Efficient Compile-Time Analysis of Cache 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

Abstract. *This paper presents an analytical method for analysing efficiently the cache behaviour of perfect loop nests containing IF statements with compile-time-analysable conditionals. We discuss the derivations of reuse vectors in the presence of IF statements, present miss equations for characterising the cache behaviour of a program and give algorithms for solving these equations for cache misses. We show that our method, together with loop sinking, can be used to analyse a large number of imperfect loop nests that cannot be analysed previously — 17% more loop nests than previously in SPECfp95, Perfect Suite, Livermore kernels, Linpack and Lapack. Validation against cache simulation demonstrates the efficiency and accuracy of our method. Our method can be used to guide compiler cache optimisations and improve the performance of cache simulators and profilers.*

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. Programmers and optimising compilers often restructure a program to improve its cache behaviour. In both cases, it is necessary to have detailed knowledge about the number of cache misses and their causes in the program.

While a large number of locality enhancement transformations exist [31], the models used for evaluating their benefits are often heuristic or approximate. For example, tiling [5, 15, 17, 23, 33, 34] and padding [16, 22] can reduce cache misses if appropriate tile sizes and pad sizes are chosen. In the case of matrix multiplication and SOR-like kernels, some heuristics-based cost models [8, 17, 22, 23, 30] can help make these choices. However, even in these simple cases, no model has emerged as a widely acceptable solution. It is well-known that the optimal tile and pad sizes are sensitive to the problem size, array base addresses and cache parameters. We need better models that can determine not only the number of cache misses but also help us understand the causes behind these misses. These models can then be employed to guide various optimisations to reduce cache misses in a systematic manner.

In the last few years, several compile-time analytical methods have been proposed to statically predict the cache behaviour of a program [4, 6, 11, 13, 14, 26]. At this early stage, all these research efforts have focused on loop-oriented programs operating on arrays. Such a method consists of (a) a procedure for setting up mathematical formulas to characterise the cache misses in a program and (b) an algorithm for finding cache misses (and their causes, if required) from these formulas. These formulas describe the relationships among loop indices, array sizes, base addresses and the cache parameters for cache misses in the program.

The Cache Miss Equations (CMEs) [13] make use of Wolf and Lam's reuse vectors [30] to characterise the cache misses in a program using a set of Diophantine equations (consisting of actually equalities and inequalities). This seminal work demonstrates the possibility of choosing desirable tile and pad sizes by reasoning about these equations rather than solving them for cache misses. However, computing the *exact* number of cache misses from the CMEs when required is expensive. Some statistics-based methods have been reported to produce efficiently a reasonable estimate of such misses [4, 14, 26] from the CMEs. The CMEs are limited to isolated perfect loop nests consisting of straight-line assignments. Recently, an attempt for *exactly* modelling the cache behaviour of loop nests using Presburger formulas is made in [6]. While the cache misses for both perfect and imperfect loop nests containing possibly IF statements can be specified, they do not yet have a practical algorithm for finding cache misses from this specification.

This paper extends the CMEs so that perfect loop nests with IF statements can be analysed. We make the following contributions. First, we present an analytical method for analysing efficiently the cache behaviour of perfect loop nests containing IF statements with a good degree of accuracy. In particular, we describe for the first time how to quantify reuse exactly and approximately in the presence of IF statements and some complications that may arise when group reuse is approximated. While our cold miss equations are similar to those in the CMEs [13], our replacement miss equations are formulated and solved differently since the references being analysed are potentially accessed in different parts of the iteration space (referred to as RISs). Second, we give two algorithms for finding cache misses from these our miss equations and discuss our prototyping implementation. *FindMisses* is exact and is useful for analysing programs of small sizes. *EstimateMisses*, based on the sampling theory [26], is capable of analysing large programs efficiently with a user-defined confidence. We have extended that theory so that the sampling technique works when the RISs for different references are different. We have also developed an algorithm for computing the volume of any possible non-convex RIS for sampling purposes. Third, we demonstrate how our method can also be used to analyse those imperfect loop nests that are sinkable by loop sinking. Finally, we have analysed the loop nests from SPECfp95, Perfect Suite, Livermore kernels, Linpack and Lapack. Our method enables 17% more loop nests in these benchmarks to be analysed than previously. We present our experimental results for some kernel loop nests to validate the efficiency and accuracy of our method. This work represents a useful step towards a mechanical analysis of complex language constructs.

The rest of this paper is organised as follows. Section 2 defines the cache architectures used. Section 3 describes our loop nest model. Section 4 discusses the derivation of reuse vectors when IF statements are present. Section 5 presents our analytical method. Section 6 applies our method to analyse imperfect loop nests. Section 7 presents experimental results. Section 8 discusses the related work. Section 9 concludes the paper and discusses some future work.

2 Cache Model

We assume a uniprocessor with a k-way set-associative cache using LRU replacement. In the case of write misses, we assume a fetch-on-write policy so that writes and reads are modelled identically. This model is used in the recent analytical work [6, 11, 13]. In a k-way set-associative cache, a cache set contains

```
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.

k distinct cache lines. C_s and L_s denote the cache size and cache line size (in array elements), respectively.

A *memory line* refers to a cache-line-sized block in the memory while a cache line refers to the actual block in which a memory line is mapped. Let $Mem\_Addr_R(\vec{\imath})$ be the memory address accessed by reference R at an iteration $\vec{\imath}$ of a loop nest. Let $Mem\_Line_R(\vec{\imath})$ and $Cache\_Set_R(\vec{\imath})$ be the memory line and cache set to which $Mem_R\_Addr(\vec{\imath})$ is mapped, respectively.

3 Program Model

Programs under consideration are perfect loop nests with affine loop bounds and affine array subscript expressions. A perfect loop nest of depth n is represented as an n-dimensional convex polyhedron in $\mathbb{Z}^n$ called the *iteration space* of the nest. Every iteration (or point) in the iteration space 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 in the nest. The four lexicographic operators, $\prec$, $\preceq$, $\succ$ and $\succeq$, are used in the usual manner [31].

Our analytical method can deal with any IF conditionals involving loop indices and compile-time constants. In loop-oriented programs with regular computations, almost all data-independent conditionals are affine expressions of loop indices and compile-time constants involving possibly operators such as ABS, MOD, MAX and MIN. In all programs that we analysed from SPECfp95, Perfect Suite, Livermore Kernels, Linpack and Lapack, we have not found any single IF conditional that is data-independent but not also affine.

We define the *reference iteration space (RIS)* of a reference as the set of iteration points where the reference is accessed. While we can analyse complex IF conditionals (resulting in non-convex RISs), the RISs in practical programs are found to be simple. In our

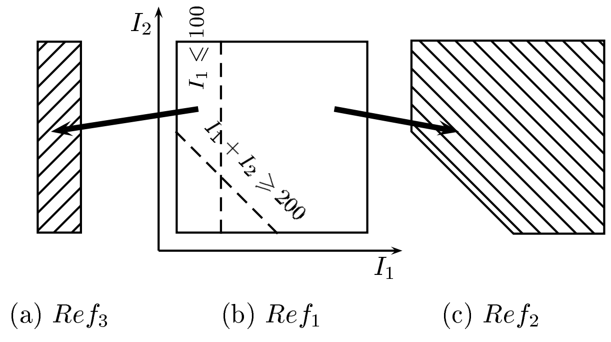


Figure 2. RISs of the z references in Figure 1.

running example given in Figure 1, the RISs for the three z references, as shown in Figure 2, are all convex. Note that the RIS for Ref_1 is the entire iteration space.

4 Reuse Vectors

Like the CMEs [13], our analytical method relies on reuse vectors to quantify the data reuse and specify the cache misses in a loop nest. To make this paper as self-contained as possible, the concept of reuse vectors is introduced and some previous work on computing reuse vectors recalled. Then we describe our formulas for deriving the required reuse vectors in the presence of IF statements. Solving these formulas exactly is possible but expensive (and unnecessary). We describe how to find an accurate approximation efficiently and provide our justifications based on an analysis of these formulas and some array reference statistics in benchmarks.

The concept of reuse vectors was introduced by Wolf and Lam in [30] as a mathematical representation to determine the direction and distance of data reuse between uniformly generated references. Let R_p (p for 'producer') and R_c (c for 'consumer') be two uniformly generated references $A(H\vec{\imath} + \vec{c}_p)$ and $A(H\vec{\imath} + \vec{c}_c)$, respectively.[1] Let $\vec{r} \succeq \vec{0}$ be an integer vector. R_c at iteration $\vec{\imath}$ (with the memory access $A(H\vec{\imath}+\vec{c}_c)$) reuses potentially from R_p at $\vec{\imath} - \vec{r}$ (with the memory access $A(H(\vec{\imath} - \vec{r}) + \vec{c}_c)$) if

$$Mem\_Line_{R_c}(\vec{\imath}) \quad = \quad Mem\_Line_{R_p}(\vec{\imath} - \vec{r})$$

Then $\vec{r}$ is said to be a *reuse vector.* It represents a potential reuse in the cache between the two memory accesses since the memory line touched in the cache at the first access (at $\vec{\imath} - \vec{r}$) may have been evicted from the cache before it gets reused at the second access (at $\vec{\imath}$). As is customary, $\vec{r}$ is *temporal* (reusing the same element) if the following additional equality also holds:

$$Mem\_Addr_{R_c}(\vec{\imath}) \quad = \quad Mem\_Addr_{R_p}(\vec{\imath} - \vec{r})$$

and *spatial* (reusing the same cache line but not the same element) otherwise. In addition, the reuse is said to be a *self reuse* if R_c and R_p are identical and a *group reuse* otherwise. Thus, there are four kinds of reuse (vectors): self-temporal, group-temporal, self-spatial and group-spatial (vectors).

Wolf and Lam [30] discuss how to compute reuse vectors for perfect loop nests consisting of straight-line assignments, assuming effectively that all RISs are the entire iteration space. By quantifying the reuse of a loop nest using a vector space spanned by (elementary) reuse vectors, they apply unimodular and tiling transformations to improve parallelism and locality in the nest. Later, Xue and Huang [34] describe an extension to allow non-elementary reuse vectors to be represented exactly. The CMEs [13] make use of reuse vectors to specify the cache misses in a loop nest consisting of straight-line assignments. Note that all arrays in FORTRAN are column-major. If the columns of every array are aligned at the memory line boundaries, Wolf and Lam's reuse framework provides all reuse vectors required. Otherwise, some extra reuse vectors are needed to represent cross-column reuse cases. Consider Figure 1, where z is a 2-D array of size $(N+1)\times(M+1)$. Suppose that a cache line has four array elements and that $z(N-1,1)$, $z(N,1)$, $z(N+1,1)$ and $z(1,2)$ reside in a common memory line in that order. For Ref_3, i.e., $z(I_1, I_2)$, the access $z(N-1,1)$ at iteration $(N-1,1)$ may potentially reuse this memory line in the cache touched by the access $z(1,2)$ at the earlier iteration $(1,2)$. This reuse is described by the self-spatial reuse vector $(N-1,1)-(1,2) = (N-2,-1)$. For details on computing Wolf and Lam's reuse vectors, see [30]. For some ad hoc techniques on computing cross-column reuse vectors required by the CMEs, see [3, 13].

Self reuse vectors for a reference are computed as before except that its RIS may be a subset of the iteration space. Section 4.1 describes below how to compute group-temporal reuse vectors in the presence of IF conditionals and some complications that may arise. Section 4.2 does the same for group-spatial reuse vectors.

4.1 Group-Temporal Reuse

Using the same notations as before, R_c at iteration $\vec{\imath}$ reuses potentially from R_p at $\vec{\imath}-\vec{r}$ via a group-temporal reuse vector $\vec{r} \succeq \vec{0}$, which is a solution to:

$$\begin{array}{rcl} Mem\_Addr_{R_c}(\vec{\imath}) & = & Mem\_Addr_{R_p}(\vec{\imath} - \vec{r}) \\ \vec{\imath} & \in & RIS_{R_c} \\ \vec{\imath} - \vec{r} & \in & RIS_{R_p} \end{array} \qquad (1)$$

where the first equality constraint is equivalent to:

$$H\vec{r} \quad = \quad \vec{c}_p - \vec{c}_c \qquad (2)$$

[1] They are *uniformly generated* since their subscript expressions share the same linear part H [30].

The difficulty in solving (1) (exactly and efficiently) is that RIS_{R_c} and RIS_{R_p} may not be identical.

If the IF conditionals guarding R_c and R_p are affine, which is usually the case for regular computations, then RIS_{R_c} and RIS_{R_p} are representable as convex polyhedra. We can use the Omega Calculator [20] to solve these constraints exactly for $\vec{r}$. But this is expensive and unnecessary (at least for regular computations).

$RIS_{R_c} = RIS_{R_p}$?	dim(ker(H))			
	$= 0$	$= 1$	> 1	$\geqslant 1$
YES	92.66	5.21	0.46	5.67
NO	1.32	0.28	0.07	0.35
YES+NO	93.98	5.49	0.53	6.02

Table 1. A classification of producer-consumer pairs in the programs in Table 3 for group-temporal reuse. All references are first moved into the innermost loop by loop sinking (see Section 6). Each producer-consumer pair is then identified from a common perfect nest. In each category, the total number of qualifying pairs over the grand total is given.

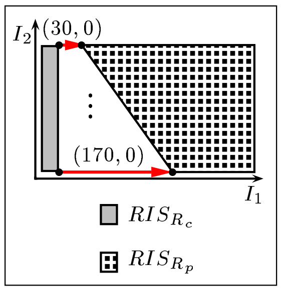
The complexity of solving (1) and (2) depends on two factors: (a) the dimensionality of ker(H) (i.e., the kernel of H) and (b) whether $RIS_{R_c} = RIS_{R_p}$ holds or not. By dividing all producer-consumer pairs from the benchmark programs listed in Table 3 into categories, Table 1 presents the percentage of the total number of qualifying pairs in each category over the grand total.

The reuse vectors are derived from (1) as follows. If dim(ker(H)) $= 0$, satisfied by 93.98% pairs as shown in Column 1, (2) has either a single solution or zero solutions. The solution thus found (if any) is taken as the solution to (1) with its last two constraints on RIS_{R_c} and RIS_{R_p} dropped. Any resulting superfluous reuse vectors will be ignored due to the presence of the first constraint in our replacement miss equations (6). If dim(ker(H)) $\geqslant 1$, satisfied by the remaining 6.02% pairs as shown in Column 4, (2) has either infinitely many or zero solutions. There are two cases. If $RIS_{R_c} = RIS_{R_p}$, (1) is solved using the techniques developed for loop nests containing no IF statements and described in [3, 13, 30, 34]. If $RIS_{R_c} \neq RIS_{R_p}$, we replace both with a common convex superset and solve (1) approximately again using [3, 13, 30, 34]. Our experimental results show that the entire iteration space is a good choice for the common superset, which avoids us from having to calculate a superset otherwise.

Figure 3 illustrates some complications for a pathological case when RIS_{R_c} and RIS_{R_p} are approximated by a common superset. R_p at its left boundary point (I_1, I_2) may reuse R_c at its right boundary point $(30, I_2)$ along $(I_1 - 30, 0)$. Thus, the set of reuse vectors

```
DO $I_1$ = 0,400
  DO $I_2$ = 0,140
    IF ($I_1$.LE.30) THEN
      $\boxed{R_c:}$ A($I_2$)
    ENDIF
    IF ($I_1$+$I_2$.GE.200) THEN
      $\boxed{R_p:}$ A($I_2$)
    ENDIF
  ENDDO
ENDDO
```

(a) Code



(b) RISs

Figure 3. Derivation of group-temporal reuse.

is $\{(30,0),(31,0),\ldots,(170,0)\}$. If the two RISs were identical (i.e., replaced with a common superset), the single vector $(0,0)$ would describe correctly the group-temporal reuse from R_c to R_p. However, when our replacement miss equations (6) are solved, $\vec{r} = (0,0)$ will be effectively ignored since RIS_{R_c} and RIS_{R_p} do not overlap. Since not all reuse vectors are used, the number of cache misses for R_p on its left boundary may be over-estimated. For practical applications, such an over-estimation should be negligible because (a) the over-estimation occurs only on a facet of a RIS (e.g., R_p's left boundary in the example) and (b) the underlying reference may reuse on the facet via other reuse vectors. In the example, R_p may reuse from itself along the self-spatial reuse vector $(1,-1)$. Thus, only a small fraction of these boundary points are mis-predicted.

Our justifications for approximating group-temporal use when $RIS_{R_c} \neq RIS_{R_p}$ are summarised below. First, the approximation happens rarely – 0.35% pairs for a collection of benchmarks (Table 1). Second, one of the two involved RISs is often the superset of the other (e.g., a cube v.s. one of its facets). So Figure 3 is only hypothetical. Third, even for Figure 3, any over-estimation of cache misses occurs only on facets of a RIS as discussed in the preceding paragraph. Finally, the accuracy of such an approximation has been validated by extensive experiments against cache simulation.

4.2 Group-Spatial Reuse

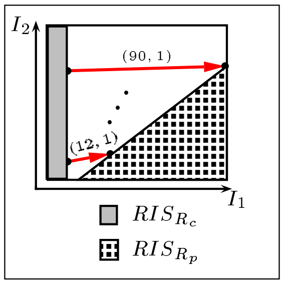
R_c at $\vec{\imath}$ reuses potentially from R_p at $\vec{\imath} - \vec{r}$, where $\vec{r} \succeq \vec{0}$ is a group-spatial reuse vector solved from:

$$\begin{array}{rcl} Mem\_Line_{R_c}(\vec{\imath}) & = & Mem\_Line_{R_p}(\vec{\imath} - \vec{r}) \\ \vec{\imath} & \in & RIS_{R_c} \\ \vec{\imath} - \vec{r} & \in & RIS_{R_p} \end{array} \quad (3)$$

Let $\vec{h}'_1$ be the first row of H and H' be the submatrix of H without its first row. Let $\vec{c}_p'$ ($\vec{c}_c'$) be obtained from $\vec{c}_p$ ($\vec{c}_c$) with its first entry removed. To find the spatial

```
DO I1 = 1,100
 DO I2 = 1,100
  IF (I1.LE.10) THEN
   Rc: A(I2, I2 + 1)
  ENDIF
  IF (I1 - I2.GE.20) THEN
   Rp: A(I2, I2)
  ENDIF
 ENDDO
ENDDO
```

(a) Code



(b) RISs

Figure 4. Derivation of group-spatial reuse.

reuse across the same array column (in FORTRAN), the first equality constraint in (3) can be rewritten to:

$$\begin{aligned} H'\vec{r} &= \vec{c}_p{}' - \vec{c}_c{}' \\ |\vec{h}_1'\vec{r}| &< L_s \end{aligned} \tag{4}$$

where L_s is the cache line size in array elements.

As in the case of group-temporal reuse, solving (3) exactly is possible but expensive. Table 2 gives the group-spatial reuse version of Table 1. If $RIS_{R_c} \neq RIS_{R_p}$, we solve (3) approximately by replacing RIS_{R_c} and RIS_{R_p} with a common superset.

$RIS_{R_c} = RIS_{R_p}$?	dim(ker(H'))			
	$= 0$	$= 1$	> 1	$\geqslant 1$
YES	0	92.66	5.67	98.33
NO	0	1.32	0.35	1.67
YES+NO	0	93.98	6.02	100

Table 2. A version of Table 3 for group-spatial reuse. Note how both tables are related.

A version of Figure 3 is given in Figure 4. By replacing RIS_{R_c} and RIS_{R_p} with a common superset, the set of group-spatial reuse vectors, $\{(12,1),(13,1),\ldots,(90,1)\}$, will be under-approximated by a subset in our implementation.

Our justifications for approximating group-temporal reuse described at the end of Section 4.1 carry over to group-spatial reuse. For the benchmark statistics given in Table 2, only 1.67% producer-consumer pairs need to be approximated. In addition, if $\dim(\ker(H)) = 1$, represented by 93.98% pairs in Table 2, the number of solutions to (4) is finite due to the inequality present in (4) unless $\vec{h}' \perp \ker(H)$ (i.e., the dot product of $\vec{h}'$ and the unique vector in $\ker(H)$ is 0). This is the case for Figure 4.

5 Analytical Method

In this section, we present the miss equations as a specification of the cache misses in a loop nest. We then discuss two algorithms for finding cache misses from these equations. In particular, our replacement miss equations are formulated and solved differently from those in the CMEs [13] since the involved RISs can be different. We also describe an algorithm for computing efficiently the volume of a RIS for sampling purposes.

5.1 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. Note that 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. Determining cache misses in this case is discussed in Section 5.2.

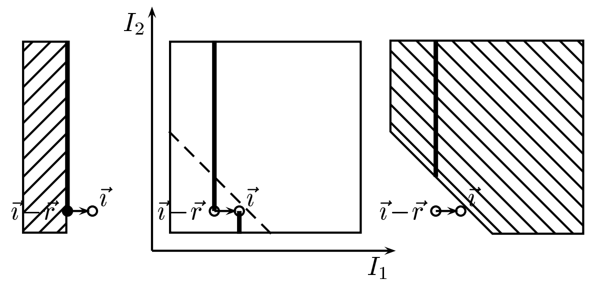
In this section, we describe the miss equations for a single reference R_c along a single reuse vector $\vec{r}$. Let R_p be the reference such that R_c reuses from R_p along $\vec{r}$. Let R_i be an intervening reference that may prevent such a reuse from being realised. Here, the subscripts c, p and 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.

5.1.1 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 cold miss at iteration $\vec{\imath}$ along $\vec{r}$ if $\vec{\imath}$ is a solution to the following equations:

$$\left.\begin{array}{c} \vec{\imath} \in RIS_{R_c} \\ \text{and} \\ (\vec{\imath} - \vec{r} \notin RIS_{R_p} \\ \text{or} \\ Mem\_Line_{R_c}(\vec{\imath}) \neq Mem\_Line_{R_p}(\vec{\imath} - \vec{r})) \end{array}\right\} \tag{5}$$

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 5. 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.

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

5.1.2 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 located between $\vec{\imath}-\vec{r}$ and $\vec{\imath}$ and 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 interference set for R_i is defined as follows:

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

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 5 shows the interference sets with the three z references when Ref_1 is analysed along its self-spatial reuse vector $\vec{r} = (1,0)$.

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 given as follows:

$$\left.\begin{array}{c} Mem\_Line_{R_c}(\vec{\imath}) = Mem\_Line_{R_p}(\vec{\imath}-\vec{r}) \\ \vec{\imath} \in RIS_{R_c} \\ \vec{\imath}-\vec{r} \in RIS_{R_p} \\ Cache\_Set_{R_c}(\vec{\imath}) = Cache\_Set_{R_i}(\vec{\jmath}) \\ \vec{\jmath} \in J_{R_i} \end{array}\right\} \qquad (6)$$

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. We use the technique presented in [13] to deal with this case.

5.2 Finding the Cache Misses

In Section 5.1, we 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. Figure 6 gives two algorithms used in our experiments for 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.

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 (5). Thus, the replacement miss equations (6) can be simplified to:

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

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

```
Algorithm MissAnalyser
for each reference R
    Sort its reuse vectors in increasing order $\prec$
    $H_R = \emptyset$          // Hits for R
    $RM_R = \emptyset$         // Replacement misses for R
    $CM_R = \mathcal{S}(R)$    // Cold misses for R initially
    for each reuse vector $\vec{r}$ of R in the sorted list
        $CM'_R$ = solutions of R's cold miss along $\vec{r}$
        for each $\vec{\imath} \in (CM_R - CM'_R)$
            if $\vec{\imath}$ is a "replacement" hit along $\vec{r}$
                $H_R = H_R \cup \{\vec{\imath}\}$
            else
                $RM_R = RM_R \cup \{\vec{\imath}\}$
        $CM_R = CM'_R$
    $Miss\_Ratio(R) = \frac{|CM_R|+|RM_R|}{|\mathcal{S}(R)|}$
$Loop\_Nest\_Miss\_Ratio = \frac{\sum_R |RIS_R| \times Miss\_Ratio(R)}{\sum_R |RIS_R|}$
```

```
Algorithm FindMisses
for each reference R (in no particular order)
    $\mathcal{S}(R) = RIS_R$ // analyse all points
MissAnalyser

Algorithm EstimateMisses
c is the confidence percentage from the user
w is the confidence interval from the user
for each reference R (in no particular order)
    compute the volume of $RIS_R$
    if $RIS_R$ is too small to achieve $(c, w)$
        if $RIS_R$ is large enough to achieve
                the default $(c', w') = (90\%, 0.15)$
            $\mathcal{S}(R)$ = a sample $(c', w')$ of $RIS_R$
        else
            $\mathcal{S}(R) = RIS_R$ // analyse all points
    else
        $\mathcal{S}(R)$ = a sample $(c, w)$ of $RIS_R$
MissAnalyser
```

Figure 6. Two algorithms for computing cache misses from cold and replacement equations.

problem sizes effectively and efficiently. The technical details for the statistical sampling technique used in this work can be found in [26]. However, we have made modifications necessary to deal with the fact that RISs can be different and possibly non-convex (in theory).

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* $0 < w < 1$ [26]. 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 $|RIS_R| \geqslant 1440$ [26]) or analyse all points in RIS_R (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 entire loop nest given in line 15. In all our experiments, real and estimated miss ratios are close.

Thus, the statistical sampling technique used requires the size of every RIS to be calculated. Our algorithm for computing the volume of an RIS is described as follows. 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 [7, 21].

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.

6 Analysing Imperfect Loop Nests

We can now analyse an important class of imperfect loop nests, i.e., those that can be made perfect by loop sinking [31]. The necessary and sufficient conditions for the legality of loop sinking can be found in [32].

A perfect loop nest is considered *non-analysable* when it has (a) a function call, (b) a return statement, (c) a non-affine loop bound or (d) a non-constant loop stride. Table 3 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 [13, 26], the number of imperfect loop nests both sinkable and analysable now and the relative percentage increase. An imperfect loop nest that is sinkable but non-analysable is not included in our statistics. The number of imperfect loop nests that are sinkable and analysable is quite large. We can analyse 262 more nests – 17.10% more than what could be analysed previously. For programs such as

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	0.00
	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 3. Analysable loop nests

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.

By applying loop sinking, we are analysing exactly the same references accessed in exactly the same order as in the original program. First, an IF conditional introduced for a reference in the transformed program serves only to identify the domain in which the reference is accessed. The loop indices involved in the IF conditional are assumed to be register-allocated. Second, loop sinking guarantees that all references in the transformed program are accessed in the same order as they are in the original program.

7 Experiments

Figure 7 depicts the framework used in finding cache misses from the miss equations and for validating the accuracy of our method against a simulator. We have implemented our method in the Coyote Miss Equations solver [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 [9] of the loop nest being analysed using Ictineo [2]. The same information obtained is fed to both our method and the simulator.

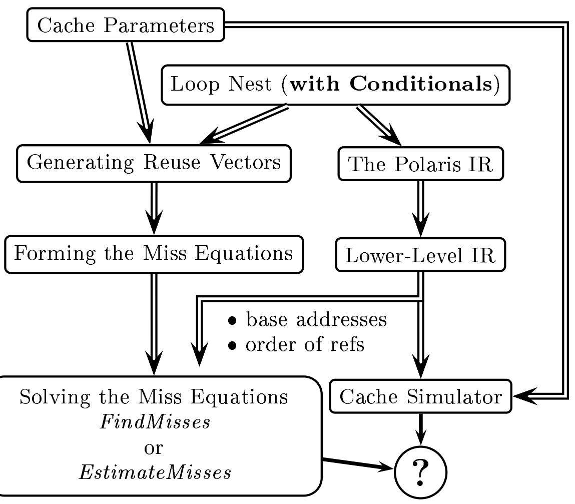


Figure 7. A framework for analysis and evaluation.

We have analysed a range of programs from SPECfp95, Perfect Suite, Livermore Kernels, Linpack and Lapack. We present our experimental results for our running example from Figure 1 and the three loop nests given in Figure 8. The problem sizes are those as given in the programs unless specified otherwise. In all cases, an array element is assumed to take 8 byes. 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.

7.1 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 4 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 are provided below. Note that COND is our running example from Figure 1 and

```
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 = 20, 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 8. Three examples (with original and transformed programs): LU (without pivoting) is taken from Lapack, LWSI is a 4-D imperfect loop nest from LWSI and MM is from Livermore kernels.

Prog.	Cache	#Cache Misses		Miss Ratio		Abs.	Exe.T (secs)	
		Sim.	*F.M.*	Sim.	*F.M.*	Err	Sim.	*F.M.*
COND	direct	1164004	1164004	81.69	81.69	0.00	0.53	55.20
	2-way	1157335	1157335	81.22	81.22	0.00	0.58	100.20
	4-way	1157335	1157335	81.22	81.22	0.00	0.58	176.54
LU	direct	81440	85193	6.13	6.41	0.28	0.32	63.09
	2-way	57441	70643	4.32	5.31	0.99	0.33	65.03
	4-way	61278	77461	4.61	5.83	1.22	0.34	67.80
MM	direct	287697	287700	7.17	7.17	0.00	1.02	55.11
	2-way	262699	262702	6.55	6.55	0.00	1.04	59.15
	4-way	262699	262702	6.55	6.55	0.00	1.10	65.28
LWSI	direct	423748	446473	10.52	11.08	0.56	1.41	76.80
	2-way	622025	645149	15.45	16.02	0.57	1.48	161.38
	4-way	600053	623578	14.90	15.48	0.58	1.52	232.41

Table 4. Cache misses for (C_s, L_s) =(32KB,32B) **and execution times for** *FindMisses* **(***F.M.***).**

LU, MM and LWSI are the kernels given in Figure 8.

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

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 $b(98,100)$, $b(99,100)$,

Prog.	Cache	Miss Ratio		Abs.	Exe.T (secs)	
		Sim.	*E.M.*	Err	Sim.	*E.M.*
COND	direct	81.69	81.29	0.40	0.53	0.26
	2-way	81.22	80.92	0.30	0.58	0.49
	4-way	81.22	80.92	0.30	0.58	0.92
LU	direct	6.13	6.49	0.36	0.32	0.20
	2-way	4.32	5.18	0.86	0.33	0.22
	4-way	4.61	5.73	1.12	0.34	0.23
MM	direct	7.17	7.18	0.01	1.02	0.12
	2-way	6.55	6.44	0.11	1.04	0.11
	4-way	6.55	6.44	0.11	1.10	0.13
LWSI	direct	10.52	10.93	0.41	1.41	0.17
	2-way	15.45	15.54	0.09	1.48	0.35
	4-way	14.90	14.93	0.03	1.52	0.40

Table 5. Miss ratios for (C_s, L_s) =(32KB,32B) and execution times of *EstimateMisses* (*E.M.*) ($c = 95\%$ and $w = 0.05$).

$b(100, 100)$ 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 a single element each, which happen to reside in four different memory lines with other array variables. *FindMisses* overestimates the cache misses by about the same margin in three cases due to the lack of reuse vectors to describe the reuse among all these memory lines.

7.2 EstimateMisses

This algorithm finds cache misses from the miss equations of a reference by taking a sample from its RIS. Table 5 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 one 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. 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.

Table 6 evaluates *EstimateMisses* further for different problem sizes on different cache configurations.

Program	Cache	Miss Ratio		Abs.	Exe.T (secs)	
		Sim.	*E.M*	Err	Sim.	*E.M.*
COND N=M=1000	$\mathcal{C}\#1$	82.42	82.22	0.20	2.19	0.60
	$\mathcal{C}\#2$	94.15	93.82	0.33	2.22	0.63
	$\mathcal{C}\#3$	31.47	31.10	0.37	2.16	0.61
LU N=1000	$\mathcal{C}\#1$	19.33	19.99	0.66	349.41	0.56
	$\mathcal{C}\#2$	44.71	44.77	0.06	387.5	1.81
	$\mathcal{C}\#3$	6.23	6.44	0.21	353.26	1.12
MM N=M=400	$\mathcal{C}\#1$	13.97	13.68	0.29	68.77	0.13
	$\mathcal{C}\#2$	50.03	50.03	0.00	74.82	0.17
	$\mathcal{C}\#3$	6.33	6.04	0.29	68.21	0.14
LWSI ns=50 natoms=1000	$\mathcal{C}\#1$	36.79	37.29	0.50	244.7	0.43
	$\mathcal{C}\#2$	78.35	78.97	0.62	262.4	0.85
	$\mathcal{C}\#3$	15.27	15.37	0.10	244.32	0.3

$\mathcal{C}\#1$: $(C_s, L_s, k) =$ (64KB, 16B, direct)
$\mathcal{C}\#2$: $(C_s, L_s, k) =$ (32KB, 8B, 2)
$\mathcal{C}\#3$: $(C_s, L_s, k) =$ (128KB, 32B, 4)

Table 6. Cache misses for three different cache configurations and execution times of *EstimateMisses (E.M.)* ($c = 95\%$ and $w = 0.05$).

8 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 [25] 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 [19] introduces the concept of overflow iteration for predicting the miss ratio for a fully set-associative LRU cache. Ferrante, Sarkar and Thrash [10] provide closed-form formulas to estimate the capacity misses of a loop nest. Temam, Fricker and Jalby

[24] also consider conflict misses but for a subset of array references studied in this paper. Wolf and Lam [30] 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. Xue and Huang [34] report an improvement. Gannon, Jalby and Gallivan [12] and Wolfe [31] discuss the use of reference window for predicting cache misses.

The CMEs [13, 14] represent a more ambitious analytical method in an attempt to provide a more accurate analysis of cache misses. This framework is targeted at perfect loop nests consisting of straight-line assignments 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 4. An efficient implementation of the CME framework based on polyhedral theory and statistical sampling techniques is reported in [26, 27]. 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.

Fraguela, Doallo and Zapata [11] rely on a probabilistic argument to analyse the same class of loop nests as the CMEs. While they have applied their method to some imperfect loop nests, the pair of references involved must be from a single perfect loop nest.

Chatterjee *et al* [6] present a method for *exactly* modelling the cache behaviour of loop nests. They use Presburger formulas to specify a program's cache misses, the Omega Calculator [20] to simplify the formulas, PolyLib [29] to obtain an indiscriminating union of polytopes, and finally, Ehrhart polynomials to count the number of integer points (i.e. misses) in each polytope [7]. While the cache misses for both perfect and imperfect loop nests containing possibly IF statements can be specified, they do not yet have a practical algorithm for finding cache misses from this specification.

In [28], we present a method for analysing the cache behaviour of whole programs (consisting of parallel loop nests and call statements) with regular computations. In this paper, we describe our technique for analysing loop nests with IF conditionals, which is the basis for the HPCA paper but was cited only as a technical report there due to space limitations.

There has been a great deal of research on applying loop and data transformations to improve the cache performance of loop-oriented codes [13, 16, 18, 22, 31, 30]. In particular, researchers have explored the use of various compiler heuristics and simple cache cost models to choose appropriate tile sizes [5, 8, 15, 30] and appropriate padding amounts [16, 22]. Analytical methods promise to provide more accurate knowledge about cache misses to guide a range of compiler optimisations.

9 Conclusion

We have presented an analytical method for analysing the cache behaviour of perfect loop nests 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. Our replacement miss equations are formulated and solved by taking into account the fact that the RISs for different references can be different. 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 [13, 26].

While this work represents a useful step towards a mechanical analysis of complex program constructs, there are several important constructs that are still non-analysable, including data-dependent conditionals and pointers. We are presently working on developing an analytical method for their efficient analysis. We intend to investigate benefits and limitations of this challenging but important research direction.

10 Acknowledgements

Thanks to Josep Llosa of Universitat Politécnica de Catalunya for his helpful comments on an early draft of this paper. This work has been supported by an Australian Research Council Grant A10007149.

References

[1] G. Ammons, T. Ball, and J. 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] E. Ayguadé, C. Barrado, A. González, J. Labarta, J. Llosa, D. López, S. Moreno, D. Padua, F. Reig, Q. Riera, and M. Valero. Ictineo: a tool for research

on ILP. In *Supercomputing'96*, 1996. Research Exhibit "Polaris at Work".

[3] N. Bermudo and X. Vera. Coyote project: Documentation. Technical Report MRTC Report 39/2001, Mälardalens Högskola, Oct. 2001.

[4] N. Bermudo, X. Vera, A. González, and J. Llosa. An efficient solver for cache miss equations. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'00)*, 2000.

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

[6] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *ACM SIGPLAN'01 Conference on Programming Language Design and Implementation (PLDI'01)*, 2001.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] S. Ghosh, M. Martonosi, and S. 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.

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

[15] 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.

[16] 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.

[17] I. Kodukul, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI'97)*, pages 346–357, Las Vegas,, 1997.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 215–228, May 1999.

[24] 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.

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

[26] X. Vera, J. Llosa, A. González, and N. Bermudo. A fast and accurate approach to analyze cache memory behavior. In *European Conference on Parallel Computing (Europar'00)*, 2000.

[27] X. Vera, J. Llosa, A. González, and C. Ciuraneta. A fast implementation of cache miss equations. In *8th International Workshop on Compilers for Parallel Computers (CPC'00)*, 2000.

[28] X. Vera and J. Xue. Let's study whole-program cache behaviour analytically. In *International Conference on High Performance Computer Architecture (HPCA-8)*, pages 175–186, Cambridge, Feb. 2002.

[29] D. Wilde. A library for doing polyhedral operations. Technical report, Oregon State University, 1993.

[30] M. E. Wolf and M. 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.

[31] M. J. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley, 1996.

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

[33] J. Xue. *Loop Tiling for Parallelism.* Kluwer Academic Publishers, Aug. 2000.

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