_R| < 1440$). The meanings of c and w are such that if we run *EstimateMisses* many times, the real miss ratio for each R obtained in c of these runs will lie in the interval $[Miss\_Ratio(R) - w/2, Miss\_Ratio(R) + w/2]$. However, this interpretation does not apply to the miss ratio for the loop nest given in line 15. Fortunately, our results are always close to those obtained by simulation.

Thus, the statistical sampling technique used requires the size of every RIS to be calculated. If the IF conditions guarding a reference form a union of convex polyhedra, then the corresponding RIS is a union of convex polyhedra because the iteration space is convex. The number of points contained in such a RIS is calculated by slicing the RIS recursively into regions of lower and lower dimensions until eventually every region is either empty or a (one-dimensional) union of line segments so that the points in the region can be counted easily. This algorithm, while exponential in terms of the dimension of the iteration space, is very efficient for practical programs with simple loop bounds and affine conditionals. Other methods for computing the volume of a convex polytope also exist [5, 13, 18].

If a reference R is guarded by some non-affine conditionals, then RIS_R can be arbitrarily complex. There is not any general method for computing the volume of RIS_R. In our implementation, we compute the volume of such a RIS by proceeding as before with all non-affine conditionals ignored and then count only those points that satisfy all non-affine conditionals. This simple extension has not been used in our experiments since we have not found any data-independent conditionals that are not affine in all programs analysed.

3 Analysing Imperfectly Nested Loops

This section presents a strategy to analyse an important class of imperfectly nested loops. We start with an imperfect loop nest of the form shown in Figure 8(a) and transform it by loop sinking to obtain a perfect loop nest as shown in Figure 8(b). The necessary and sufficient conditions for the legality of loop sinking can be found in [28]. Informally, loop sinking must ensure that if an iteration of a statement would have executed in the original program, then it is executed in the transformed program.

A perfect loop nest is considered *non-analysable* when (a) it has a function call, (b) it has a return statement, and (c) it has a non-affine loop bound or a non-constant loop stride.

Loop sinking enables many important imperfect loop nests to be analysed now. Table 2 shows the coverage of our method for a collection of benchmark programs. For each program, the table summarises the number of perfect loop nests analysable previously [11, 24], the number of imperfect loop nests both sinkable and analysable now, and the relative percentage

```
DO $I_1 = L_1, U_1$
  $S_1$
  DO $I_2 = L_2, U_2$
    $S_2$
    ...
      DO $I_n = L_n, U_n$
        $ST_n$
      ENDDO
    ...
    $T_2$
  ENDDO
  $T_1$
ENDDO
```

(a) Original loop nest

```
DO $I_1 = L_1, U_1$
    DO $I_2 = L_2, U_2$
    ...
      DO $I_n = L_n, U_n$
        IF ($I_2$.EQ.$L_2$ .AND. $\cdots$ .AND. $I_n$.EQ.$L_n$) THEN S1 ENDIF
        IF ($I_3$.EQ.$L_3$ .AND. $\cdots$ .AND. $I_n$.EQ.$L_n$) THEN S2 ENDIF
        ...
        $ST_n$
        ...
        IF ($I_3$.EQ.$U_3$ .AND. $\cdots$ .AND. $I_n$.EQ.$U_n$) THEN T2 ENDIF
        IF ($I_2$.EQ.$U_2$ .AND. $\cdots$ .AND. $I_n$.EQ.$U_n$) THEN T1 ENDIF
      ENDDO
    ...
    ENDDO
ENDDO
```

(b) Transformed loop nest

Figure 8: Conversion of imperfect to perfect loop nests by loop sinking

Benchmark	Program	Analysable Before	Sinkable & Analysable	Increase
SPECfp95	Tomcatv	2	0	0.00%
	Swim	16	0	0.00%
	Su2cor	33	5	15.15%
	Hydro2D	81	2	2.47%
	Mgrid	10	1	10.00%
	Applu	18	2	11.11%
	Apsi	72	19	26.39%
	Turb3D	19	10	52.63%
	Fppp	12	0	0.0%
	Wave	141	40	28.37%
PERFECT	CSS	45	4	8.89%
	LGSI	64	0	0.00%
	LWSI	11	7	63.64%
	MTSI	30	1	3.33%
	NASI	105	12	11.43%
	OCSI	40	11	27.50%
	SDSI	52	17	32.59%
	SMSI	46	29	63.04%
	SRSI	105	15	14.29%
	TFSI	56	7	12.50%
	WSSI	98	33	33.67%
Livermore	Kernels	12	4	33.33%
Linpack	Kernels	21	0	0.00%
Lapack	Kernels	443	43	9.71%
TOTAL		1532	262	17.10%

Table 2: Analysable loop nests

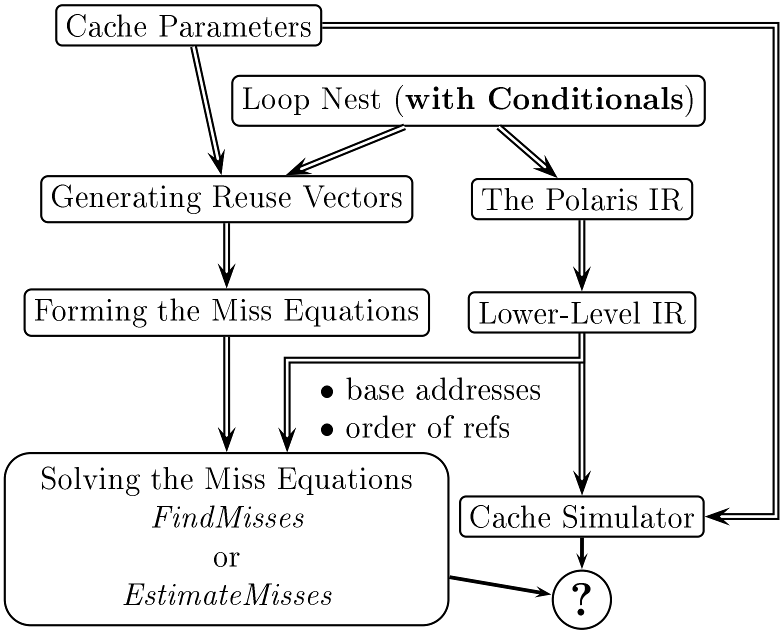


Figure 9: A framework for cache miss analysis and evaluation.

increase. An imperfect loop nest that is sinkable but non-analysable is not included in our loop statistics. The number of imperfect loop nests that are sinkable and analysable in these benchmarks is quite large. We can analyse 262 more loop nests, which is 17.10% more than what can be analysed previously. For programs such as Turb3D, SMSI and LWSI, the improvements are impressive reaching 52.63%, 63.04% and 63.34%, respectively.

When collecting the above loop statistics, we find that the number of loop nests with affine conditionals is quite small. This is not surprising since such a loop nest would have been written as an imperfect loop nest in the first place! However, there are a large number of loop nests (about 277) with data-dependent conditionals in the above benchmarks analysed. Their successful analysis will be an interesting future research topic.

4 Experiments

Figure 9 depicts the framework used in finding cache misses from the miss equations and for validating the accuracy of our method against a cache simulator. We have implemented our method in the Coyote Miss Equation solver [23]. The required reuse vectors for a reference are calculated using some libraries provided in Coyote. The miss equations for a reference are generated as discussed in Section 2.3. We have written a program to obtain the base addresses and the relative access order of references from a load-store lower-level IR, which is produced from the Polaris IR [7] of the loop nest being analysed. The same information obtained is fed to both our miss equation solvers and the cache simulator used.

We have analysed a range of programs from SPECfp95, Perfect Suite, Livermore Kernels, Linpack and Lapack. We report our experimental results for the following four examples:

Program	Cache	#Cache Misses		%Loop Nest Miss Ratio		Error	Execution Time (secs)
		Simulator	*FindMisses*	Simulator	*FindMisses*		
COND	direct	1164004	1164004	81.69	81.69	0.0	55.62
	2-way	1157335	1157335	81.22	81.22	0.0	105.00
	4-way	1157335	1157335	81.22	81.22	0.0	180.64
LU	direct	81440	85193	6.13	6.41	0.28	67.09
	2-way	57441	70643	4.32	5.31	0.99	71.04
	4-way	61278	77461	4.61	5.83	1.22	77.83
MM	direct	287697	287700	7.17	7.17	0.0	55.24
	2-way	262699	262702	6.55	6.55	0.0	59.31
	4-way	262699	262702	6.55	6.55	0.0	65.01
LWSI	direct	802	816	0.16	0.16	0.0	1.50
	2-way	802	816	0.16	0.16	0.0	14.36
	4-way	802	816	0.16	0.16	0.0	30.69

Table 3: Cache miss ratios for caches with $\mathcal{C}$=32KB and $\mathcal{L}$=32B and execution times of *FindMisses* on a 933MHz Pentium III PC running on SunOS 5.6.

- COND: our running example (Figure 1).
- LU: LU decomposition without pivoting from Lapack (Figure 10).
- LWSI: a four-dimensional imperfect loop nest from LWSI (Figure 10).
- MM: matrix multiplication from Livermore kernels (Figure 10).

The problem sizes used for the four examples are those as specified in the programs.

We assume a cache of $\mathcal{C}$=32KB with $\mathcal{L} = 32$ bytes per cache line. In all four examples, the size of each array element is 8 bytes. Therefore, every cache line has four array elements.

The execution times of *FindMisses* and *EstimateMisses* are obtained on a 933MHz Pentium III PC.

All simulation results are obtained using a trace-driven simulator.

4.1 *FindMisses*

This algorithm finds the cache misses from the miss equations by analysing all iteration points (i.e., all memory accesses) in the loop nest. It is computationally expensive for large iteration spaces since it performs essentially a compile-time cache simulation of the loop nest. However, this algorithm can be used ideally to evaluate the accuracy of our method, in particular, our reuse vector analysis. Table 3 compares *FindMisses* and a cache simulator for caches of different associativities. The absolute error between the miss ratios in both cases in all examples is negligible. The execution times in all cases indicate that analysing all points is too expensive to be used at compile-time in guiding compiler optimisations.

Some further discussions about the four examples are provided below.

COND Both *FindMisses* and the simulator yield the same results in all cache configurations.

```
PROGRAM LU
PARAMETER (N = 100)
REAL*8 a(N,N)
DO i = 1,N
  DO j = i+1,N
    a(j,i) = a(j,i)/a(i,i)
    DO k = i+1,N
      a(j,k) = a(j,k)-a(j,i)*a(i,k)
    ENDDO
  ENDDO
ENDDO
END
```

```
...
DO i = 1,N
  DO j = i+1,N
    DO k = i+1,N
      IF (k .EQ. i+1) THEN
        a(j,i) = a(j,i)/a(i,i)
      ENDIF
      a(j,k) = a(j,k)-a(j,i)*a(i,k)
    ENDDO
  ENDDO
ENDDO
END
```

```
PROGRAM MM
PARAMETER (N=100)
REAL*8 a(N,N), b(N,N), c(N,N)
DO i = 1,N
  DO j = 1,N
    a(i,j) = 0
    DO k = 1,N
      a(i,j) = a(i,j)+b(i,k)*c(k,j)
    ENDDO
  ENDDO
ENDDO
END
```

```
...
DO i = 1,N
  DO j = 1,N
    DO k = 1,N
      IF (k.EQ.1) THEN
        a(i,j) = 0
      ENDIF
      a(i,j) = a(i,j)+b(i,k)*c(k,j)
    ENDDO
  ENDDO
ENDDO
END
```

```
PROGRAM LWSI
PARAMETER (NS = 10, natoms = 100)
DOUBLE PRECISION xt, yt, xc, yc, zc
DOUBLE PRECISION zero, wsin, wcos, z, xs
DIMENSION xc(natoms, ns), yc(natoms, ns)
DIMENSION zc (natoms, ns), xt (natoms)
DIMENSION wsin(1), wcos(1), zero(1), z(1)
DIMENSION xs(1), yt (natoms)
DO i = 1, ns, 1
 xt(1) = xt(2)+wcos(1)
 xt(3) = xt(1)
 yt(2) = zero(1)
 DO j = 1, ns, 1
  yt(1) = yt(2)+wsin(1)
  yt(3) = yt(2)-wsin(1)
  z(1) = zero(1)
  DO k = 1, ns, 1
   DO l = 1, natoms, 1
    xc(l,k) = xt(l)
    yc(l,k) = yt(l)
    zc(l,k) = z(1)
   ENDDO
   z(1) = z(1)+xs(1)
  ENDDO
  yt(2) = yt(2)+xs(1)
 ENDDO
 xt(2) = xt(2)+xs(1)
ENDDO
END
```

```
...
DO i = 1, ns, 1
 DO j = 1, ns, 1
  DO k = 1, ns, 1
   DO l = 1, natoms, 1
    IF (j.EQ.1 .AND. k.EQ.1
     .AND. l.EQ.1) THEN
     xt(1) = xt(2)+wcos(1)
     xt(3) = xt(1)
     yt(2) = zero(1)
    ENDIF
    IF (k.EQ.1 .AND. l.EQ.1) THEN
     yt(1) = yt(2)+wsin(1)
     yt(3) = yt(2)-wsin(1)
     z(1) = zero(1)
    ENDIF
    xc(l,k) = xt(l)
    yc(l,k) = yt(l)
    zc(l,k) = z(1)
    IF (l.EQ.natoms) THEN
     z(1) = z(1)+xs(1)
    ENDIF
    IF (k.EQ.ns .AND. l.EQ.natoms) THEN
     yt(2) = yt(2)+xs(1)
    ENDIF
    IF (j.EQ.ns .AND. k.EQ.ns
     .AND. l.EQ.natoms) THEN
     xt(2) = xt(2)+xs(1)
    ENDIF
   ENDDO
  ENDDO
 ENDDO
ENDDO
END
```

Figure 10: Three examples (with original and transformed programs).

LU *FindMisses* over-estimates the cache misses in all cache configurations used. The mis-predictions are due to the lack of reuse vectors to describe the reuse that exists among the non-uniformly generated references: *a(j,i)*, *a(i,i)*, *a(j,k)* and *a(i,k)*. For example, *a(i,i)* accesses *a(1,1)* and *a(j,i)* accesses *a(2,1)* at the same iteration $(1, 1, 2)$. Both accesses are to the same cache line. The lack of a reuse vector to describe this particular reuse results in the memory access $a(1, 1)$ to be classified incorrectly as a miss. To validate this assumption, we ran *FindMisses* by adding four additional group-spatial reuse vectors: $(0, 0, 0)$ from $a(j, i)$ to $a(i, i)$, $(0, 1, 0)$ from $a(i, i)$ to $a(j, i)$, $(0, 0, 0)$ from $a(j, k)$ to $a(i, k)$ and $(0, 1, 0)$ from $a(i, k)$ to $a(j, k)$. The cache misses obtained for the "direct", "2-way" and "4-way" cases have been reduced to 81553, 64704 and 71200, respectively. As a result, the absolute errors in these cases have been reduced to 0.00, 0.55 and 0.75, respectively.

MM *FindMisses* over-estimates the number of misses in all three cases by a margin of three. The three mis-predictions are due to the lack of reuse vectors to describe the spatial reuse between references *b(i,k)* and *c(k,j)*. The base addresses for b and c are 230136 and 310136, respectively. Thus, the memory addresses of $b(98, 100)$, $b(99, 100)$, $b(100, 100)$ and $c(1, 1)$ are 310112, 310120, 310128 and 310136, respectively. This implies that all four elements reside in the same memory line (starting at 475). A simple analysis shows that the access $b(i, 100)$ at iteration $(i, 1, 100)$ reuses this memory line brought into the cache by the access $c(1, 1)$ at iteration $(i, 1, 1)$, where $98 \leqslant i \leqslant 100$. Due to the lack of reuse vectors, these three accesses to b are classified as misses.

LWSI The transformed program by loop sinking consists of five conditionals some of which are quite complex. In our experiments, the five scalars ($zero, wsin, wcos, z$ and xs) are treated as one-dimensional arrays of single elements each, which happen to reside in four different memory lines with other array variables. *FindMisses* over-estimates the cache misses by 14 in all three cases due to the lack of reuse vectors to describe the reuse among all these memory lines.

4.2 *EstimateMisses*

This algorithm finds cache misses from the miss equations of a reference by taking a sample from its RIS. We have modified the statistical sampling technique in [24] so that we can cope with references with different RISs and references whose RISs are non-convex.

Table 4 shows the accuracy and efficiency of *EstimateMisses* using a 95% confidence percentage with an interval width of 0.05. In all but one case, the difference between the estimated miss ratio and the real miss ratio is less than 1.0. The difference in the exceptional 4-way LU case is 1.12. This is due to the lack of reuse vectors for describing the reuse among the non-uniformly generated references as discussed previously. To validate this assumption, we ran *EstimateMisses* by adding the same four additional group-spatial reuse vectors as before: $(0, 0, 0)$ from $a(j, i)$ to $a(i, i)$, $(0, 1, 0)$ from $a(i, i)$ to $a(j, i)$, $(0, 0, 0)$ from $a(j, k)$ to $a(i, k)$ and $(0, 1, 0)$ from $a(i, k)$ to $a(j, k)$. The miss ratios for the loop nest obtained for the "direct", "2-way" and "4-way" cases have been reduced to 6.35, 4.85 and 5.42, respectively.

Program	Cache	%Loop Nest Miss Ratio		Error	Execution Time (secs)
		Simulator	*EstimateMisses*		
COND	direct	81.69	81.29	0.40	0.40
	2-way	81.22	80.92	0.70	0.64
	4-way	81.22	80.92	0.70	0.97
LU	direct	6.13	6.49	0.36	0.68
	2-way	4.32	5.18	0.86	0.70
	4-way	4.61	5.73	1.12	0.69
MM	direct	7.17	7.18	0.01	0.12
	2-way	6.55	6.44	0.11	0.11
	4-way	6.55	6.44	0.11	0.13
LWSI	direct	0.16	0.15	0.01	0.35
	2-way	0.16	0.15	0.01	0.50
	4-way	0.16	0.15	0.01	0.65

Table 4: Cache miss ratios for caches with $\mathcal{C}$=32KB and $\mathcal{L}$=32B and execution times of *EstimateMisses* on a 933MHz Pentium III PC running on SunOS 5.6 ($c = 95\%$ and $w = 0.05$).

As a result, the absolute errors in these cases have been reduced to 0.22, 0.53 and 0.81, respectively.

The execution times in all cases are less than a second on a 933MHz Pentium III PC.

A version of Table 4 for larger problem sizes is not given because (a) the results for *EstimateMisses* are similar since samples of similar sizes will be analysed and (b) many hours of cache simulation will have to be consumed.

5 Related Work

Programs must exhibit sufficient locality to achieve good cache performance. Compiler optimisations for improving the cache behaviour need to have detailed knowledge about the number and causes of cache misses. Such an information can be obtained by time-consuming cache simulation [22] and architecture-dependent hardware counters [1].

Analytical methods use mathematical formulas to provide a characterisation of a program's cache behaviour so that we can not only obtain the number of cache misses but also reason about the causes of such misses from these formulas. The ultimate goal is to develop an analytical method that can provide accurate assessments of when and why cache misses occur using a reasonable amount of computational resources (e.g., CPU time, memory and disk usage). Then such a method will be useful in guiding various automatic memory optimisations and also in improving the simulation times of cache simulators and profilers.

Porterfield [17] introduces the concept of overflow iteration for predicting the miss ratio for a fully set associative LRU cache. Ferrante, Sarkar and Thrash [8] provide closed-form formulas to estimate the capacity misses of a loop nest. Temam, Fricker and Jalby [21] also consider conflict misses but for a subset of array references studied in this paper. Wolf and Lam [27] propose to use vectors to describe data reuse for uniformly generated references in a perfect loop nest. They also use reuse vectors to derive an estimate of cache misses to guide their data locality algorithm. Gannon, Jalby and Gallivan [10] and Wolfe [26] discuss the use of reference window for predicting cache misses.

The CMEs [11, 12] represent a more ambitious analytical method in an attempt to provide a more accurate analysis of cache misses. This framework is targeted at perfectly nested loops with affine loop bounds and data accesses. If all reuse vectors of a reference are used, all cache misses for the reference can be found from the CMEs provided all the points in the reference's RIS are analysed. Unfortunately, analysing all points this way is expensive as shown in Table 3. An efficient implementation of the CME framework based on polyhedral theory and statistical sampling techniques is reported in [3, 24, 25]. In principle, programs of arbitrary problem sizes can be analysed efficiently. The estimated miss ratio is known to fall within a confidence interval with a confidence percentage.

Recognising that the CMEs are expensive to solve if all iteration points in all RISs are to be analysed, Fraguela, Doallo and Zapata [9] rely on a probabilistic analytical method instead. They assume implicitly that a loop nest is free of IF statements. While they have applied their method to some imperfect loop nests, the pair of references generating the reuse must still be confined within a single perfect loop nest. Their experimental results indicate that their method can achieve a good degree of accuracy in estimating cache misses. Unlike the CMEs, this probabilistic method fails to characterise precisely all cache misses in a program. It is unclear how the causes of cache misses can be deducted from their method.

There has been a great deal of research on applying loop and data transformations to improve the cache performance of loop-oriented codes [11, 15, 16, 19, 20, 26, 27]. In particular, researchers have explored the use of various compiler heuristics and simple cache cost models to choose appropriate tile sizes in the case of loop tiling [4, 6, 14, 27] and appropriate padding amounts in the case of data padding [15, 20]. Analytical methods promise to provide more accurate knowledge about cache misses to guide a range of compiler optimisations. The CMEs [11] are limited to perfectly nested loops only, which must be free of IF statements. This paper presents an analytical method for analysing perfect loop nests with compile-time-analysable IF conditionals.

6 Conclusion

We have presented an analytical method for analysing the cache behaviour of perfectly nested loops containing IF statements with compile-time-analysable conditionals. In the presence of these conditionals, different references may be executed in different parts of iteration spaces, which are not necessarily convex. We described how reuse vectors are calculated and how the miss equations are formed and solved. We have presented two algorithms for finding the cache misses from these miss equations. *FindMisses*, which analyses all points in a reference iteration space, is applicable to programs of small problem sizes. In addition, this algorithm has been used to evaluate the accuracy of our analytical framework. *EstimateMisses* analyses a sample of a reference iteration space and achieves close to real cache miss ratio in practical cases efficiently. We have done extensive experiments over a range of programs. Our experimental results show that our method, together with loop sinking, can be used to analyse 17% more loop nests in SPECfp95, Perfect Suite, Livermore kernels, Linpack and Lapack than previously [11, 24].

While this work represents an important step towards a mechanical analysis of complex program constructs, there are several important constructs that are still non-analysable, including (a) imperfect loop nests with several loops at the same level, (b) data-dependent conditionals, and (c) subroutine calls. We are presently working on developing an analytical method that aims at analysing these complex language constructs. We intend to investigate benefits and limitations of this challenging but important research direction.

7 Acknowledgements

This work has been supported by an Australian Research Council Grant A10007149.

References

[1] G. Ammons, T. Ball, and J.R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI'97)*, pages 85–96, 1997.

[2] Nerina Bermudo, Xavier Vera, Antonio González, and Josep Llosa. An efficient solver for cache miss equations. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'00)*, 2000.

[3] Nerina Bermudo, Xavier Vera, Antonio González, and Josep Llosa. Optimizing cache miss equations polyhedra. In *4th Workshop on Interaction between Compilers and Computer Architecture (Interact)*, 2000.

[4] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Supercomputing '92*, pages 114–124, Minneapolis, Minn., Nov. 1992.

[5] P. Clauss. Counting solutions to linear and non-linear constraints through Ehrhart polynomials. In *ACM International Conference on Supercomputing (ICS'96)*, pages 278–285, Philadelphia, 1996.

[6] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 279–290, June 1995.

[7] K. A. Faigin, J. P. Hoeflinger, D. A. Padua, P. M. Petersen, and S. A. Weatherford. The Polaris internal representation. *International Journal of Parallel Programming*, 22(5):553–586, Oct. 1994.

[8] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *4th Workshop on languages and compilers for parallel computing (LCPC'91)*, pages 328–343, 1991.

[9] Basilio B. Fraguela, Ramon Doallo, and Emilio L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, 1999.

[10] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[11] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[12] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Automated cache optimizations using CME driven diagnosis. In *International Conference on Supercomputing (ICS'00)*, pages 316–326, 2000.

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

[14] F. Irigoin and R. Triolet. Supernode partitioning. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, California., Jan. 1988.

[15] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *International Conference on Microprogramming and Microarchitecture*, pages 285–296, 1998.

[16] K. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, Jul. 1996.

[17] A. K. Porterfield. *Software Methods for improvement of cache performance on supercomputer applications.* PhD thesis, Department of Computer Science, Rice University, May 1989.

[18] W. Pugh. Counting solutions to Presburger formulas: how and why. In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94)*, pages 121–134, 1994.

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

[20] Gabriel Rivera and Chau-Wen Tseng. Eliminating conflict misses for high performance architectures. In *Internacional Conference on Supercomputing (ICS'98)*, 1998.

[21] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *ACM SIGMETRICS'94 Conference on Measurement and Modeling of Computer Systems*, pages 261–271, May 1994.

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

[23] Xavier Vera and Nerina Bermudo. Technical report, Mälardalens Högskola, 2001.

[24] Xavier Vera, Josep Llosa, Antonio González, and Nerina Bermudo. A fast and accurate approach to analyze cache memory behavior. In *European Conference on Parallel Computing (Europar'00)*, 2000.

[25] Xavier Vera, Josep Llosa, Antonio González, and Carlos Ciuraneta. A fast implementation of cache miss equations. In *8th International Workshop on Compilers for Parallel Computers (CPC'00)*, 2000.

[26] Michael E. Wolf. *High performance compilers for parallel computing.* Addison-Wesley, 1996.

[27] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI'91)*, pages 30–44, Toronto, Ont., Jun. 1991.

[28] J. Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Computing*, 22(12):1621–1645, 1997.

[29] J. Xue and C.-H. Huang. Reuse-driven tiling for data locality. *International Journal of Parallel Programming*, 26(6):671–696, 1998.