Let's Study Whole-Program Cache Behaviour Analytically[*]

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. Based on a new characterisation of data reuse across multiple loop nests, we present a method, a prototyping implementation and experimental results for analysing the cache behaviour of whole programs with regular computations. Validation against cache simulation using real codes confirms the efficiency and accuracy of our method. The largest program we have analysed, Applu from SPECfp95, has 3868 lines, 16 subroutines and 2565 references. Assuming a 32KB cache with a 32B line size, our method obtains the miss ratio with an absolute error of about 0.8% in about 128 secs while the simulator used runs for nearly 5 hours on a 933MHz Pentium III PC. Our method can be used to guide compiler locality optimisations and improve cache simulation performance.

1 Introduction

Data caches are a key component to bridge the increasing performance gap between processor and main memory speeds. However, caches are effective only when programs exhibit sufficient data locality in their memory access patterns. Optimising compilers attempt to apply loop transformations such as tiling [2, 5, 13, 23, 25] and data transformations such as padding [11, 12, 16, 17] to improve the cache performance of a program. The models guiding these transformations (in making an appropriate choice of parameter values such as tile and padding sizes) are mostly heuristic or approximate. Memory system designers often use cache simulators to evaluate alternative design options. In both cases, a fast and accurate assessment of a program's cache behaviour at compile time is useful in guiding compiler optimisations and improving cache simulation performance.

In the past few years, some progress has been made in the development of compile-time analytical methods for predicting cache behaviour. These include the Cache Miss Equations (CMEs) [10], the probabilistic method described in [8] and the Presburger-formulas-based method described in [3]. The underlying idea is to set up mathematical formulas to provide a precise characterisation of the cache behaviour of a program in the hope that, if these formulas can be solved or manipulated efficiently, then the information gathered such as the number or causes of cache misses can be exploited for various performance-enhancing purposes. However, the CMEs [10] and the probabilistic method [8] are limited to analysing perfect loop nests with straight-line assignments. The Presburger-formulas-based method [3], which is capable of handling multiple nests and IF conditionals, has been applied only to loop nests of small problem sizes with a few references. Its feasibility in analysing larger loop nests of realistic problem sizes remains to be seen. Neither of these methods can handle call statements.

This paper presents an analytical method for predicting the cache behaviour of complete programs with regular computations. By building primarily on and extending Wolf and Lam's framework [23] for quantifying reuse and the CMEs for characterising cache misses, we make the following contributions.

Reuse Analysis. By generalising traditional concepts such as iteration vectors and uniformly generated references for perfect loop nests, we introduce reuse vectors for quantifying reuse between references contained in multiple nests. Our reuse representation includes Wolf and Lam's reuse vectors [23] as a special case, allowing potentially existing reuse-driven optimisations to be applied to multiple nests.

Whole Program Analysis. We can handle programs with regular computations consisting of subroutines, call statements, IF statements and arbitrarily nested loops. In order to predict a program's cache behaviour statically, these programs must be free of data-dependent constructs (e.g., variable bounds, data-dependent IF conditionals, indirection arrays and recursive calls).

Prototyping Implementation. Our prototyping

[*]HPCA'02, pages 175 – 186, 2002. This work was carried out during the 1st author's visit to the 2nd author at UNSW.

system consist of components on normalising loop nests, inlining calls (abstractly), generating reuse vectors, sampling memory accesses and forming and solving equations for cache misses.

Whole Program Evaluation Results. We have validated our method against cache simulation using programs from SPECfp95, Perfect Suite, Livermore kernels, Linpack and Lapack. We include the results for three kernels and three whole programs. The largest program we have analysed, Applu from SPECfp95, has 3868 lines of FORTRAN code, 16 subroutines and 2565 references. Assuming a 32KB (directed, 2-way or 4-way) cache with a 32B line size, our method obtains the miss ratios with absolute errors (0.78%, 0.82% and 0.84%) in about 128 secs while the cache simulation runs for nearly 5 hours on a 933MHz Pentium III PC. In comparison with the three recent compile-time analytical methods [3, 8, 10], ours is the only one capable of analysing this scale of programs efficiently with accuracy.

The rest of this paper is organised as follows. Section 2 defines the cache architectures used. Section 3 describes our program model, with a particular emphasis on abstract call inlining and our new representation of reuse vectors for multiple nets. Section 4 outlines two algorithms for cache behaviour analysis. Section 5 describes the structure of our prototyping implementation. Section 6 contains some experimental results. Section 7 discusses the related work. Section 8 concludes the paper and discusses future work.

2 Cache Model

We assume a uniprocessor with a k-way set associative data 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. In a k-way set associative cache, a cache set contains k distinct cache lines. C_s and L_s denote the cache size and 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.

3 Program Model

Presently, we restrict ourselves to analysing FORTRAN programs with regular computations. We can handle programs made up of subroutines consisting of possibly IF statements, call statements and arbitrarily nested loops. In order to predict at compile time a program's cache behaviour, the following restrictions are imposed. All loop bounds and array subscript expressions must be affine in terms of the enclosing loops. All IF conditionals must be expressions consisting of loop indices and compile-time known constants. The base addresses of all non-register variables including actual parameters (scalars or arrays) must be known at compile time. The sizes of an array in all but the last dimension must be known statically.

Our program model excludes all and only data-dependent constructs (e.g., variable bounds, data-dependent IF conditionals and indirection arrays).

3.1 Loop Nest Normalisation

We normalise all loop nests to put a program into a suitable form for analysis. During normalisation, we apply loop sinking to move all statements into their respective innermost loops by adding IF conditionals appropriately [23, 24]. After normalisation, all loop nests are n-dimensional, and, in addition, all loop variables at depth k are normalised to I_k.

3.2 Iteration Vectors

A particular instance of a statement S (known as an *iteration* or *iteration point*) of the enclosing loop nest is identified by a $2n$-dimensional *iteration vector* of the form $\vec{\imath} = (\ell_1, I_1, \ell_2, I_2, \ldots, \ell_n, I_n)$, where

- $\vec{L} = (\ell_1, \ell_2, \cdots, \ell_n)$ is the *loop label* (*vector*) for the innermost loop containing S, and
- $\vec{I} = (I_1, I_2, \ldots, I_n)$ is the *index vector* consisting of the indices of the n loops enclosing S.

Figure 1 lists the iteration vector for each statement in the example. It is not difficult to see how the iteration vectors are derived in general.

```
DO $I_1 = \ldots$                                Iteration Vector
  DO $I_2 = \cdots$
    $S_1$: $B(I_2 - 1, I_1) = \cdots$            $(1, I_1, 1, I_2)$
   DO $I_2 = \ldots$
    $S_2$: $\cdots = B(I_2, I_1)$                $(1, I_1, 2, I_2)$
DO $I_1 = \cdots$
  DO $I_2 = \cdots$
    $S_3$:$B(I_1, I_2) = \cdots$                 $(2, I_1, 1, I_2)$
```

Figure 1: The iteration vectors for statements.

As usual, the set of all iterations for a particular loop nest is called the *iteration space* of that nest.

In a sequential execution, all iteration points are executed in lexicographical order. The usual lexicographic order operators $\prec$, $\preceq$, $\succ$ and $\succeq$ are used later.

3.3 Reference Iteration Spaces

The *reference iteration space (RIS)* of a reference R, denoted RIS_R, is defined 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 enclosing loop nest. Otherwise, the RIS can be a subspace of that iteration space.

We can handle any IF conditionals involving loop indices and compile-time constants. For a dedicated treatment of IF statements, we refer to [20].

3.4 Uniformly Generated References

After loop nest normalisation, $\vec{I} = (I_1, I_2, \ldots, I_n)$ is the index vector of all n-dimensional loop nests. The concept of uniformly generated references for perfect loop nests [9] can be carried over to multiple nests. There are two uniformly generated reference sets in Figure 1: $\{B(I_2-1, I_1), B(I_2, I_1)\}$ and $\{B(I_1, I_2)\}$.

3.5 Reuse Vectors

We generalise Wolf and Lam's reuse framework [23, 25] to calculate reuse vectors for multiple nests. We also add additional spatial reuse vectors to capture the reuse spanning two adjacent columns of an array.

Let R_p and R_c be two uniformly generated references (c stands for consumer and p for producer), which may be contained in different nests. Let R_p be the producer $A(M\vec{I}+\vec{m_p})$ nested inside the innermost loop labelled by $(\ell_1^p, \ell_2^p, \cdots, \ell_n^p)$ and R_c be the consumer $A(M\vec{I}+\vec{m_c})$ nested inside the innermost loop labelled by $(\ell_1^c, \ell_2^c, \cdots, \ell_n^c)$, where $\vec{I} = (I_1, I_2, \ldots, I_n)$.

Let $\vec{x} = (x_1, x_2, \ldots, x_n)$ be a solution to:

$$M\vec{x} \quad = \quad \vec{m_p} - \vec{m_c} \qquad (1)$$

and

$$\vec{r_t} = (\ell_1^c - \ell_1^p, x_1, \ell_2^c - \ell_2^p, x_2, \ldots, \ell_n^c - \ell_n^p, x_n)$$

such that $\vec{r_t} \succeq 0$. Then $\vec{r_t}$ is a *temporal reuse vector* from R_p to R_c. In addition, $\vec{r_t}$ represents *self* reuse if R_p and R_c are identical and *group* reuse otherwise.

There are two kinds of spatial reuse vectors depending on whether they span a single array column or not. Those spanning a single column are derived algebraically just like temporal reuse vectors.

In FORTRAN, all arrays are column-major. Let $\vec{y} = (y_1, y_2, \ldots, y_n)$ be a solution to:

$$\begin{array}{l} M'\vec{y} = \vec{m'_p} - \vec{m'_c} \\ |M_1\vec{y} - (m_p^1 - m_c^1)| < L_s \end{array} \qquad (2)$$

but not a solution to (1), where M_1 is the first row of M, m_p^1 (m_c^1) is the first entry of $\vec{m_p}$ ($\vec{m_c}$), and every primed term is obtained from its corresponding term in (1) with its first row or entry removed. Let

$$\vec{r_s} = (\ell_1^c - \ell_1^p, y_1, \ell_2^c - \ell_2^p, y_2, \ldots, \ell_n^c - \ell_n^p, y_n)$$

such that $\vec{r_s} \succeq 0$. Then $\vec{r_s}$ is a *spatial reuse vector* from R_p to R_c. In addition, $\vec{r_s}$ represents *self* reuse if R_p and R_c are identical and *group* reuse otherwise.

If a memory line spans two adjacent columns of an array, we will add spatial reuse vectors to capture such reuse. The spatial reuse vectors of this second kind are added individually depending on the iteration space shapes and cache parameters used.

Let us derive reuse vectors for the first two references to B in Figure 1. Let R_p be $B(I_2-1, I_1)$ nested in the inner loop labelled by $\vec{L_p} = (1,1)$ and R_c be $B(I_2, I_1)$ nested inside the inner loop labelled by $\vec{L_c} = (1,2)$. The subscript expressions for both references are affine:

$$\begin{array}{rcl} M\vec{I} + \vec{m_p} & = & \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right] \left[\begin{smallmatrix} I_1 \\ I_2 \end{smallmatrix}\right] + \left[\begin{smallmatrix} -1 \\ 0 \end{smallmatrix}\right] \\ M\vec{I} + \vec{m_c} & = & \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right] \left[\begin{smallmatrix} I_1 \\ I_2 \end{smallmatrix}\right] + \left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right] \end{array}$$

In this case, the equation (1) becomes:

$$\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right] \left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right] = \left[\begin{smallmatrix} -1 \\ 0 \end{smallmatrix}\right]$$

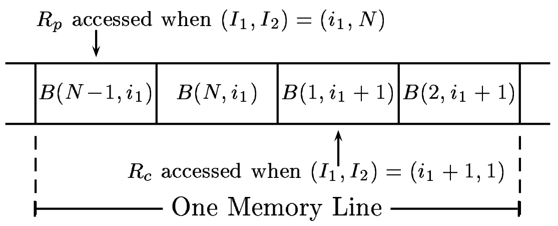
which has the unique solution $(0, -1)$. Thus, the unique temporal reuse vector from $B(I_2-1, I_1)$ to $B(I_2, I_1)$ is $(0, 0, 1, -1)$. To find the spatial reuse vectors spanning a single column of B, we solve:

$$\begin{array}{rcl} \left[\begin{smallmatrix} 1 & 0 \end{smallmatrix}\right] \left[\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix}\right] & = & 0 \\ \left[\begin{smallmatrix} 0 & 1 \end{smallmatrix}\right] \left[\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix}\right] + 1 & < & L_s \end{array}$$

which is the instance of the equation (2) for this case. Thus, all these spatial reuse vectors have the form $(0, y_2)$. Our reuse vector generator produces the following spatial reuse vectors $(0,0,1,-2), (0,0,1,-3), \ldots, (0,0,1,-L_s)$.

Finally, our reuse vector generator will generate $(0,1,0,1-N)$ to capture the reuse for the elements at the end of one array column and the beginning of the next column. This is illustrated in Figure 2.

If a reference is guarded by an IF conditional, its RIS may not be the entire iteration space of the enclosing loop nest. This causes complications only in



R_p accessed when $(I_1, I_2) = (i_1, N)$
$B(N-1, i_1)$
$B(N, i_1)$
$B(1, i_1+1)$
$B(2, i_1+1)$
R_c accessed when $(I_1, I_2) = (i_1+1, 1)$
One Memory Line

Figure 2: Spatial reuse across array columns ($L_s = 4$).

the derivation of group temporal reuse vectors. The self temporal and spatial reuse vectors for a reference are defined and derived without a need to refer to its RIS. As for the group reuse vectors from R_p to R_c, our implementation will generate all potential ones conservatively. In the case of group temporal reuse, there can be infinitely many reuse vectors to from some facets of RIS_{R_p} to some facets of RIS_{R_c}. In our implementation, these reuse vectors are ignored. Our extensive validation reported in [20] confirm that an overestimation of cache misses thus caused is negligible since (a) we overestimate only on some facets of RIS_{R_c} and (b) R_c may reuse on the facets by other reuse vectors (usually self reuse vectors).

3.6 Call Statements

In FORTRAN, all arguments are passed by reference. In an attempt to analyse exactly a program containing call statements, we perform an *abstract inlining* for a call whenever possible. We do not actually generate the inlined code. We only need to obtain the information required for analysing the inlined code. Each subroutine is associated with an *abstract function* consisting of the information about the memory accesses to the run-time stack, its code body (i.e., its loop nests with references), and local variable and formal parameter declarations. As shown in Figure 3, every call to a subroutine is abstractly inlined by replacing the call with the information in the abstract function associated with the subroutine.

The calling conventions used for a program are compiler- and architecture-dependent. Figure 3 depicts one such a convention for 32 bit machines. *Stack* denotes the run-time stack modelled as a one-dimensional array of an infinite size. If SP is 0 initially, its value is known at compile time at every call site due to the absence of recursive calls. The base address of *Stack*, if unknown at compile time, has to be obtained at run time. Then *Stack* is treated just like an ordinary array reference. For large programs,

Program	Actual Parameters			Calls	
	P-able	R-able	N-able	Total	A-able
Tomcatv	0	0	0	0	0
swim	0	0	0	5	5
su2cor	503	87	0	150	150
hydro2d	122	0	19	82	82
mgrid	68	0	35	23	2
applu	79	0	0	23	23
apsi	1601	0	210	186	118
fppp	83	0	3	17	16
turb3D	759	0	75	111	86
wave5	591	2	110	171	127
CSS	2489	0	8	965	965
LWSI	140	0	19	28	18
MTSI	186	0	2	63	63
NASI	236	0	237	75	41
OCSI	620	0	48	244	209
SDSI	189	18	49	129	103
SMSI	321	0	41	53	38
SRSI	242	0	176	50	13
TFSI	137	0	91	44	13
WSSI	836	127	7	185	179
TOTAL	9202	234	1130	2604	2251
%	87.09	2.21	10.89	100	86.44

Table 1: Statistics for the actual parameters and calls in SPECfp95 and Perfect benchmarks.

```
                    ...
                    Stack[BP] = RetAddr
                    Stack[BP + 4] = @A
                    Stack[BP + 8] = @B
...                 ... = Stack[BP − 4]
CALL f(A, B)  ⇒     ... = Stack[BP − 8]
...                 f's code body (with the formals
                    replaced by actuals or renamed)
                    RetAddr = Stack[BP − 12]
                    ...
```

Figure 3: Abstract inlining of a subroutine call.

the impact of these stack accesses is insignificant.

Not every call can be inlined according to Table 1. To analyse a call exactly, our method needs to know at compile time the base addresses of all its actual parameters. Let AP be an actual parameter that is either a scalar or an array variable or a subscripted variable with an affine data access expression and FP be its matching formal parameter.

AP is *propagateable* if, after inlining, every reference to FP can be replaced by a reference to AP. This allows the reuse to AP both in the caller and in all the callees to be potentially exploited. In Column "P-able", we consider AP as propagateable if FP is a scalar, or an one-dimensional array or if both AP and FP are arrays of the same dimensionality with matching sizes in all but the last dimension.

AP is *renameable* if, after inlining, every reference to FP can be replaced by a reference to AP' such that AP and AP' have the same base address (i.e., $@AP = @AP'$). The propagateable actuals are not also classified as renameable. In Column "R-able", we consider AP as renameable if the sizes of all but the last dimension for AP and FP are known statically. This still allows the reuse between the references to FP in the same subroutine to be exploited.

In Column "N-able", the actuals that are neither propagateable nor renameable, known as *non-analysable*, are represented. The propagateable and renameable actuals are potentially *analysable* since all references to FP can be analysable if affine.

A call can be abstractly inlined, i.e., is potentially *analysable*, if all its actuals are analysable. Table 1 shows that we can inline 86.44% of calls from SPECfp95 and Perfect benchmarks. These statistics are obtained by examining only a call and its callee.

Figure 4 serves to illustrate the inlining of a code segment (which may have out of array bound accesses if loop bounds are not chosen properly). The inlined code does not compile but can be analysed by our method. Hence, the name abstract inlining.

Finally, system calls (to I/O subroutines and intrinsic functions) are not inlined. The memory accesses inside are not accounted for. Theses calls can be inlined if their abstract functions are known.

```
REAL*8 X, A, B
DIMENSION A(10, 10), B(20, 20)
DO I1 = ...
  DO I2 = ...
    A(I1, I2) = ···
    CALL f(X, A, B, B(I1, I2))
    CALL g(A(I1, I2), A(1, I2), B)

SUBROUTINE f(Y, C, D, S)
REAL*8 Y, C, D, S
DIMENSION C(10, 10), D(400), S(10, 10, *)
DO I3 = ...
  DO I4 = ...
    C(I3, I4 − 1) = Y + D(I3 − 1 + 20 * (I4 − 1))
    S(I3, I4, 2) = ···

SUBROUTINE g(E, F, T)
...
REAL*8 E, F, T
DIMENSION E(10, 10), F(10), T(100, 4)
DO I3 = ...
  DO I4 = ...
    E(I3, I4) = F(I4) − T(I3, I4)
```

$$\Downarrow$$

```
REAL*8 X, A, B, B1, B2
DIMENSION A(10, 10), B(20, 20)
C THE FOLLOWING LINE DOES NOT COMPILE
DIMENSION B1(10, 10, *), B2(100, 4)
DO I1 = ...
  DO I2 = ...
    A(I1, I2) = ···
    DO I3 = ...
      Do I4 = ...
        A(I3, I4−1) = X + B(I3−1 + 20 * (I4−1))
        B1(I1 + 10 * (I2 − 1) + I3 − 1, I4, 2) = ···
    DO I3 = ...
      Do I4 = ...
        A(I1+I3−1, I2+I4−1) = A(I4, I2)−B2(I3, I4)
```

Figure 4: Propagation and renaming of actual parameters. All actuals but the last are propagated. The last actuals in both calls are renamed to $B1$ and $B2$, respectively. After inlining, $@B = @B1 = @B2$.

4 Cache Behaviour Analysis

There are two kinds of miss equations: *compulsory or cold (miss) equations* and *replacement (miss) equations*. *Cold* misses represent the first time a memory line is touched while *replacement* misses are those accesses that result in misses because the cache lines that would have been reused were evicted from the cache before they get reused.

4.1 Forming Equations

Let $\vec{r}$ be a reuse vector from the producer reference R_p to the consumer reference R_c. We want to find out if R_c at iteration $\vec{\imath}$ can reuse the cache line accessed by R_p at $\vec{\imath} - \vec{r}$. Let R_i be an *intervening* reference such that the access of R_i at some iteration point $\vec{\jmath}$ between $\vec{\imath} - \vec{r}$ and $\vec{\imath}$ may be mapped to the same cache set as the access of R_p at $\vec{\imath} - \vec{r}$. If that happens, a *set contention* occurs between the access of R_p at $\vec{\imath} - \vec{r}$ and the access of R_i at $\vec{\jmath}$. In a k-way set associative cache, it takes k distinct set contentions to evict the cache line touched by the access of R_p at $\vec{\imath} - \vec{r}$.

We give below the miss equations that can be further analysed to determine if the access of R_c at $\vec{\imath}$ is a miss or hit, assuming the single reuse vector $\vec{r}$ from R_p to R_c and the single intervening reference R_i.

$Mem\_Line_R(\vec{\imath})$ ($Cache\_Set_R(\vec{\imath})$) denotes the memory line (cache set) to which the memory address accessed by reference R at iteration $\vec{\imath}$ is mapped.

4.1.1 Cold Equations

The cold equations for R_c along $\vec{r}$ represent the iteration points where the memory lines are brought to the cache for the first time:

$$\begin{gathered}\vec{\imath} \in RIS_{R_c}\\ \text{and}\\ (\vec{\imath} - \vec{r} \notin RIS_{R_p}\\ \text{or}\\ Mem\_Line_{R_c}(\vec{\imath}) \neq Mem\_Line_{R_p}(\vec{\imath} - \vec{r}))\end{gathered}$$

If $\vec{r}$ is temporal, the inequality is false and thus redundant.

```
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 5: Two algorithms for computing the cache misses from the cold and replacement miss equations.

4.1.2 Replacement Equations

The *replacement equations* for R_c along $\vec{r}$ are to investigate if R_c at iteration $\vec{\imath}$ can reuse the cache line that R_p accessed at iteration $\vec{\imath} - \vec{r}$ subject to the set contentions caused by the memory accesses from R_i at all intervening points executed between $\vec{\imath} - \vec{r}$ and $\vec{\imath}$:

$$\begin{gathered} Mem\_Line_{R_c}(\vec{\imath}) = Mem\_Line_{R_p}(\vec{\imath} - \vec{r}) \\ \vec{\imath} \in RIS_{R_c} \\ \vec{\imath} - \vec{r} \in RIS_{R_p} \\ Cache\_Set_{R_c}(\vec{\imath}) = Cache\_Set_{R_i}(\vec{\jmath}) \\ \vec{\jmath} \in J_{R_i} \end{gathered}$$

where J_{R_i} denotes the set of all these intervening iteration points, called the *interference set* for R_c along $\vec{r}$, and is specified precisely by:

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

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.

4.2 Solving Equations

Figure 5 gives two algorithms for obtaining the cache misses from a looping construct consisting of multiple references and reuse vectors. Both *FindMisses* and *EstimateMisses* analyse each reference by going through its reuse vectors in lexicographical order $\prec$. If an iteration point is a solution to the cold equations along the current reuse vector $\vec{r}$, its behaviour is indeterminate and will be examined further using the other reuse vectors later in the list. Otherwise, the iteration point is classified either as a hit or a miss using the replacement equations along $\vec{r}$. After all reuse vectors have been tried, the remaining indeterminate iteration points are cold misses.

Our replacement equations represent only cache set contentions. In a k-way set associative cache, it takes k distinct cache set contentions to cause a cache line to be evicted from the cache set. There will be a cache miss only when k distinct solutions are found [10, 20].

FindMisses analyses all iteration points in a RIS and is practical only for programs of small sizes [10, 20]. *EstimateMisses* analyses a sample of a RIS and is capable of analysing programs significantly more efficiently with a controlled degree of accuracy. For technical details regarding c and w, we refer to [6, 19].

We compute the volume of a RIS using an algorithm discussed in [20]. Other methods for computing the volume of convex polytopes exist [4, 15].

5 Prototyping Implementation

Figure 6 depicts the structure of our prototyping system for finding cache misses and validating the accuracy of our method against a cache simulator. The component *Opts* optimises the program and allocates variables to registers or memory. The reuse vectors, the base addresses of variables and the relative access order of memory references are obtained from a load-store lower-level IR, which is produced from the Polaris IR [7] of the program. The inlining component is currently being implemented. The same information obtained is fed to both our algorithms and the cache simulator used.

6 Experiments

We present our results for three isolated kernels and three whole programs. For a detailed evaluation of

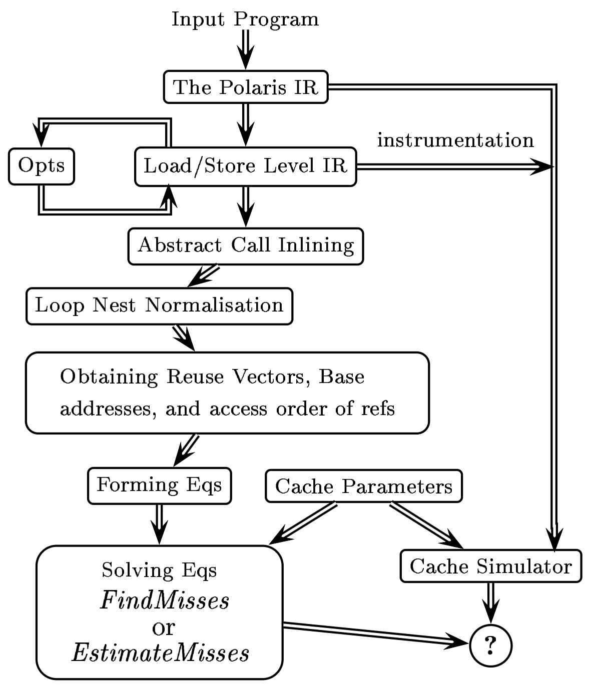


Figure 6: A framework for analysis and evaluation.

codes consisting of IF statements, we refer to [20].

Unless otherwise specified, we assume a 32KB cache with a 32B cache line size. The execution times are all obtained on a 933MHz PentiumIII PC.

6.1 Multiple Loop Nest Kernels

We evaluate the accuracy of our method by comparing *FindMisses* (which analyses all iteration points) with a cache simulator. Table 2 presents the results in both cases for caches of different associativities. In all but one case, our method obtains exactly the same miss ratio as the simulator. In the exceptional case, we overestimate slightly the miss ratio by 0.05.

Figure 7 shows the three kernels used:

- **Hydro** is a 2-D explicit hydrodynamics from Livermore (kernel 18). *FindMisses* and the simulator yield the same results in all cases.
- **MGRID** is a 3-D loop nest from MGRID. Again *FindMisses* and the simulator agree on their results in all cache configurations.
- **MMT** is a 3-D blocked loop nest taken from [8] that computes the matrix multiplication A and B^T. The two references to WB are not uniformly generated due to the transposition of B. Being unable to exploit their reuse, *FindMisses* over-estimates the cache miss ratios in all three cases slightly. Due to transposition, the degree of reuse between the two references is rather minimal. The inaccuracy lies in the incompleteness of reuse information rather than our method iself.

Program	Cache	Abs. Error	Execution Time (secs)
Hydro (KN=JN=100)	direct	0.05	0.27
	2-way	0.05	0.32
	4-way	0.05	0.36
MGRID (M=100)	direct	0.36	0.19
	2-way	0.32	0.22
	4-way	0.32	0.22
MMT (B=BJ=100 & BK=50)	direct	0.23	0.10
	2-way	0.37	0.10
	4-way	0.37	0.11

Table 3: Cache misses from *EstimateMisses* for 32KB caches with a 32B line size ($c = 95\%$ and $w = 0.05$).

Table 2 indicates that *FindMisses*, while being capable of finding exactly cache miss numbers, does so at the expense of large execution times.

Table 3 shows the accuracy and efficiency of *EstimateMisses* using a 95% confidence with an interval of 0.05 for all references in the program. In all cases, the absolute errors are less than 0.4 and the execution times less than 0.5 seconds. Note that *EstimateMisses* yields only the miss ratio for a program. The actual miss ratio of each kernel is available in Table 2.

6.2 Whole Programs

We evaluate *EstimateMisses* against a simulator using three programs from SPECfp95 detailed in Table 4. In each case, we have succeeded in abstractly inlining all the calls and obtained one loop nest for the program. In addition, all actual parameters are propagateable, meaning that the references to every actual can be potentially exploited across calls. Since our inlining component is not working yet, all calls were inlined by hand. Each program is analysed using the reference input data. Thus, the variables in all READ statements are initialised from the reference data and then treated as compile-time constants.

Table 5 presents the experimental results obtained. For a scale of programs such as Applu, *EstimateMisses* obtains close to real miss ratios in about 128 seconds. This translates into a three orders of magnitude speedup over the cache simulator used!

Our results are further discussed below.

Tomcatv from SPECfp95. This example is used to demonstrate the capability of of our method in analysing real codes. The number of iterations of the outermost loop is data-dependent.

```
PROGRAM Hydro
REAL*8 ZA, ZP, ZQ, ZR, ZM, ZB, ZU, ZV, ZZ
DIMENSION ZA(jn+1,kn+1), ZP(jn+1,kn+1), ZQ(jn+1,kn+1), ZR(jn+1,kn+1), ZM(jn+1,kn+1))
DIMENSION ZB(jn+1,kn+1), ZU(jn+1,kn+1), ZV(jn+1,kn+1), ZZ(jn+1,kn+1)
T= 0.003700D0
S=0.004100D0
DO k= 2,KN
  DO j= 2,JN
    ZA(j,k)=(ZP(j-1,k+1)+ZQ(j-1,k+1)-ZP(j-1,k)-ZQ(j-1,k))*(ZR(j,k)+ZR(j-1,k))/(ZM(j-1,k)+ZM(j-1,k+1))
    ZB(j,k)= (ZP(j-1,k)+ZQ(j-1,k)-ZP(j,k)-ZQ(j,k))*(ZR(j,k)+ZR(j,k-1))/(ZM(j,k)+ZM(j-1,k))
  ENDDO
ENDDO
DO k= 2,KN
  DO j= 2,JN
    ZU(j,k)= ZU(j,k)+S*(ZA(j,k)*(ZZ(j,k)-ZZ(j+1,k))-ZA(j-1,k)*(ZZ(j,k)-ZZ(j-1,k))
             -ZB(j,k)*(ZZ(j,k)-ZZ(j,k-1))+ZB(j,k+1) *(ZZ(j,k)-ZZ(j,k+1)))
    ZV(j,k)= ZV(j,k)+S*(ZA(j,k)*(ZR(j,k)-ZR(j+1,k))-ZA(j-1,k) *(ZR(j,k)-ZR(j-1,k))
             -ZB(j,k) *(ZR(j,k)-ZR(j,k-1))+ZB(j,k+1) *(ZR(j,k)-ZR(j,k+1)))
  ENDDO
ENDDO
DO k= 2,KN
  DO j= 2,JN
    ZR(j,k)= ZR(j,k)+T*ZU(j,k)
    ZZ(j,k)= ZZ(j,k)+T*ZV(j,k)
  ENDDO
ENDDO
END
```

```
PROGRAM MGRID
REAL*8 U,Z
DIMENSION U(M,M,M), Z(M,M,M)
DO 400 I3=2,M-1
  DO 200 I2=2,M-1
    DO 100 I1=2,M-1
      U(2*I1-1,2*I2-1,2*I3-1)=U(2*I1-1,2*I2-1,2*I3-1)
        +Z(I1,I2,I3)
100 CONTINUE
    DO 200 I1=2,M-1
      U(2*I1-2,2*I2-1,2*I3-1)=U(2*I1-2,2*I2-1,2*I3-1)
        +0.5D0*(Z(I1-1,I2,I3)+Z(I1,I2,I3))
200 CONTINUE
  DO 400 I2=2,M-1
    DO 300 I1=2,M-1
      U(2*I1-1,2*I2-2,2*I3-1)=U(2*I1-1,2*I2-2,2*I3-1)
        +0.5D0*(Z(I1,I2-1,I3)+Z(I1,I2,I3))
300 CONTINUE
    DO 400 I1=2,M-1
      U(2*I1-2,2*I2-2,2*I3-1)=U(2*I1-2,2*I2-2,2*I3-1)
        +0.25D0*(Z(I1-1,I2-1,I3)+Z(I1-1,I2,I3)
        +Z(I1, I2-1,I3)+Z(I1, I2,I3))
400 CONTINUE
STOP
```

```
PROGRAM MMT
REAL*8 A, B, D, WB
DIMENSION A(N,N), B(N,N), D(N,N), WB(N.N)
DO J2 = 1,N,BJ
  DO K2 = 1,N,BK
    DO J=J2,J2+BJ-1
      DO K=K2,K2+BK-1
        WB(J-J2+1,K-K2+1)=B(K,J)
      ENDDO
    ENDDO
    DO I = 1,N
      DO K=K2,K2+BK-1
        RA=A(I,K)
        DO J=J2,J2+BJ-1
          D(I,J)=D(I,J)+
                 WB(J-J2+1,K-K2+1)*RA
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
END
```

Figure 7: Three kernels.

Program	Cache	#Cache Misses		%Loop Nest Miss Ratio		Abs.	Execution
		Simulator	*FindMisses*	Simulator	*FindMisses*	Error	Time (secs)
Hydro (KN=JN=100)	direct	52603	52603	14.12	14.12	0.00	1.07
	2-way	52603	52603	14.12	14.12	0.00	1.35
	4-way	42703	42703	11.47	11.47	0.00	1.64
MGRID (M=100)	direct	1518879	1518879	9.49	9.49	0.00	91.29
	2-way	1424038	1424038	8.90	8.90	0.00	99.45
	4-way	1424038	1424038	8.90	8.90	0.00	100.70
MMT (N=BJ=100 & BK=50)	direct	145671	147075	4.82	4.87	0.05	43.09
	2-way	171647	172592	5.68	5.71	0.03	47.06
	4-way	246980	247744	8.18	8.20	0.02	57.44

Table 2: Cache miss ratios for 32KB caches with a 32B line size from *FindMisses* and a cache simulator.

	Tomcatv	Swim	Applu
#lines	190	429	3868
#subroutines	1	6	16
#call-statements	0	6	27
#references	79	52	2565

Table 4: Three whole programs.

Program	Cache	Miss Ratio		Abs.	Exe.T	Sim.T
		Sim.	*E.M*	Err	(secs)	(secs)
Tomcatv	direct	11.42	11.02	0.40	0.30	3676.2
	2-way	11.40	11.0	0.40	0.37	3750.3
	4-way	11.41	11.0	0.41	0.58	3860.2
Swim	direct	7.26	7.01	0.25	2.47	8136.0
	2-way	6.98	6.73	0.25	2.63	8281.1
	4-way	7.24	6.97	0.27	3.23	8425.8
Applu	direct	6.95	7.73	0.78	127.31	17089
	2-way	6.60	7.42	0.82	127.6	17155
	4-way	6.56	7.40	0.84	127.5	17278

Table 5: Cache misses from *EstimateMisses* for 32KB caches with a 32B line size ($c = 95\%$ and $w = 0.05$).

N	BJ	BK	C_s	L_s	k	Δ_P	Δ_E
200	100	100	16	8	2	6.23	0.1
200	100	100	256	16	2	2.73	0.5
200	200	100	32	8	1	6.88	0.06
200	200	100	128	8	2	2.86	0.05
200	200	100	128	32	2	44.25	16
200	50	200	16	4	1	4.62	0.05
200	100	200	32	8	2	12.51	0.1
200	100	200	64	16	1	3.31	0.4
400	100	100	16	8	2	4.48	0.03
400	100	100	256	16	2	4.26	0.5
400	200	100	32	8	1	2.65	0.4
400	200	100	128	8	2	5.82	0.05
400	200	100	128	32	2	44.68	16
400	50	200	16	4	1	2.02	0.05
400	100	200	32	8	2	5.55	0.06
400	100	200	64	16	1	7.12	0.3

Table 6: Comparison with Fraguela et al's probabilistic method using MMT. Δ_p denotes the relative error between the estimated and real miss ratios for the probabilistic method and Δ_E for our *EstimateMisses*.

For the reference input data used, the outermost loop runs for 750 iterations. The only data-dependent IF conditional in the program is always false. The memory accesses contained in this conditional are included in our analysis.

Swim from SPECfp95. This example demonstrates that we can analyse codes consisting of call statements. All calls are parameterless. The outermost loop is an *IF-GOTO* construct, which has been converted into a *DO* statement.

Applu from SPECfp95. This shows that our method is capable of analysing this scale of programs efficiently with a good degree of accuracy. All actual parameters are propagateable. In subroutine SSOR, there are some data-dependent constructs. All but one are guarded by a IF branch that is false at compile time and are thus ignored. The remaining data-dependent IF construct is a WRITE statement for a register-allocated scalar. The memory accesses in this IF conditional are included in our analysis.

7 Related Work

We review in detail the three recent compile-time analytical methods for predicting cache behaviour [3, 8, 10]. For a survey on trace-driven simulation, see [18]. Recently, Weikle *et al* [21] introduce a trace-based idea of viewing caches as filters. Their framework can potentially handle any programs consisting of any pattern of memory references.

Ghosh *et al* [10] present their seminal work on using the CMEs to analyse statically a program's cache behaviour. This framework is targeted at isolated perfect nests consisting of straight-line assignments by emploiting only the reuse vectors between uniformly generated references in the same nest. They show that the CMEs can provide insights in choosing appropriate tile and padding sizes. Since analysing all iteration points is costly, an efficient implementation of the CMEs based on polyhedral theory and statistical sampling techniques is discussed in [1, 19].

Fraguela *et al* [8] rely on a probabilistic analytical method to provide a fast estimation of cache misses. While allowing multiple nests, they exploit only the reuse between references contained in the same nest (as can also be done in the CMEs.) These references differ by constants in their matching dimensions, forming a subset of uniformly generated references considered in the CMEs. Their experimental results using three examples indicate that their method can achieve a good degree of accuracy in estimating cache misses for perfect nests. Their two perfect nest examples can be analysed by the CMEs and are not compared here. The other one is a 3-D blocked imperfect nest computing AB^T (named MMT in Figure 7). Table 6 compares their method with ours. Our *EstimateMisses* produces better results in all cases. The two largest relative errors occur since the total number of misses is small.

Chatterjee *et al* [3] present an ambitious method for *exactly* modelling the cache behaviour of loop nests. They use Presburguer formulas to specify a program's cache misses, the Omega Calculator [14] to simplify the formulas, PolyLib [22] to obtain an indiscriminating union of polytopes, and finally, Ehrhart polynomials to count the number of integer points (i.e., misses) in each polytope [4]. They can formulate Presburger formulas for a looping structure consisting of imperfect nests, IF statements, references with affine accesses and non-linear data layouts. That is, they are not restricted to uniformly generated references and linear array layouts. When solving their formulas, they provide only the cache miss numbers for 20×20 and 21×21 matrix multiplication without giving any execution times. In the case of matrix-vector product, they give Presburger formulas for $N = 100$ but did not solve them.

Exact analysis is undoubtedly useful but can be too costly for realistic codes to be of any use in guiding compiler optimisations to improve performance. *FindMisses* can be exact if all necessary reuse vectors are used. Our current implementation exploits only the reuse among uniformly generated references. One future work is to derive systematically the reuse vectors for non-uniformly generated references.

Neither of the three methods discussed above can handle call statements. In comparison with these existing techniques, our method can analyse complete regular programs efficiently with accuracy.

8 Conclusions

We have introduced a new characterisation of reuse vectors for quantifying reuse across multiple nests. Based on these reuse vectors, we have developed an analytical method for statically predicting the cache behaviour of complete programs with regular computations. We outlined two algorithms for computing cache misses. *FindMisses* analyses all iteration points and can predict exactly the cache misses for programs of small problem sizes. *EstimateMisses* analyses a sample of all memory accesses and can achieve close to real cache miss ratios in practical cases efficiently. The experimental results obtained for three kernels and three whole programs (one of which is Applu from SPECfp95 with 3868 lines, 16 subroutines and 2565 references) confirm the efficiency and accuracy of our method. Our method can be used to guide compiler optimisations and improve the speeds of cache simulators.

While this work represents a step towards an automatic analysis of whole programs, data-dependent constructs (e.g., variable bounds, data-dependent IF conditionals and indirection arrays) are still non-analysable. We plan to investigate techniques for their analysis. To go beyond FORTRAN, we need to cope with pointer constructs and recursive calls.

Acknowledgements

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

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] M.H. DeGroot. *Probability and statistics.* Addison-Wesley, 1998.

[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] 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.

[9] 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.

[10] 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.

[11] S. Chatterjee V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layout for hierarchical memory systems. In *Procs. of Int. Conf. on Supercomputing (ICS'99)*, pages 444–453, Rhodes, Greece, June 1999.

[12] M. Kandemir, A. Choudhary, P. Banerjee, and J. Ramanujam. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):115–135, Feb. 1999.

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

[14] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Communications of the ACM*, 1992.

[15] 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.

[16] 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.

[17] O. Temam, E.D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assesing when data copying should be used to eliminate cache conflicts. In *Procs. of Supercomputing Conf. (SC'93)*, pages 410–419, 1993.

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

[19] 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.

[20] X. Vera and J. Xue. Analysing cache behaviour for programs with if statements. Technical Report UNSW-CSE-TR0107, University of New South Wales, May 2001.

[21] D. A. B. Weikle, K. Skadron, , S. A. McKee, and W. A. Wulf. Cache as filters:a unifying model for memory hierarchy analysis. Technical Report CS-2000-16, University of Virginia, June 2000.

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

[23] 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.

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

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