Predicting the Cache Miss Ratio of Loop-Nested Array References[*]

John S. Harper[†] Darren J. Kerbyson Graham R. Nudd

December 1, 1997

Abstract

The time a program takes to execute can be massively affected by the efficiency with which it utilizes cache memory. Moreover the cache-miss behavior of a program can be highly unpredictable, in that small changes to input parameters can cause large changes in the number of misses. In this paper we present novel analytical models of the cache behavior of programs consisting mainly of array operations inside nested loops, for direct-mapped caches. The models are used to predict the miss-ratios of three example loop nests; the results are shown to be largely within ten percent of simulated values. A significant advantage is that the calculation time is proportional to the number of array references in the program, typically several orders of magnitude faster than traditional cache simulation methods.

1 Introduction

It is widely recognized that the cache behavior of a program can be one of the most important factors affecting its performance. As the gulf between processor and memory speed widens, this factor is becoming more and more relevant to all but the most trivial of programs. In the field of scientific computation where large data arrays are commonly manipulated, and high performance is especially desirable, making efficient use of cache memory is a natural way to increase a program's performance. Optimizing cache performance in an *ad hoc* manner can yield some success (for example trying to access memory sequentially), but due to the number of parameters and the complicated nature of the processes involved, it is not possible to detect more subtle effects. It has also been shown that cache behavior can be very unstable, with small program changes often leading to large differences in execution time [2, 13, 8].

Evaluating cache performance has traditionally been restricted to *simulation* and *profiling*. Simulating the memory reference behavior of a program requires that the effect of every single memory access be emulated one by one.

[*] Research report CS-RR-336; Dept. of Computer Science, University of Warwick, Coventry CV4 7AL, UK.

[†] john@dcs.warwick.ac.uk

Profiling a program usually requires some level of hardware support, and allows memory access statistics to be recorded while the program is executing. Both of these methods give accurate results, the drawback being that evaluation takes at least as long as the execution time of the program being examined. Generally the situation will be significantly worse than this, with conventional trace-driven simulators many times slower than the program's execution time, and more recent simulation techniques still significantly slower [14].

This level of evaluation speed is acceptable for some applications, but when the number of possible scenarios to be evaluated increases, or results are required quickly, simulation or profiling may simply be too slow to be useful. As an example of just how long comprehensive simulation can take, Gee *et al.* reported that *40 months* of CPU time were required to simulate the ten SPEC92 benchmarks [6]. Another problem with simulation is that results produced will give little explanation of *why* a program incurs a particular miss-ratio on a particular cache configuration—the type of information valuable when optimizing a program. These combined problems suggest that an alternative to simulation would be useful.

There have been several attempts at analytically modeling cache behavior. One technique has been to analyze memory access traces, deriving a model of cache behavior. Agarwal *et al.* presented an analytical model that used parameters acquired from a trace, together with parameters describing the cache, to predict the miss-rate of several reasonably large reference traces [1]. Their method allows the general trends in cache behavior to be predicted, but lacks the fine-detail needed to detect the unstable nature of cache interference often observed [2, 13, 8].

Analytical models have also been used in compilers that attempt to optimize cache use, for example Wolf and Lam [15], Fahringer [4], and McKinley *et al.* [9]. These systems use analytical cache models to guide the selection of source code transformations. The models have often been designed specifically for this purpose, not with more general use in mind. As a result some inaccuracy may be tolerated, when it has no effect on which code transformations are selected.

The cache performance of specific types of algorithm has also been the subject of analytical modeling, with the performance of blocked algorithms a common subject. Lam and Wolf examined how blocked algorithms utilize the cache [8], giving guidelines for selecting the blocking factor. Coleman and McKinley described another method of selecting the block size, with the emphasis placed on minimizing interference of all types [3]. Fricker *et al.* also looked at blocking, examining in detail the interference occurring in a "matrix-vector multiply" kernel [5]. They stressed the need to consider parameters such as array base address in order to detect all types of interference, and that interference must be modeled precisely for sub-optimal performance to be avoided.

It is evident that all of these methods lack either the generality needed to model all types of algorithms, or the accuracy in all situations (not just over general trends) that is needed. An attempt at addressing these problems was made by Temam *et al.* [12, 13]. They outlined a model of interference in direct-mapped caches that can be applied to a wide range of numerical loop nestings. From the level of interference predicted they generated an estimate of the total number of cache misses incurred. Although their methods work well when applied to some types of loop nesting they are incapable of modeling certain types of simple loop nesting without introducing significant approximations.[1]

In this paper we present novel analytical models for predicting the direct-mapped cache miss behavior of a wide range of program fragments. The method combines some of the methods of Temam *et al.* for evaluating reuse and interference with new techniques that remove some of the restrictions on the types of code fragments that may be modeled accurately. We also provide experimental data from an implementation of the models, showing that accuracy within ten per cent of simulated values can generally be expected, calculated orders of magnitude faster than by simulation. The low cost of prediction creates new uses for cache modeling, for example, on the fly optimization of data layout to minimize cache misses; this has an important advantage over compile-time optimization in that the target cache configuration need not be known until run-time.

In the next section we formally define the problem. In Section 4 we describe the initial steps needed to prepare for the actual evaluation, including how each reference's reuse source is identified. Sections 5 and 6 show how the number of misses due to either self reuse or group reuse are evaluated. Several example code fragments are examined in Section 7, showing the type of problems that the model can address, and the accuracy that can be achieved; these results are discussed in Section 8. Finally, in Section 9 we summarise the work and present our conclusions. Appendix A lists the symbols and notation used throughout the paper.

2 Classification of Cache Behavior

A commonly used classification of cache misses is due to Hill [7], in which three types are given. *Compulsory misses* occur the first time a data element is referenced, *capacity misses* occur when the size of the cache is insufficient to hold the entire working set of the program, and *conflict misses* occur when two elements of the working set map to the same cache line. Sugumar further subdivided this last category into *mapping misses* and *replacement misses* [11].

These classifications describe the *effects* occurring, when modeling analytically it is helpful to concentrate on the *causes* of cache behavior. We achieve this by considering both capacity and conflict misses as a single category: *interference misses*, and modeling the level of interference on each array reference. In this classification a compulsory miss occurs the first time a data element is accessed, followed by interference misses each time the element is subsequently accessed but for some reason is no longer resident in the cache.

[1] For example, when an array is not accessed sequentially.

```
DO j1 = 0, N1 - 1
  DO j2 = 0, N2 - 1
    DO j3 = 0, N3 - 1
      A(j3,j1) = B(j3,j1) + C(j3) + C(j3+1)
    ENDDO
  ENDDO
ENDDO
```

Figure 1: Example loop nest

Classification of interference. Data reuse is created by avoiding interference misses; two types of reuse can be identified, *self-dependence reuse* due to an array reference repeatedly accessing the same elements, and *group-dependence reuse* where a reference accesses data recently accessed by another. For example, consider the loop nesting shown in Figure 1. Reference $A(j_3, j_1)$ exhibits self-dependence reuse—loop variable j_2 is unused therefore iterations of the middle loop access the same elements of matrix A as the previous iteration. On the other hand, reference $C(j_3)$ shows group-dependence reuse, it will access the element of array C that reference $C(j_3 + 1)$ accessed on the previous iteration of loop j_3.

The type of reuse achieved by each reference is found by identifying that reference's *reuse dependence*. This is the reference that most recently accessed data elements subsequently accessed by the reference in question. The "most-recently" metric is measured in iterations of the innermost loop, and is termed the *reuse distance* of the dependence. Having grouped all references into these two fundamental types the interference affecting each can be classified.

For a self dependent reference there are two main types of interference. *Self interference*, in which the reference itself prevents reuse, occurring when more than one data element maps to the same cache location, and *cross interference*, where other references access data that interferes. For evaluation purposes cross interference can be further subdivided into *internal* and *external* forms. Internal cross interference occurs between references with similar access patterns, and external between those with dissimilar access patterns. Group dependent references also suffer from cross interference, in both internal and external forms, but self-interference does not occur.

To illustrate these definitions again consider reference $A(j_3, j_1)$ in Figure 1. Cross interference on the reuse of $A(j_3, j_1)$ can be caused by any of the other references, i.e., $B(j_3, j_1)$, $C(j_3)$, or $C(j_3 + 1)$. Since reference $B(j_3, j_1)$ accesses memory in exactly the same pattern as $A(j_3, j_1)$, it could cause internal cross-interference. The other two references, $C(j_3)$ and $C(j_3 + 1)$, access memory in a different pattern to $A(j_3, j_1)$ and therefore can only be sources of external cross interference.

The final piece of our classification structure is to split all interference effects into two subtypes, *temporal* and *spatial* interference. Temporal interference occurs where reuse of individual data elements is disrupted, spatial interference when the reuse of data elements mapping to the same cache line is disrupted.

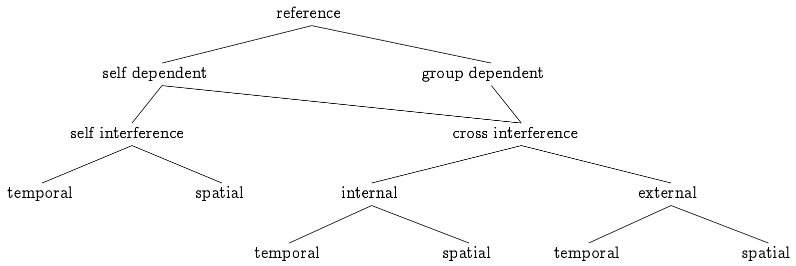

Figure 2: Interference hierarchy

Generally spatial interference is found to be reasonably low, but in certain cases it can account for very large variations in program behavior. For example in the matrix-multiply example in Section 7 the massive fluctuations in miss-ratio are caused almost solely by spatial interference on one of the references.

Figure 2 shows the full interference hierarchy. For each array reference being examined the relevant nodes of the tree are evaluated, giving the total number of interference misses for that reference.

3 Problem Description

Given a set of nested loops containing array references, whose index expressions depend on the loop variables, the problem is to predict the cache miss-ratio of the program fragment. Each reference is examined in turn to find the number of cache misses it causes. The description of the problem can be split into three parts, the structure of the cache, the arrays being accessed, and the loop nestings to be evaluated.

Definition 1. *Two parameters give the form of any direct-mapped cache: the number of elements in the cache, $\mathcal{C}$, and the number of elements in each line of the cache, $\mathcal{L}$. Both are defined in bytes.*

Definition 2. *Each array is given by three parameters, the first being its base address, the position in memory of its first element. The other parameters are the sizes of each dimension, and the size of each data element $\mathcal{E}$.*

The arrays must be discrete, no array may share memory locations with any other array.

In general, all arrays declared in Fortran will meet this restriction, some care may have to be taken when using the C language to avoid pointer aliasing.

For the model to evaluate a set of nested loops, they must follow a certain structure. Firstly, each loop must be normalized,

Definition 3. *The number of iterations of each loop must be constant, and each loop variable must start at zero and count upwards in steps of one.*

This is less of a constraint than it seems since any loop whose boundaries are constant, and which counts upwards, can be translated to this normal form.

In fact the implementation of the model mentioned in Section 7 performs this normalization automatically.

Each level of loop nesting is numbered, the outermost labeled as level 1, the innermost as level n. The number of iterations of a loop at level i is labeled $\mathcal{N}_i$ and the loop variable j_i. How this relates to the normal Fortran loop structure is shown in Figure 3.

```
DO j1 = 0, N1 - 1
  DO j2 = 0, N2 - 1
    DO j3 = 0, N3 - 1
      ...
    ENDDO
  ENDDO
ENDDO
```

Figure 3: Loop layout

Definition 4. *Each array reference must be of the form:*

$$X(\alpha_1 j_{\gamma_1} + \beta_1, \ldots, \alpha_m j_{\gamma_m} + \beta_m),$$

where X *is the name of the array,* m *is the number of dimensions it has, and* α_k*,* β_k*, and* γ_k *are constants (with* $1 \leq \gamma_k \leq n$*).*

In practice most programs contain only references that follow this form, but it does exclude the modeling of one important type of algorithm, those using a blocked structure. This limitation will be removed in the near future.

Without loss of generality it is assumed that arrays are arranged in memory as in the Fortran language, with the leftmost dimension a contiguous memory region.

Reiterating, the total number of loops is n, and the total number of dimensions of an array being considered is m. For the loop at level i, its loop variable is called j_i and its total number of iterations is $\mathcal{N}_i$. For a reference R, $\gamma_k(R)$ is the loop whose variable is referred to by dimension k of the reference. It follows that $j_{\gamma_k(R)}$ is the loop variable associated with this loop.

4 Preliminaries

Given a loop nesting as defined in the previous section, the model performs a number of preliminary actions before it attempts evaluation. Firstly each reference is translated into a secondary form that makes modeling its effects easier, then all references are divided into classes, discarding references which are redundant. Finally the reuse source of each reference is identified.

4.1 Linear Forms

The form of an array reference as defined in Section 3 is based on the individual dimensions in the array being accessed. When modeling a reference's behavior it is useful to use a notation based on loop variables. For example, the array reference $X(j_1, j_3)$ is equivalent to $X_0 + j_1 + Mj_3$, where X_0 is the base address of array X, and M its leading dimension. Such an expression is a *linear form*, the general form being,

$$\mathcal{B} + \mathcal{A}_1 j_1 + \ldots + \mathcal{A}_n j_n, \tag{1}$$

where $\mathcal{B}$ and all $\mathcal{A}_x$ are constants. The base address of the array and the β_k values combine to form $\mathcal{B}$. The $\mathcal{A}_x$ values are derived from the loop multipliers α_k and the dimensions of the array. Expanding the following equivalence allows each constant in a reference's linear form to be found (where the function $\dim(X, i)$ gives the number of elements in dimension i of array X),

$$\begin{aligned} &X(\alpha_1 j_{\gamma_1} + \beta_1, \ldots, \alpha_m j_{\gamma_m} + \beta_m) \\ &\qquad \equiv X_0 + \sum_{k=1}^{m} \left(\left(\prod_{i=1}^{k-1} \dim(X, i) \right) (\alpha_k j_{\gamma_k} + \beta_k) \, \mathcal{E}_X \right) \end{aligned}$$

The size of the elements being referred to is also included when calculating $\mathcal{B}$ and $\mathcal{A}_x$. Therefore evaluating a linear form for a particular set of values of j_i gives the actual memory location of the element being referred to.

4.2 Translation Groups

When considering cross interference it is important that redundant interference is not included; when two references both interfere with a potentially reusable data element, only a single miss occurs. To solve this problem all array references in the loop being considered are grouped into separate classes, or *translation groups*. The interference caused by each group of references is then considered as a whole. Since each group accesses the cache in different patterns, redundant interference is kept to a minimum.

Two references are *in translation* if their reference patterns are identical, and always a constant distance apart in the cache. For this to be true both references must share the same values of $\mathcal{A}_i$ for $i \leq n$, only their $\mathcal{B}$ values may differ. This can be formalized for two references R and R′,

$$\text{inTranslation}(R, R') = (\forall i : i \leq n) \left(\mathcal{A}_i(R) = \mathcal{A}_i(R') \right).$$

This definition is used to sort all references in a loop nest into separate translation groups. As few groups as possible are formed, such that each reference in the group is in translation with all others members of the group. For example, the four references shown in Figure 1 are sorted into two translation groups, $\{A(j_3, j_1), B(j_3, j_1)\}$, and $\{C(j_3), C(j_3 + 1)\}$.

Some of the references in a loop nesting are discarded before the evaluation starts. When two references in the same translation group always access the same cache lines only the misses due to one of them is evaluated. This removes redundant cache misses and simplifies the calculations involved. The assumption is made that there will be no interference misses in a single iteration of the innermost loop, an assumption also made by McKinley et al. [9] and supported by Lam et al. [8].

This pruning of the references is straightforward to achieve. For each translation group the $\mathcal{B}$ parameter of each reference is used to detect whether it is close enough to another reference to be discarded. For two references R and R′ if $|\mathcal{B}(R) - \mathcal{B}(R')| < \mathcal{L}$ then one of them is discarded.

4.3 Self Or Group Dependence?

The source of any temporal reuse achieved by each reference must be identified before the number of cache misses can be evaluated. As noted in Section 2, for any reference R there is a single reference $\vec{R}$ in the same loop nesting on which it depends for reuse. This reference $\vec{R}$ is the source of all temporal reuse exploited by R, and is called R's *reuse dependence*.

There are two types of reuse dependence: a *self dependence* in which R reuses data elements that it has previously accessed itself, or a *group dependence* in which R reuses elements that a different reference has accessed. In the case of a self dependence $R = \vec{R}$, while for a group dependence $R \neq \vec{R}$.

Example. Looking again at the example in Figure 1, the reference $A(j_3, j_1)$ exhibits a self dependence, each iteration of loop 2 references exactly the same elements as every other iteration of the loop. As an example of group dependence there is the reference $C(j_3)$, each data element that it uses was accessed by reference $C(j_3 + 1)$ on the previous iteration of loop 3. Interestingly, $C(j_3)$ also has the possibility of a self dependence, with reuse able to occur on the outer two loops.

The possibility of multiple dependences complicates the task of finding $\vec{R}$; only one dependence is actually exploited, the one that referenced the data most recently. The length of time between a data element being accessed by a reference, and it subsequently being reused by R, is called the *reuse distance* of the dependence. It is measured in iterations of the innermost loop. Finding the dependence with the smallest reuse distance finds the actual source of reuse. For example, reference $C(j_3)$ has a reuse distance of one iteration of loop 3, and $A(j_3, j_1)$ a reuse distance of $\mathcal{N}_3$ iterations of loop 3.

Calculating the reuse distance of a self dependence is trivial, simply multiply the sizes of the loops inside the level at which reuse occurs (defined as loop l, see Section 5). The reuse distance of a group dependence is more complicated and is found by adding the reuse distance for each dimension of the array being accessed. Given that both R and R′ refer to the same array, and that they are in the same translation group, the reuse distance between them $\mathrm{dist}(R, R')$ is given by,

$$\mathrm{dist}(R, R') = \begin{cases} \prod_{l<i\leq n} \mathcal{N}_i & \text{if } R = R'; \\ \sum_{1\leq k\leq m} \left(\left(\prod_{\gamma_k(R)<i\leq n} \mathcal{N}_i \right) \times \mathrm{dist}_k(R, R') \right) & \text{if } R \neq R, \end{cases}$$

$$\mathrm{dist}_k(R, R') = \begin{cases} \infty & \text{if } \delta_k(R, R') < 0 \text{ or } \delta_k(R, R') \geq \mathcal{N}_{\gamma_k(R)}; \\ \mathcal{N}_{\gamma_k(R)} & \text{if } \delta_k(R, R') = 0 \text{ and } R \prec R'; \\ \delta_k(R, R') & \text{otherwise,} \end{cases}$$
$$\delta_k(R, R') = \beta_k(R') - \beta_k(R). \tag{2}$$

The function dist_k calculates the reuse distance for dimension k of the references, in terms of the loop variable referenced by this dimension (j_{γ_k}). It recognizes three different situations, dependent on δ_k, the distance between the references in dimension k,[2]

Case 1: the distance between the references is either negative or greater than the size of the loop. In either case the dependence R' can't be exploited, hence the reuse distance is infinite,

Case 2: the distance is zero, but the reference for which a dependence is being sought, R, occurs before R' in the loop body (this is what the $R \prec R'$ notation means). Here there is the possibility for reuse, but not until the *next* complete set of iterations of loop γ_k,

Case 3: reuse occurs within this loop, the distance is simply the distance between the two references in this dimension, $\delta_k(R, R')$.

When finding a reference's reuse dependence only those references with which it is in translation are examined. Whichever has the smallest reuse distance is taken as the source of any temporal reuse. This leads to a definition of $\vec{R}$ as follows, given that G is the translation group that R and $\vec{R}$ belong to,

$$(\exists \vec{R} : G)(\forall R' : G)(\mathrm{dist}(R, \vec{R}) \leq \mathrm{dist}(R, R')).$$

This means that in the example case of reference $C(j_3)$ above, the dependence chosen is the group dependence with a reuse distance of one iteration, rather than the self dependence which has reuse distance $\mathcal{N}_3$.

4.4 The Evaluation Procedure

After creating each translation group and identifying the reuse dependence of each reference in these groups, the model is then used to evaluate the number of cache misses for each reference in all translation groups.

Step 1: Sort the list of references in the loop nesting into translation groups, as described by Section 4.2.

Step 2: For each reference R in each translation group, find its reuse dependence $\vec{R}$. The definition of $\vec{R}$ was given in Section 4.3.

Step 3: For each translation group G, for each reference $R \in G$, calculate the number of cache misses sustained by R given that it is dependent on $\vec{R}$ for any temporal reuse achieved.

[2] Since group dependence is only considered among references in the same translation group, only the β_k parts of the index expressions need be looked at.

Of this procedure **Step 3** is the most complex, and the bulk of the paper is devoted to the techniques used. These techniques are split into two sections: Section 5 describes how self dependence reuse is modeled, Section 6 examines group dependence reuse.

5 Self Dependence Reuse

An array reference can achieve some level of temporal self reuse only if one of the loop variables, j_i, is not included in the reference. This means that any iteration of loop i references exactly the same elements of the array as any other iteration. For example, reference $A(j_3, j_1)$ in Figure 1 does not include loop variable j_2, meaning that reuse occurs on loop 2.

The loop variable j_i not being included in a reference is equivalent to the coefficient $\mathcal{A}_i$ of the reference being zero. This gives the innermost loop on which reuse occurs for a particular reference, defined as loop l,

$$l = \max\left\{i \mid i \leq n,\, \mathcal{A}_i = 0\right\}, \tag{3}$$

If none of $\mathcal{A}_i$ are zero then no temporal self reuse occurs, and by convention $l = 0$; in this case it is still possible that some degree of spatial reuse may be achieved (see Section 5.1.2). If temporal reuse does occur, the *Theoretical Reuse Set* (abbreviated as TRS) of the reference is next identified. For a reference R, its TRS contains all array elements that R could possibly reuse, assuming that the size of the cache is unbounded. Since reuse occurs on loop l, its TRS contains all elements accessed by the loops inside level l. This is given by,

$$\mathrm{TRS}_l(R) = \{R\}\left(0 \leq j_i < \mathcal{N}_i\right)_{l<i\leq n}, \tag{4}$$

However, the model does not need to expand a reference's TRS, only to calculate the number of array elements that it contains, defined as,

$$\|\mathrm{TRS}_l(R)\| = \prod_{l<i\leq n} \mathcal{N}_i. \tag{5}$$

The notation $\|\mathrm{TRS}_l(R)\|$ stands for the size of the TRS of reference R, defined on loop l.

The size of a reference's TRS allows the number of compulsory misses of that reference to be found. For the first iteration of the reuse loop l, no temporal reuse can occur since the TRS has not previously been accessed. For reference R the number of temporal compulsory misses is given by,

$$\left(\prod_{1\leq i<l} \mathcal{N}_i\right) \times \|\mathrm{TRS}_l(R)\|, \tag{6}$$

No spatial effects are included in this expression, these are given later (see Section 5.1.2). Also note that if no reuse occurs (i.e., $l = 0$) the reuse set contains all accessed elements, and therefore the number of compulsory misses is the total number of references in all loops.

Example. Identifying the TRS of reference $A(j_3, j_1)$, and its size, is straightforward. The loop on which reuse occurs was identified above as loop 2, therefore,

$$\begin{aligned} \mathrm{TRS}_2(A(j_3, j_1)) &= \{A(0, j_1), A(1, j_1), \ldots, A(\mathcal{N}_3 - 1, j_1)\}, \\ \|\mathrm{TRS}_2(A(j_3, j_1))\| &= \mathcal{N}_3. \end{aligned}$$

A TRS is defined without regard for the physical structure of the cache, as such there is no guarantee that any of its elements will actually be reused. The rest of this section is devoted to evaluating the number of cache misses for a self dependent reference, given that temporal reuse occurs on loop l.

5.1 Self-Interference

This section shows how self-interference is evaluated, the effect occurring when a reference obstructs its own reuse. The calculation is split into two halves, the first dealing with temporal reuse, the second with spatial reuse. These are later combined with the effects of compulsory misses to give the number of self interference and compulsory misses for a single reference.

5.1.1 Temporal Self-Interference

Temporal self-interference occurs when more than one element of the TRS maps to the same cache location. When this happens none of the elements mapping to the location can be reused since they continually overwrite each other. To model this phenomenon it is necessary to map the TRS into the cache removing any elements that occupy the same cache location. This forms another set, the *Actual Reuse Set*, or ARS, containing only those elements in the cache that are reused with a particular cache size. Subtracting the size of this set from the size of the associated TRS gives the number of temporal misses on each iteration of the reuse loop. For a reference R, this is,

$$\|\mathrm{TRS}_l(R)\| - \|\mathrm{ARS}_l(R)\|. \tag{7}$$

The form of an ARS is usually very regular, allowing it to be characterized by several parameters. The first parameter S is the average size of each region, or *interval*, in the ARS. The second parameter σ is the average distance in the cache from the start of one such interval to the start of the next. The next parameter B is the average number of distinct cache lines in an interval of size S[3]. Adding two final parameters, the position of the first interval ϕ, and the number of intervals in total N, allows an ARS to be defined as follows,

$$\mathrm{ARS}_l(R) = \langle S, \sigma, B, N, \phi \rangle . \tag{8}$$

Figure 4 shows these parameters in relation to the cache, the black rectangles form the ARS. The actual number of elements contained by an ARS is then,

$$\|\mathrm{ARS}_l(R)\| = \frac{NS}{\mathcal{E}_R}. \tag{9}$$

[3] It is not always the case that $B = S/\mathcal{L}$ since the elements in an interval are not necessarily continuous.

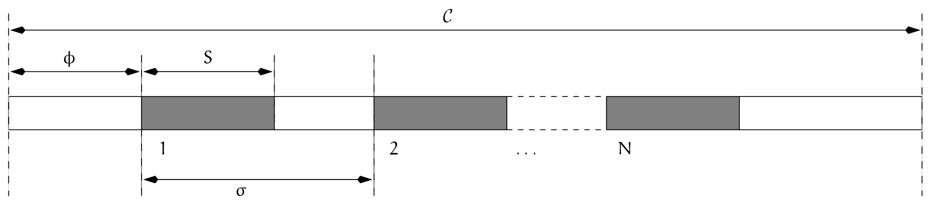

Figure 4: ARS parameters

It follows from this and from (7), that only two of the parameters, S and N, are required to calculate the temporal self-interference on a reference. The other parameters are required when evaluating the level of cross-interference (see Section 5.2).

Finding the structure of a reference's ARS is a two step process; first the reference's footprint in the array it accesses is found, then this footprint is mapped to a particular cache size. The footprint is described by the size, spacing and number of continuous memory regions accessed by a reference. These values are given by S^t, σ^t and N^t, the "theoretical" values of S, σ and N. After being mapped into the cache these theoretical values lead to the five parameters defining the ARS.

Finding S^t, σ^t and N^t. The first part of the procedure is relatively straightforward, the loops inside the reuse loop (i.e., loops $l+1, \ldots, n$) are sorted so that the coefficients of the reference are in size order, from smallest to largest, ignoring those that are zero. The ordered loops are numbered from one to p, so that the loop at position p has the largest coefficient. This ordering is defined by the values $\tau_1 \ldots \tau_p$, such that,

$$0 < \mathcal{A}_{\tau_1} \leq \ldots \leq \mathcal{A}_{\tau_p},$$

that is, τ_k identifies the loop with the k'th smallest coefficient.

For each level k, from $1 \ldots p$, a working value of S^t is maintained, S^t_k. This reflects the size of the continuous region on the loop referred to by level k. The initial value S^t_0 is the size of a single element, $\mathcal{E}$.

$$S^t_k = \left(\prod_{1 \leq i \leq k} \mathcal{N}_{\tau_k} \right) \mathcal{E}$$

For each level k considered, if the coefficient of the loop is equal to the current size of the continuous region, the size of the region is multiplied by the number of iterations in the loop. On the other hand, if the current size doesn't match the next coefficient then the largest continuous region has been found and the process is complete. The level at which this happens is labeled υ,

$$\upsilon = \max \left\{ k \;\middle|\; 1 \leq k \leq p,\, \mathcal{A}_{\tau_k} = S^t_{k-1} \right\}.$$

Once υ is known it is possible to find the values of S^t, σ^t and N^t,

$$
\begin{aligned}
S^t &= S^t_\upsilon, \\
\sigma^t &= \begin{cases} \mathcal{A}_{\tau_{\upsilon+1}} & \text{if } \upsilon < p; \\ S^t_p & \text{if } \upsilon \geq p. \end{cases} \\
N^t &= \begin{cases} \mathcal{N}_{\tau_{\upsilon+1}} & \text{if } \upsilon < p; \\ 1 & \text{if } \upsilon \geq p. \end{cases}
\end{aligned} \tag{10}
$$

If all non-zero coefficients have been considered, i.e., $\upsilon = p$, then only one continuous region exists, and the definitions of σ^t and N^t are altered accordingly.

It should be noted that if more than one unused coefficient exists, i.e. when $\upsilon < p - 1$, the footprint will be inaccurate. Temam et al. [13] have an approximate method which could be used in these cases, but in practice they rarely occur[4].

Example. Consider the reference $X(j_2, j_1)$. Assuming that for this array $\mathcal{E} = 1$, the linear form is simply $X_0 + \mathcal{N}_2 j_1 + j_2$. The smallest coefficient, $\mathcal{A}_2$, is equal to 1—the value of S^t_0. This means that all $\mathcal{N}_2$ iterations of loop 2 form a single region containing $\mathcal{N}_2$ elements, i.e. $S^t_1 = \mathcal{N}_2$. The next (and final) coefficient, $\mathcal{A}_1$, is $\mathcal{N}_2$, which is equal to S^t_1. Thus, completely executing loop 1 adds $\mathcal{N}_1$ copies of the current region, giving the total size of the continuous region S^t as $\mathcal{N}_1 \mathcal{N}_2$.

Since all coefficients have been used there is only one interval, $\sigma^t = \mathcal{N}_1 \mathcal{N}_2$, and $N^t = 1$. This is reassuring since it is self-evident that accessing all elements of an array (a matrix in this case) means that the continuous region will simply be the array itself.

Mapping S^t, σ^t and N^t into the cache. After identifying the array footprint of the reuse set, the three theoretical values are known. The final step in creating the ARS is to map these three parameters into the physical layout of the cache. After removing the elements that collide whatever is left may be reused.

The method of mapping the footprint into the cache uses a recursive procedure presented by Temam et al. [13]. It progressively subdivides the area of the cache that it is examining so that each area has a structure similar to all other areas. This structure results from the N^t intervals mapping into the cache. Depending on the size of the cache, the intervals may wrap around the end of the cache, possibly overwriting previous intervals.

At each level of recursion k, areas of size σ_k are mapped into a single area of size σ_{k-1}, illustrated in Figure 5 for part of the cache. The following recurrence relation defines the sequence of σ_k values, representing how the array footprint intervals map into a cache of size $\mathcal{C}$,

$$\sigma_0 = \mathcal{C}, \qquad \sigma_1 = \sigma^t, \qquad \sigma_k = \sigma_{k-1} - \sigma_{k-2} \bmod \sigma_{k-1}, \qquad \text{for } k \geq 0. \tag{11}$$

[4] This problem stems from the S, σ representation used, which is unable to model footprints whose continuous regions are separated by more than one value of σ^t.

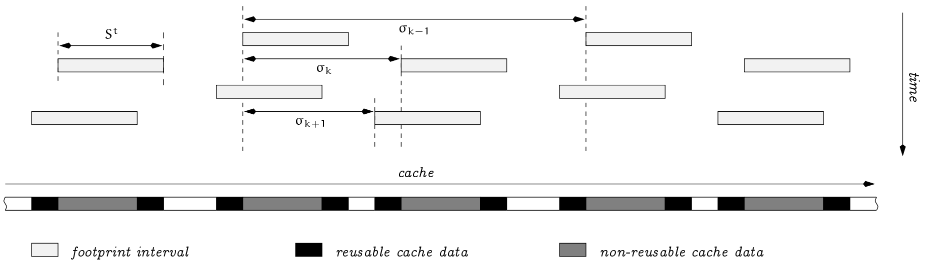

Figure 5: Example of mapping array footprint intervals into the cache.

Each area is continually subdivided until a level s where either all N^t of the footprint intervals have been mapped into the cache, or overlapping occurs between intervals, that is,

$$s = \min\left\{k \;\middle|\; \left\lfloor\frac{\sigma_0}{\sigma_k}\right\rfloor \geq N^t \quad \text{or} \quad \tilde{\sigma}_k < S^t\right\}, \tag{12}$$

$$\tilde{\sigma}_k = \min(\sigma_k, \sigma_{k-1} - \sigma_k). \tag{13}$$

The $\tilde{\sigma}_k$ expression is used due to the "circular" nature of caches; σ_k is the distance moving *forwards* in the cache, from one interval to the next. There may be a closer interval in the other direction, possibly overlapping with the interval being considered.

Once the subdivision process has stopped at level s, the values of σ_0, σ_{s-1} and σ_s are used to find the ARS parameters S, σ, B and N. These are then used to evaluate the actual size of the footprint using (9). At level s, the cache has been divided into σ_0/σ_{s-1} areas, each of size σ_{s-1}. Within each area there are a certain number of intervals, each of which is $\tilde{\sigma}_s$ cache locations from the next. The number of intervals within any one area will be one of two values, either n_s, or $n_s + 1$, this is due to the number of intervals N^t not usually being exactly divisible by the number of areas. The number of areas of each type and the value of n_s are found using,

$$n_s = \left\lfloor\frac{N^t}{\sigma_0/\sigma_{s-1}}\right\rfloor, \text{ and} \tag{14}$$

$$r = \left\lfloor N^t - n_s\left(\sigma_0/\sigma_{s-1}\right)\right\rfloor. \tag{15}$$

This means that there are r areas, of size σ_{s-1}, containing $n_s + 1$ intervals of size S^t, and $(\sigma_0/\sigma_{s-1} - r)$ areas containing n_s intervals. At this point the values of N, σ, and ϕ can be found,

$$\sigma = \begin{cases} \sigma_s & \text{if } s = 1; \\ \sigma_{s-1} & \text{if } s \neq 1. \end{cases} \tag{16}$$

$$N = \begin{cases} n_s & \text{if } s = 1; \\ \lfloor\sigma_0/\sigma s - 1\rfloor & \text{if } s \neq 1. \end{cases} \tag{17}$$

$$\phi = \mathcal{B} \bmod \mathcal{C}. \tag{18}$$

The two cases are necessary to handle the situation where all intervals map into the cache without overflowing, i.e. when $s = 1$.

The only parameters left to find are S, the average size of each interval, and B, the average number of cache lines in each interval. From (9) it follows that S is the total size of the ARS divided by the number of intervals it contains, N. Since N is given by (17) above, it is only necessary to calculate the total size of the ARS. This calculation is split into two parts, one finds the total size of the areas containing n_s intervals, the other of those areas with $n_s + 1$ intervals. The whole equation for S is as follows,

$$S = \frac{\min\left(\sigma_0, f_r(n_s) \times (\sigma_0/\sigma_{s-1} - r) + f_r(n_s + 1) \times r\right)}{N} \tag{19}$$

where $f_r(x)$ gives the amount of data that can be reused in an area of size σ_{s-1}, given that there are x intervals of size S in the area, each $\tilde{\sigma}_s$ from the next. This function is defined as,

$$f_r(x) = \begin{cases} f_{r'}(x) & \text{if } f_l(x) < \sigma_{s-1}; \\ \left(f_{r'}(x) - 2\left(f_l(x) - \sigma_{s-1}\right)\right)^+ & \text{if } f_l(x) \geq \sigma_{s-1}, \end{cases} \tag{20}$$

$$f_{r'}(x) = \begin{cases} xS^t & \text{if } x = 1 \text{ or } \tilde{\sigma}_s \geq S^t; \\ 2\tilde{\sigma}_s + (x-2)^+ \left(S^t - 2\tilde{\sigma}_s\right)^+ & \text{otherwise,} \end{cases}$$

$$f_l(x) = S^t + (x-1)^+ \tilde{\sigma}_s. \tag{21}$$

The condition in $f_{r'}(x)$ detects overlapping between the intervals within a single area, if overlapping does occur then it is only possible to reuse $2\tilde{\sigma}_s$ cache locations. Alternatively if there is no overlapping then xS^t elements may be reused (i.e., all of them). The condition in the definition of $f_r(x)$ checks whether the footprint overlaps with the intervals in the next area; in which case the overlapping locations are removed. The function $f_l(x)$ gives the "extent" of the footprint in the area being considered, this is the distance from the start of the first interval to the end of the last one.

The final parameter of the ARS, B, is calculated from the value of S and its theoretical value S^t defined in (10). Each interval of size S contains on average S/S^t intervals of size S^t, while each theoretical interval of size S^t inhabits $\lceil S^t/\mathcal{L} \rceil$ whole cache lines. Therefore,

$$B = \left\lceil \frac{S^t}{\mathcal{L}} \right\rceil \frac{S}{S^t}. \tag{22}$$

5.1.2 Spatial Self-Interference

The previous section showed how the number of temporal misses suffered by a TRS is evaluated, assuming that each element not in the ARS generates a single cache miss per access. However, since the cache accesses memory in units of lines, not elements, the effects of spatial reuse must also be included to give the actual number of cache misses.

The best use of the cache is generally achieved when the elements of an array are accessed sequentially. This occurs when the coefficient of the innermost loop, $\mathcal{A}_n$, is the size of a single element. Since elements from the same cache line are being accessed on successive iterations of the innermost loop the probability of the line being overwritten is extremely small. This means that each line referenced will usually only generate a single cache miss. In this case the actual number of misses is found by dividing the number of temporal misses by the number of elements held in each cache line.

Example. Consider reference $X(j_2, j_1)$ in Figure 6. This reference uses the variable of the innermost loop, j_2, to index the innermost dimension of the array. Due to the innermost dimension being stored in contiguous memory locations the array is traversed sequentially, therefore $\mathcal{N}_2/(\mathcal{L}/\mathcal{E}_X)$ cache misses are needed to load each row of the array into the cache.

```
DO j1 = 0, N1 - 1
  DO j2 = 0, N2 - 1
    X(j2,j1) = Y(j1,j2)
  ENDDO
ENDDO
```

Figure 6: Spatial reuse example

For a reference R, some level of spatial reuse can occur only when one or more of its coefficients $\mathcal{A}_i$ is less than the size of a cache line. This gives the innermost loop on which spatial reuse occurs, l_{spat}, as,

$$l_{spat} = \max \{ i \mid 0 < \mathcal{A}_i < \mathcal{L} \} .$$

When this is not the innermost loop (i.e. $l_{spat} < n$) then the reuse distance of the spatial reuse (defined as the number of iterations of loop n between a cache line being accessed twice by R) is larger than when the reuse occurs on the innermost loop. It follows that the possibility of the spatial reuse being subject to interference is also greater. When this interference is caused by R itself it is called *spatial self-interference*.

If l_{spat} is undefined, due to none of the reference's $\mathcal{A}_i$ values being less than the size of a cache line, there is no possibility of spatial reuse occurring and the actual number of cache misses is the same as the number of temporal misses.

Example. For reference $Y(j_1, j_2)$ in Figure 6, spatial reuse occurs on loop 1[5]. An iteration of loop 1 may reference the same cache lines accessed by the previous iteration. However, between a cache line being accessed twice, $\mathcal{N}_2 - 1$ other elements are also referenced (since spatial reuse doesn't occur on the innermost loop). If any of these elements map to a line already being used self-interference will occur.

To find the actual number of cache misses sustained, the number of temporal misses is multiplied by a factor M_{spat}. This factor encapsulates all spatial self-interference effects of the reference being considered. It is formed by combining the compulsory spatial miss-ratio C_{spat} with the probability of spatial self-interference P_{spat},

$$M_{spat} = \max (C_{spat}, P_{spat}) . \tag{23}$$

[5] Assuming that the width of each column of array Y is greater than $\mathcal{L}$.

This is the maximum of the two terms; if there is no possibility of interference then the number of temporal misses is multiplied by the compulsory miss-ratio. If the probability of interference outweighs the compulsory miss-ratio then this probability gives the actual number of cache misses.

The compulsory spatial miss-ratio is defined as the reference's smallest non-zero coefficient divided by the size of each cache line. The smallest coefficient is the same as the minimum access stride of the reference; dividing this by the line size gives the ratio of compulsory misses to accesses. The minimum stride of a reference is given by,

$$\text{stride} = \min \left\{ \mathcal{A}_i \mid 1 \leq i \leq n,\ \mathcal{A}_i \neq 0 \right\}. \tag{24}$$

If the stride is greater than the line size then no two referenced elements occupy the same cache line, and therefore every access generates a cache miss—a compulsory miss-ratio of 1.

$$C_{\text{spat}} = \frac{\min\left(\text{stride}, \mathcal{L}\right)}{\mathcal{L}} \tag{25}$$

When the minimum stride is equal to the size of an element $\mathcal{E}$ the compulsory miss-ratio is equal to the lower bound $\mathcal{E}/\mathcal{L}$, as in the first example in this section.

Example. Reference $X(j_2, j_1)$ in Figure 6 gives $\mathcal{N}_1\mathcal{N}_2$ temporal misses (it has no temporal reuse). Since $X(j_2, j_1) \equiv X_0 + \dim(X, 1)\mathcal{E}_X j_1 + \mathcal{E}_X j_2$ it follows from (24) and (25) that,

$$C_{\text{spat}} = \frac{\min\left(\mathcal{E}_X, \dim(X, 1)\mathcal{E}_X, \mathcal{L}\right)}{\mathcal{L}} = \frac{\mathcal{E}_X}{\mathcal{L}}.$$

Assuming that $P_{\text{spat}} = 0$, then $M_{\text{spat}} = C_{\text{spat}}$ by (23), and the actual number of cache misses due to reference $X(j_2, j_1)$ is $M_{\text{spat}}\mathcal{N}_1\mathcal{N}_2$, or $\mathcal{N}_1\mathcal{N}_2\mathcal{E}_X/\mathcal{L}$.

Finding P_{spat} of a reference. The probability of spatial interference occurring for a particular reference depends almost completely on a single parameter: the layout of the data elements accessed by the reference between each iteration of the spatial reuse loop. If these elements map into the cache such that more than one element occupies a particular cache line, interference occurs.

One special case is easily identifiable, when the reference contains no instances of the loop variables $j_{l_{\text{spat}}+1}, \ldots, j_n$. In this case no other elements are accessed by the reference between using two elements in the same cache line. Therefore no spatial self-interference occurs and $P_{\text{spat}} = 0$.

The other case, when one or more of these loop variables is used, is harder to classify. The method used is in some ways similar to the method of calculating temporal self-interference in that they both use the same recursive subdivision of the cache to detect interference, (11) and (12) (see Section 5.1.1). After subdividing the cache, the resulting σ_k values are used to find the level of spatial interference.

As in the temporal reuse calculation, the first step is to identify the initial data layout, specified by the same parameters as for temporal reuse, S^t, σ^t and

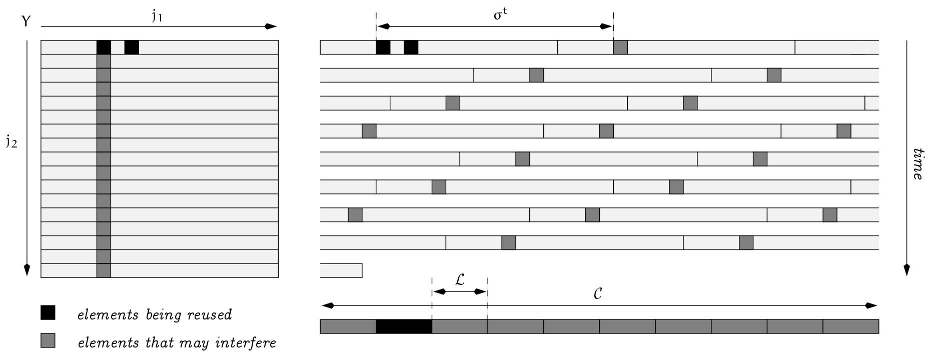

Figure 7: Example of spatial reuse from $Y(2j_1, j_2)$

N^t (see Page 13). In this case the objective is to characterize the layout of the array elements referenced by a single iteration of the spatial reuse loop.

In the majority of cases there will only be a single non-zero coefficient referring to the loops $l_{spat}+1$ through n; this allows the layout to be characterized exactly; σ^t is the value of this coefficient and N^t is the number of elements referenced by one iteration of the reuse loop. Since interference between cache lines is being examined, S^t is always an integer number of lines, the number of lines needed to hold one data element. This gives the three parameters needed to map the spatial characteristics of a reference into the cache[6]:

$$\begin{aligned}
\sigma^t &= \min\left\{\mathcal{A}_i \mid l_{spat} < i \le n,\ \mathcal{A}_i \neq 0\right\},\\
N^t &= \prod_{\substack{l_{spat} < i \le n \\ \mathcal{A}_i \neq 0}} \mathcal{N}_i,\\
S^t &= \mathcal{L}\left\lceil \frac{\mathcal{E}}{\mathcal{L}} \right\rceil.
\end{aligned}$$

Example. To illustrate the problem consider Figure 7. This shows how the array reference $Y(2j_1, j_2)$ maps into the cache, and highlights the elements that may interfere with the spatial reuse of elements $Y(4,0)$ and $Y(6,0)$ (i.e. $j_1 = 2$ and $j_1 = 3$.) For example if any of $Y(4,1), \ldots, Y(4, \mathcal{N}_2)$ map to the same cache line as $Y(4,0)$ spatial reuse of this element is prevented.

Here σ^t, defined as the distance between each element accessed by a single iteration of loop l_{spat}, is the width of the matrix. Counting the number of blocks for a single value of j_1 gives N^t, the number of elements accessed by each iteration of loop l_{spat}. In this case N^t equals the number of rows in the matrix.

The level of spatial interference P_{spat} is found by using the cache subdivision process defined by (11) and (12) to map the layout of the elements accessed by a single iteration of loop l_{spat} into the cache. The amount of overlapping between these elements leads to P_{spat}. When mapping σ^t, N^t, and S^t into the cache the

[6] When there is more than one non-zero coefficient inside the reuse loop the definitions of σ^t, N^t, and S^t may be inaccurate. This happens very rarely; never with two-dimensional matrices, seldom with higher-order matrices. We aim to address this in the future.

subdivision process again has initial values of $\sigma_0 = \mathcal{C}$ and $\sigma_1 = \sigma^t$. Due to S^t being greater than or equal to the size of a cache line, the subdivision will not stop until either all N^t elements have been used, or overlapping between cache lines occurs, (12).

After stopping at level s the cache has been divided into a number of areas of size σ_{s-1}. The number of elements in each area is again given by (14), and the number of areas of each type by (15). For each type of area the degree of spatial reuse is calculated, this is the number of elements in the area that can be reused divided by the total number. Combining these values, weighted to the number of areas with n_s elements and the number with $n_s + 1$, gives P_{spat},

$$P_{spat} = 1 - \begin{cases} 1 & \text{if } (\forall i : i > l_{spat})\,(\mathcal{A}_i = 0); \\ f_s(n_s)\dfrac{(\lfloor \sigma_0/\sigma_{s-1} \rfloor - r) \times n_s}{N^t} \\ \quad + f_s(n_s+1)\dfrac{r \times (n_s + 1)}{N^t} & \text{otherwise.} \end{cases} \tag{26}$$

with r defined in (15) and n_s in (14); $\lfloor \sigma_0/\sigma_{s-1} \rfloor$ is the number of areas the cache has been divided into. The value of P_{spat} allows the total number of misses due to spatial interference to be predicted using (23).

The function $f_s(x)$ calculates the degree of spatial reuse for an area of size σ_{s-1} containing x elements, each a distance $\tilde{\sigma}_s$ from the surrounding elements.

Deriving $f_s(x)$. This is a ratio; the number of cache locations in an area in which an element can be the sole occupier of a cache line divided by the number of cache locations in which interference occurs.

Two special cases of total or zero reuse can be identified immediately. If there is only one element in the area (i.e. $x = 1$), or the distance between two elements is as large as a cache line (i.e. $\tilde{\sigma}_s \geq \mathcal{L}$) there is no possibility of interference and total spatial reuse occurs: $f_s(x) = 1$. Conversely, when the distance between each element is zero (i.e. $\tilde{\sigma}_s = 0$) interference occurs between all elements in the area and no spatial reuse is possible: $f_s(x) = 0$.

The cases left, more than one element in the area, with a distance between them of less than a cache line but greater than zero, can be divided into two sub-problems. Both are tackled by calculating the number of positions in a cache line that an element could occur, and subsequently be reused.

Elements at either end of an area. Examples of this case are shown in Figure 8. Each diagram depicts the second black element attempting to reuse the first. The white elements are those that are referenced between accessing the two black elements, if any of the white elements are in the same cache line as either of the black elements the reuse is prevented.

As can be seen the number of positions that allow an element to be reused depends on whether the distance between successive elements in the area, σ_s, is negative or positive[7]. When it is positive the number of

[7] Actually σ_s is never less than zero. When $(\sigma_{s-1} - \sigma_s) < \sigma_s$ it is useful to think of σ_s as being negative. This is related to the definition of $\tilde{\sigma}_s$ in (13).

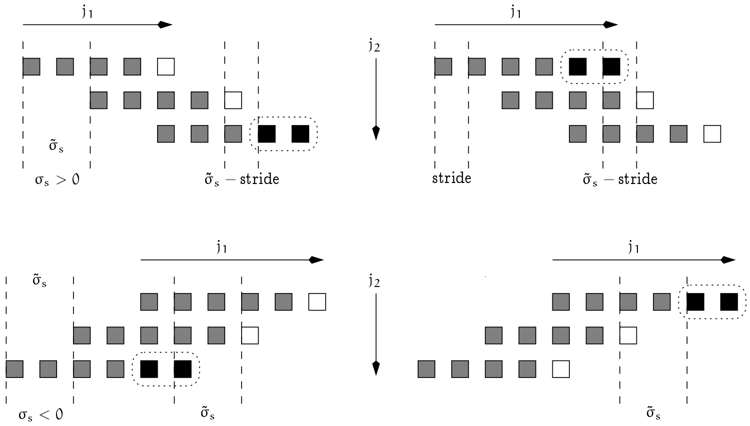

Figure 8: Reuse of elements at the edge of an area.

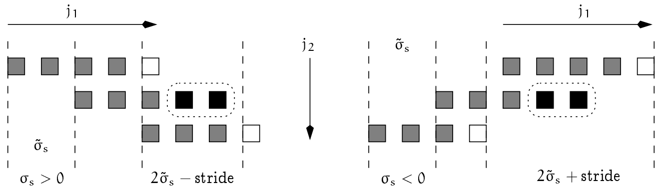

Figure 9: Reuse of elements in the middle of an area.

reusable positions is smallest, since the elements loaded between referencing the two black elements are moving towards the reused elements. The number of reusable positions for an element on the edge of an area is given by,

$$p_e = \begin{cases} \tilde{\sigma}_s - \text{stride} & \text{if } \sigma_s = \tilde{\sigma}_s; \\ \tilde{\sigma}_s & \text{if } \sigma_s \neq \tilde{\sigma}_s. \end{cases} \tag{27}$$

where the stride is defined in (24).

The ratio of reusable positions occurring at the edge of an area is found by dividing p_e by the size of a line, and multiplying by $2/x$, i.e. $(2/x)(p_e/\mathcal{L})$.

Elements in the middle of an area. This case is depicted in Figure 9, using the same conventions as in Figure 8. Each area contains $(x-2)^+$ elements that are not on the edge of the area; each has a neighbor on either side, $\tilde{\sigma}_s$ locations away. As before, when σ_s is positive the number of reusable positions is greatest, p_m is defined as the number of reusable positions,

$$p_m = \begin{cases} ((2\tilde{\sigma}_s - \text{stride}) - \mathcal{L})^+ & \text{if } \sigma_s = \tilde{\sigma}_s; \\ ((2\tilde{\sigma}_s + \text{stride}) - \mathcal{L})^+ & \text{if } \sigma_s \neq \tilde{\sigma}_s. \end{cases} \tag{28}$$

The size of a cache line is subtracted from the distance between the nearest two elements that could interfere with the reuse. This gives the number of positions in which neither white element can interfere.

Using p_m to form a ratio, weighted to the proportion of elements in the middle of the area, gives the actual reuse ratio, for this class of elements, i.e. $((x-2)^+/x)(p_m/\mathcal{L})$.

These definitions are correct when the distance between the first and last elements in the area $(x\tilde{\sigma}_s)$ is smaller than the size of the area, σ_{s-1}. When this is not the case the elements in one area may interfere with those in another. A factor to detect this and scale the reuse ratio by the level of inter-area interference is as follows,

$$\text{overlap} = \min\left(1, \frac{x\tilde{\sigma}_s - 2\left(x\tilde{\sigma}_s - \sigma_{s-1}\right)}{x\tilde{\sigma}_s}\right). \tag{29}$$

This is the full extent of the elements in the area $(x\tilde{\sigma}_s)$ with the number of non-reusable locations subtracted.

Combining the two special cases of total and zero reuse, with the expressions for the two element types and (29) gives the full definition of $f_s(x)$:

$$f_s(x) = \begin{cases} 1 & \text{if } x = 1 \text{ or } |\sigma_s| \geq \mathcal{L}; \\ 0 & \text{if } \tilde{\sigma}_s = 0; \\ \dfrac{2p_e + (x-2)^+ p_m}{x\mathcal{L}} \times \text{overlap} & \text{otherwise.} \end{cases} \tag{30}$$

5.1.3 Combining Temporal and Spatial Effects

The previous sections have shown how to separately calculate the number of misses incurred due to temporal and spatial interferences, (7) and (23), and due to compulsory misses, (6). Naturally, the idea is to combine these into a single value, giving the actual number of self-interference misses for any self-dependent reference. In fact it would be useful to develop two versions, one that includes compulsory misses and one that doesn't. Excluding compulsory misses is the most straightforward, for a reference R,

$$\text{self interference misses} = M_{\text{spat}} \times \left(\prod_{1 \leq i \leq l} \mathcal{N}_i\right) \times \left(\|\mathrm{TRS}_l(R)\| - \|\mathrm{ARS}_l(R)\|\right).$$

If compulsory misses are included the definition becomes slightly more complex, this is because the first iteration of the reuse loop is handled by the compulsory miss calculation. It follows that there are no interference misses on this iteration of loop l.

self interference and compulsory misses

$$\begin{aligned} &= \left(M_{\text{spat}} \times \left(\prod_{1 \leq i < l} \mathcal{N}_i\right) \times \|\mathrm{TRS}_l(R)\|\right) \\ &\quad + \left(M_{\text{spat}} \times \left(\prod_{1 \leq i < l} \mathcal{N}_i\right) \times (\mathcal{N}_l - 1) \times \left(\|\mathrm{TRS}_l(R)\| - \|\mathrm{ARS}_l(R)\|\right)\right). \end{aligned}$$

If no temporal reuse occurs (i.e. $l = 0$) then only compulsory effects are included, and the second term of the expression is ignored.

5.2 Cross-Interference

When there is more than one reference in a loop nesting, as there almost always will be, the other references will also interfere with any reuse occurring. This is called *cross interference*. It is modeled by comparing the ARS of the reference being considered with the *Actual Interference Set* (or AIS) of each translation group. The AIS contains all cache locations referenced by a translation group; any intersection between an AIS and the ARS represents possible cross-interference between that translation group and the reference. Sections 5.2.1 and 5.2.2 show how the definition of an ARS in Section 5.1.1 is adapted to form the basis of the AIS.

The cache misses caused by temporal cross-interference can be divided into two separate types,

Internal cross-interferences: this type of interference occurs between references in the same translation group. Since the distance between the two references is constant it is fairly straightforward to predict the level of interference.

External cross-interferences: interference that occurs between references in *different* translation groups. This is a more complex phenomenon since the reuse and interference sets are constantly changing in relation to one another. The model developed uses an approximate method that gives reasonably accurate results.

By calculating the number of cache misses due to cross-interference for a self-dependent reference and adding them to the number of self-interference misses (see Section 5.1.3) the total number of cache misses by the reference is found. For a reference R each translation group G is studied in turn. If R is a member of group G then internal cross-interference is assumed, otherwise external interference is evaluated. Section 5.2.3 gives the method used to calculate the level of internal interference, while Section 5.2.4 examines external interference.

Cross-interference can also disrupt the spatial reuse of a reference, especially if the spatial reuse distance is relatively large (see Section 5.1.2). Section 5.2.5 shows how the cache misses due to this effect are also included.

5.2.1 Interference sets

An AIS is very similar to an ARS, the only conceptual difference being that when creating an AIS any array elements that map to the same cache locations are not removed, as they are with an ARS. Although overlapping elements can't be reused, they can still interfere with other elements that could otherwise be reused.

The definition of a reference's AIS contains exactly the same parameters as in the definition of an ARS,

$$\mathrm{AIS}_l(R') = \langle S, \sigma, B, N, \phi \rangle$$

For a reference R, whose ARS is defined on loop l, the interference sets compared with this ARS are also defined on loop l. If an ARS and an AIS were defined on different loops they would not be comparable.

When computing the parameters of an AIS, the same recursive subdivision method is used, (given by (11), see Section 5.1.1). The B, N, σ and ϕ parameters are calculated in exactly the same way. The definition of S is changed, however, to meet the condition stated above of retaining those elements with identical cache locations. Instead of using the function f_r to find the size of each interval, a function f_i is used that calculates the size without removing any overlapped elements,

$$\begin{aligned} S &= \frac{\min\left(\sigma_0, f_i(n_s) \times \left(\left\lfloor \frac{\sigma_0}{\sigma_{s-1}} \right\rfloor - r\right) + f_i(n_s+1) \times r\right)}{N}, \\ f_i(x) &= f_l(x) - (x-1)^+ (\tilde{\sigma}_s - S^t)^+. \end{aligned}$$

The function f_i calculates the extent of the x intervals in the area, minus the gaps between each interval. This gives the total size of the area's footprint with overlapped cache locations being counted once. When calculating B the definition of S given above is used, not the definition for an ARS, (19).

Comparing the data layout represented by each AIS with that of each ARS allows cross interference to be detected. Wherever the two sets of cache locations intersect there is the possibility of interference.

5.2.2 AIS of a Translation Group

Most redundant cross interference is ignored by using each translation group as the cause of interference instead of each individual reference. References that are in translation, and therefore moving through the cache at a constant rate, are treated as a single cause of interference. For a reference with reuse on loop l, each translation group forms a single interference set, which is also defined on loop l, by combining the individual interference sets of all the references in the group. If the notation $\mathrm{AIS}^*_l(G)$ represents the combined AIS of a translation group G, then conceptually,

$$\mathrm{AIS}^*_l(G) = \mathrm{AIS}_l(G_1) \cup \mathrm{AIS}_l(G_2) \cup \ldots \cup \mathrm{AIS}_l(G_{\#G}),$$

where G_i is the i'th reference in that group and there are #G references in the group in total. The representation of an actual interference set, $\langle S, \sigma, B, N, \phi\rangle$, is too restrictive to accurately model the whole interference set of a translation group. To overcome this problem the combined interference set is represented as a vector of interference sets, each describing a separate continuous part of the cache.

The algorithm used to combine several interference sets is quite involved. The underlying idea is to split each interference set into as few parts as possible, so that each part models an area of the cache with a structure that can be described using the five parameters $\langle S, \sigma, B, N, \phi\rangle$. For example, consider Figure 10 in which two interference sets are being combined. One starts at cache

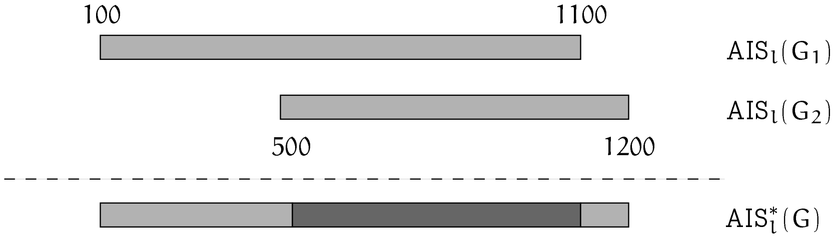

Figure 10: Example translation group interference set

position 100 and finishes at position 1100, while the other starts at position 500 and ends at 1200. The combined set has three parts, the first and last are portions of the original sets, and the middle part is made by combining the internal structure (i.e., the S and σ parameters) of the two original sets.

5.2.3 Internal Cross-Interference

Internal cross-interference on a reference is caused by its own translation group. Since the distance between references in the same translation group is constant, the translation group's interference set (excluding the reference itself) can be compared with the reference's reuse set very easily. The number of misses follows from the intersection between the two sets. Each interval in the two sets must be considered separately; the size of the intersection between two intervals, X and Y, is given by,

$$\begin{aligned} \|X \cap Y\| = &\min\left((\|Y\| - \delta_{X,Y})^+, \|Y\|\right) \\ &+ \min\left((\|X\| + \delta_{X,Y} - \mathcal{C})^+, \|Y\|\right), \end{aligned} \tag{31}$$

where $\|X\|$ is the size of interval X and $\delta_{X,Y}$ is the distance between the two intervals in the cache. This distance is defined as the number of cache locations traversed when moving forwards from X to Y. If ϕ_X is the position in the cache of interval X, then,

$$\delta_{X,Y} = \begin{cases} \phi_X - \phi_Y & \text{if } \phi_X - \phi_Y \geq 0; \\ \mathcal{C} + (\phi_X - \phi_Y) & \text{if } \phi_X - \phi_Y \ngeq 0. \end{cases} \tag{32}$$

The number of misses due to internal interference between two intervals is calculated by applying the degree of spatial reuse achieved by the reference to the size of the intersection between the two intervals. The spatial reuse of a reference is identified in Section 5.1.2, with M_{spat} defined in (23).

As each interference and reuse set is composed of more than one interval the total number of internal misses due to a single interference set is found from the cumulative intersection between all combinations of intervals in the two sets. The interference set of a translation group is a vector of actual interference sets, so the total number of internal misses on a reference R from the translation group G, such that $R \in G$, is as follows,

internal interference misses

$$= M_{spat} \times \left(\prod_{1 \leq i \leq l} \mathcal{N}_i \right)$$

$$\times \left(\sum_{i=1}^{\#AIS^*_l(G)} \sum_{j=1}^{\#ARS_l(R)} \sum_{k=1}^{\#AIS^*_l(G)_i} \frac{\|ARS_l(R)_j \cap AIS^*_l(G)_{i,k}\|}{\mathcal{E}_R} \right)$$

The notation $\#X$ stands for the number of elements in set X, either the number of actual interference sets in the translation group, or the number of intervals in an interference or reuse set (the parameter N). Also, $ARS_l(R)_j$ refers to the j'th interval in the reuse set of R; similarly, $AIS^*_l(G)_{i,k}$ is the k'th interval in the j'th part of translation group G's interference set.

Due to the model's representation of reuse and interference sets, the size of all intervals in a particular set is the same. Since interference between between the two sets is being considered, which happens when two cache *lines* collide, not two cache elements, the size of each interval is defined using the B parameter of the set from (22),

$$\|ARS_l(R)_j\| \equiv B(ARS_l(R)) \times \mathcal{L},$$
$$\|AIS^*_l(G)_{i,k}\| \equiv B(AIS^*_l(G)_i) \times \mathcal{L},$$

where $B(X)$ refers to the B parameter of X, an ARS or AIS. Recall that parameter B represents the average number of cache lines in each interval (see Section 5.1.1).

5.2.4 External Cross-Interference

When evaluating interference from a translation group which the reuse reference doesn't belong to, there is no constant distance δ between the reuse and interference sets. As a consequence, the size of the intersection between the sets is not constant and the number of misses sustained is more difficult to evaluate. Since an accurate method of calculating external cross-interferences would be quite complex, an approximate method is used that gives good results in the majority of cases.

The approximation made is to assume that the distance between two intervals in the reuse and interference sets is a random variable, ranging from 0 to $\mathcal{C} - 1$. This gives the number of misses on an iteration of loop l for any two intervals, X and Y, as,

$$\overline{\|X \cap Y\|} = \frac{1}{\mathcal{C}} \left(\sum_{\delta_{X,Y}=0}^{\mathcal{C}-1} \|X \cap Y\| \right).$$

Due to the definition of the intersection operator in (31), the above expression can be shown to be equivalent to the following, which is much faster to evaluate,

$$\overline{\|X \cap Y\|} = \frac{\|X\|\|Y\|}{\mathcal{C}}.$$

with $||X||$ and $||Y||$ defined using $B(X)$ and $B(Y)$ in the same way as for internal cross-interference.

The approximation made above, of δ varying between zero and the size of the cache is valid when the arrays being accessed by the reference and the translation group are large in relation to the size of the cache. However, if they are significantly smaller than the size of the cache it is possible for all arrays to map to distinct cache locations. If this happens then there is no chance of cross-interference occurring; in this case the assumption made about the distribution of δ is not valid.

To overcome this problem an *overlap ratio*, the function $f_o(R, G)$, is introduced. This is defined by the size of the overlap in the cache between the array accessed by reference R and those by translation group G,

$$f_o(R, G) = \frac{1}{||R||} \times \min\left(||R||, \sum_{\substack{i=1 \\ G_i \neq R}}^{\#G} ||R \cap G_i|| \right). \tag{33}$$

Note that although the same intersection operation is used as in (31), in this case the operands are arrays, not intervals in the cache. The same definition of δ is used, but the sizes of the arrays, $||R||$ and $||G_i||$, are defined slightly differently, to give the size of the array in the cache, rounded up to the next cache line,

$$||R|| = \min\left(\mathcal{C}, \mathcal{L}\left\lceil \frac{(\#R)\mathcal{E}_R}{\mathcal{L}} \right\rceil\right).$$

Where $\#R$ is the size of the array referenced by R, in elements. The size of G_i is defined similarly.

Given the value of $f_o(R, G)$, the overall number of misses due to external cross-interference between the reference R, and the translation group G (with $R \notin G$), can be evaluated.

Similarly to when finding internal cross-interferences, the average value of the intersection, in lines, is summed over all combinations of intervals to give the total number of misses for each interference set in the translation group. In fact, since the size of the intervals in a single reuse or interference set is constant, the two inner sums can be replaced by multiplications,

$$\begin{aligned}
&\text{external interference misses} \\
&\quad = M_{\mathrm{spat}} \times f_o(R, G) \times \left(\prod_{1 \le i \le l} \mathcal{N}_i \right) \\
&\qquad \times \sum_{i=1}^{\#\mathrm{AIS}_l^*(G)} \left(\#\mathrm{ARS}_l(R) \times \#\mathrm{AIS}_l^*(G)_i \frac{\overline{||\mathrm{ARS}_l(R)_1 \cap \mathrm{AIS}_l^*(G)_{i,1}||}}{\mathcal{E}_R} \right)
\end{aligned}$$

5.2.5 Spatial Cross-Interference

The spatial reuse of a reference can also be victim to cross-interference if another reference overwrites a cache line before it has been fully reused. The chance of a reference's spatial reuse being obstructed increases with its spatial reuse distance. A probabilistic method is used to model this effect.

Given a reference R, whose spatial reuse occurs on loop l_{spat} (see Section 5.1.2), the probability P_R, that accessing any single cache element will push an element of R out of the cache, is given by,

$$P_R = B(AIS_{l_{spat}}(R)) \times N(AIS_{l_{spat}}(R)) \times \frac{\mathcal{L}}{\mathcal{C}}$$

where B and N refer to two of the parameters of the interference set of R defined on the spatial reuse loop l_{spat}.

The average number of elements that could be reused, but are pushed out of the cache before they can be, is then P_R multiplied by the number of cache lines the interfering reference uses on loop l_{spat}. This gives the total number of misses due to spatial cross- interference as $P_R BN$, where B and N refer to parameters of the interference set from the reference causing the interference on R.

As with external cross-interference this assumes that the arrays are large in comparison to the cache, scaling the number of misses by the array's overlap ratio will correct the result.

Naturally, this has to be repeated for each part of the translation group's interference set, so that the total number of misses due to spatial cross-interference on R, from the translation group G, is given by,

$$\begin{aligned}
&\text{spatial cross-interference misses} \\
&\quad = (1 - P_{spat}) \times f_o(R, G) \times \left(\prod_{1 \le i \le l_{spat}} \mathcal{N}_i \right) \\
&\quad\quad \times \sum_{i=1}^{\#AIS^*_{l_{spat}}(G)} \Big(P_R \times B(AIS^*_{l_{spat}}(G)_i) \times N(AIS^*_{l_{spat}}(G)_i) \Big)
\end{aligned}$$

The expression is multiplied by $1 - P_{spat}$ to remove the spatial interference caused by the reference itself, which has already been included in the total number of misses, (23).

6 Group Dependences

The number of cache misses due to group dependence reuse, as identified in Section 4.3, is also evaluated. Group dependence reuse occurs when a reference R, accesses data that a different reference $\vec{R}$, has recently used. As an example of this phenomenon consider Figure 11. Here the reference $Y(j_2)$ has two possible reuse dependencies: one to itself, and one to the reference $Y(j_2 + 1)$ which will have accessed the same element in the previous iteration of loop j_2. The group dependence has the smallest reuse distance so this is where reuse occurs for $Y(j_2)$.

When evaluating group reuse the method used is quite different from that used for self dependences, although similar terms are used to describe what is

```
DO j₁ = 0, 𝒩₁ - 1
  DO j₂ = 0, 𝒩₂ - 2
    X(j₂) = Y(j₂) + Y(j₂+1)
  ENDDO
ENDDO
```

Figure 11: Group reuse example

happening. The reuse loop, l_g, is simply defined as the loop on which the group reuse occurs, and the theoretical reuse set of R is defined as the elements referenced by $\vec{R}$ that are subsequently referenced by R. This is somewhat different to in a self dependence where the TRS is defined only for loop l and the loops contained by it; here the set is defined over *all* loops. The group reuse loop l_g is given by,

$$l_g = \max \left\{ \gamma_i(R) \;\middle|\; 1 \le i \le m,\ \beta_i(R) \ne \beta_i(\vec{R}) \right\}$$

where m is the number of dimensions in the array accessed by R and $\vec{R}$. Evaluating the size of the TRS is also rather different, it is calculated in three parts: the loops inside the reuse loop, the reuse loop itself, and the loops containing the reuse loop,

$$\|\mathrm{TRS}(R, \vec{R})\| = \left(\prod_{1 \le i < l_g - 1} \mathcal{N}_i \right) \times \mathrm{size}_{l_g}(R, \vec{R}) \times \left(\prod_{l_g < i \le n} \mathcal{N}_i \right) \tag{34}$$

$$\mathrm{size}_{l_g}(R, \vec{R}) = \begin{cases} \mathcal{N}_{l_g - 1} - 1 & \text{if } \delta_{l_g}(R, \vec{R}) = 0 \text{ and } R \prec \vec{R}; \\ \mathcal{N}_{l_g - 1} \times \left(\mathcal{N}_{l_g} - \delta_{l_g}(R, \vec{R}) \right) & \text{otherwise.} \end{cases}$$

In actual fact, the middle portion of the calculation, the number of iterations achieving reuse, represented by $\mathrm{size}_{l_g}(R, \vec{(}R)$ handles both loops l_g and $l_g - 1$. This is to allow the first case of the expression, where both references refer to the same element, but R occurs *before* $\vec{R}$, and therefore can't reuse the data until the next iteration of loop $l_g - 1$.

The definition of $\delta_{l_g}(R, \vec{R})$ is the same as in (2). Note that δ_{l_g} is a convenient notation meaning the difference between R and $\vec{R}$ in the dimension referring to loop variable j_{l_g}, l_g is a loop, not an array dimension!

6.1 Compulsory Misses

The iterations of loop l_g that don't reuse any of the elements previously accessed by $\vec{R}$ generate a number of compulsory misses. These are evaluated by subtracting the size of the theoretical reuse set from the total number of times the reference is executed, and applying the degree of spatial reuse attained by the reference (see Section 5.1.2). For a reference R,

$$\text{group compulsory misses} = M_{\mathrm{spat}} \times \left(\left(\prod_{1 \le i \le n} \mathcal{N}_i \right) - \|\mathrm{TRS}(R, \vec{R})\| \right).$$

6.2 Internal Interference

The cross-interference disrupting a group dependence is also evaluated. Again, it is split into internal and external forms. For group reuse, internal interference behaves in a markedly different manner from in a self dependence. Since any references causing internal interference are, by definition, in the same translation group, they are accessing the cache in exactly the same pattern, except at a different position in the cache. For each reference the starting position of the pattern is equal to $\mathcal{B} \bmod \mathcal{C}$.

If the starting position of any of the other references in the translation group (apart from R and $\vec{R}$) is between the starting points of R and $\vec{R}$, then self interference occurs. The other reference will overwrite any data elements loaded by $\vec{R}$ before R can access them. It follows that because the references are all "moving" at the same rate, internal interference can only take two forms, none, or total; meaning that if interference does occur it overwrites *all* elements in the reuse set.

For a reference R with a group dependence $\vec{R}$, interference from a reference R' in the same translation group, is detected using the following definition, assuming $\phi(R) = \mathcal{B}(R) \bmod \mathcal{C}$,

$$\mathrm{hasInt}(R, \vec{R}, R') = \begin{cases} \phi(R) < \phi(R') \quad \text{and} \quad \phi(\vec{R}) > \phi(R') & \text{if } \phi(R) \leq \phi(\vec{R}); \\ \phi(R) > \phi(R') \quad \text{and} \quad \phi(\vec{R}) < \phi(R') & \text{if } \phi(R) > \phi(\vec{R}). \end{cases}$$

An extra clause is added to detect when R and R' reference the same part of a single array. If internal interference does occur then the number of misses caused is simply the size of the reuse set multiplied by the spatial interference ratio, that is,

$$\text{group internal misses } = M_{\mathrm{spat}} \times \|\mathrm{TRS}(R, \vec{R})\|,$$

but only when $\mathrm{hasInt}(R, \vec{R}, R')$ is true.

6.3 External Interference

As with self-dependence reuse, external interference of group reuse is evaluated using an approximate model. In fact the model used is very similar to that used with self-dependences (see Section 5.2.4). Group interference can only occur in the period between an element first being loaded and it being reused—related to the reuse distance of the dependence. When modeling the interference occurring in this period only the iterations of the reuse loop corresponding to the reuse distance of the loop are included in the AIS.

This is defined as the following expression, where $\mathrm{AIS}^*_{l_g,\delta_{l_g}(R,\vec{R})}(G)$ represents the actual interference set of the translation group G. But with the iterations of loop l_g restricted to $\delta_{l_g}(R, \vec{R})$. This is achieved by letting $\mathcal{N}_{l_g}$ equal $\delta_{l_g}(R, \vec{R})$ while computing the interference set. The actual number of misses is then,

$$\text{group external misses} = M_{\mathrm{spat}} \times \overline{\|\mathrm{TRS}(R, \vec{R}) \cap \mathrm{AIS}^*_{l_g,\delta_{l_g}(R,\vec{R})}(G)\|} \times f_o(R, G).$$

As with self reuse, the overlap ratio $f_o(R, G)$, from (33), and M_{spat} from (23), are used to scale the result to reflect the size of the overlap, and the spatial reuse achieved.

To complete the evaluation of a group dependent reference, spatial cross interference is also included; it is calculated in exactly the same way as with self dependence reuse (see Section 5.2.5).

7 Example Results

This section presents experimental results obtained using the techniques described in the previous section. The models have been implemented in C, driven by a front-end that accepts descriptions of loop nestings in a simple language. Three code fragments are examined here, each for a range of problem sizes and cache configurations; the fragments were chosen for their contrasting characteristics. Each example manipulates matrices of double precision values, arranged in a single contiguous block of memory.

1. A matrix-multiply. It consists of three nested loops, containing four array references in total. Each reference allows temporal reuse to occur within one of the loops. The Fortran code is shown in Figure 12(a).

2. A two dimensional Jacobi loop, from [2], originally part of an application that computes permeability in porous media using a finite difference method. This kernel exhibits large amounts of group dependence reuse. The matrices `IVX` and `IVY` contain 32-bit integers. See Figure 12(c).

3. A "Stencil" operation, from [10]. This kernel shows group dependence reuse, but sometimes doesn't access memory sequentially. See Figure 12(b).

For each code fragment the predicted miss ratios are plotted against the matrix size in Figures 13, 14, and 15. The "differences from simulation" shown are calculated by subtracting the miss ratio obtained by standard simulation techniques[8]. The cache parameters used—8, 16, and 32 kilobyte capacities, each with 32 and 64 bytes per line—were chosen to be illustrative rather than to represent any particular system, although several processors have primary data caches matching these parameters, e.g. SUN UltraSPARC. Table 1 shows the average errors for each experiment, as percentages of the simulated values. Also shown, in Table 2, are the range of times taken to evaluate each problem on a 147MHz SUN ULTRA-1 workstation, for a single cache configuration.

8 Discussion

When examining the results several general effects are evident. Firstly, the underlying trend of the miss-ratio decreases as the cache and line size are increased;

[8] A locally written cache simulator was used that accepts loop descriptions in the same form that the analytical model uses. It was validated by comparing its results with Hill's Dinero III trace-driven simulator [7].

```
DO I = 0, N-1
  DO J = 0, N-1
    Z(J, I) = 0.0
    DO K = 0, N-1
      Z(J, I) = Z(J, I)
        + X(K, I) * Y(J, K)
    ENDDO
  ENDDO
ENDDO
```

(a) Matrix multiply

```
DO I = 0, N-1
  DO J = 0, N-1
    A(J, I) = A(J, I+1)
      + B(J, I) + B(J+1, I)
      + C(I, J) + C(I+1, J)
  ENDDO
ENDDO
```

(b) Stencil

```
DO J = 1, N-1
  DO I = 1, N-1
    VXN(I,J) = (c0 * VXO(I,J) + dty2 * (VXO(I-1,J) + VXO(I+1,J))
      + dtx2 * (VXO(I,J+1) + VXO(I,J-1))
      - dtx * (PO(I,J) - PO(I,J-1)) - c1) * IVX(I,J)
    VYN(I,J) = (c0 * VYO(I,J) + dty2 * (VYO(I-1,J) + VYO(I+1,J))
      + dtx2 * (VYO(I,J+1) + VYO(I,J-1))
      - dty * (PO(I-1,J) - PO(I,J)) - c2) * IVY(I,J)
  ENDDO
ENDDO
```

(c) 2D Jacobi

Figure 12: Example kernels

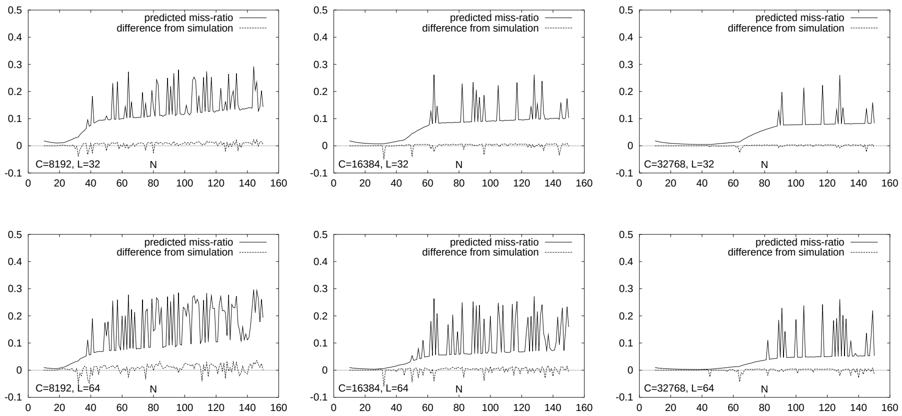

Figure 13: Matrix multiply results

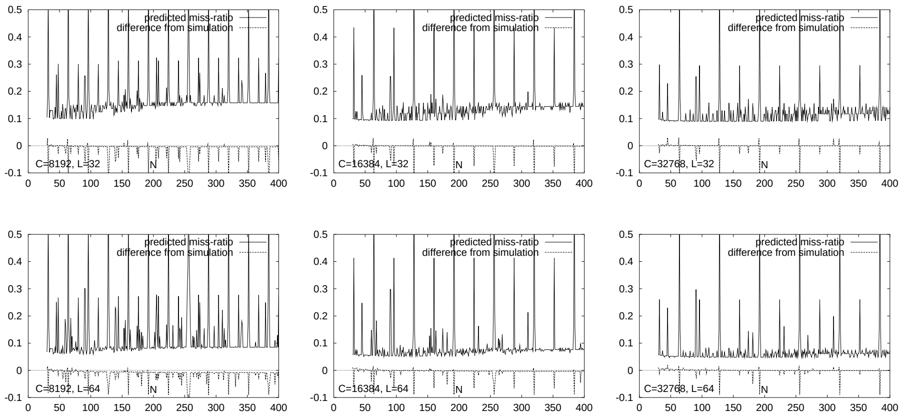

Figure 14: 2D Jacobi results

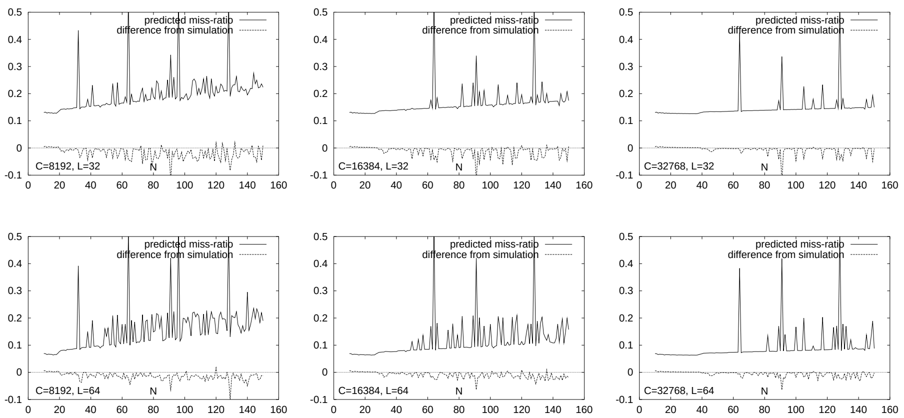

Figure 15: Stencil results

Experiment	$\mathcal{L}$	% Error		
		$\mathcal{C} = 8192$	$\mathcal{C} = 16384$	$\mathcal{C} = 32768$
Matrix multiply	32	8.39	6.75	5.18
	64	14.2	12.3	10.3
2D Jacobi	32	4.11	1.91	1.39
	64	8.64	4.37	2.73
Stencil	32	8.35	6.30	3.76
	64	8.48	7.64	5.81

Table 1: Average errors

	Analytical Model			Simulation		
Experiment	Min.	Max.	Mean	Min.	Max.	Mean
Matrix mult.	0.00115	0.00135	0.00156	0.00780	25.1	7.14
2D Jacobi	0.00149	0.00262	0.00201	0.0206	4.20	1.54
Stencil	0.00069	0.000879	0.000821	0.00122	0.207	0.0768

Table 2: Calculation times for $\mathcal{C} = 16384$, $\mathcal{L} = 32$ experiments (seconds.)

this is expected since increasing the capacity will help temporal locality, while increasing the line size benefits spatial locality. Perhaps slightly more interesting is that interference also decreases as the capacity is increased, but *increases* with the line size. This follows from the total number of cache lines, $\mathcal{C}/\mathcal{L}$, being related to the level of interference. The results also show that interference *is* significant, with over a factor of 5 difference in miss-ratio witnessed between similar experiments, but with the width and height of the matrices increased by one element.

Looking at the matrix multiply results in Figure 13 shows that there are generally three stages in each plot. Initially, for small problem sizes, all three matrices fit in the cache and the miss-ratio is very low. After a certain point the matrices are large enough that they can't all fit in the cache and the miss-ratio starts to increase, followed by a period in which the underlying miss-ratio increases quite slowly (largely as a result of increased cross interference), with points at which spatial interference on reference $\mathsf{Y}(\mathsf{J},\mathsf{K})$ massively increases the miss-ratio.

The results shown for the Jacobi example, in Figure 14, are very different to those of the matrix multiply. Ignoring the largest peaks for the moment, two almost constant underlying trends can be seen, one for small problem sizes, one for larger values of N. As the problem size increases, so does the interference, and the miss-ratio gradually moves from the lower trend to the upper, until the miss-ratio is almost constant again. This effect is largely due to increasing internal cross interference on the group dependence reuse. The large peaks in the results are caused by "ping-pong" interference, that occurs when two reference with similar access patterns always access the same cache line, preventing all *reuse* by these references.[9]

The stencil results show a combination of the effects already mentioned; substantial self interference can disrupt the spatial reuse of references $\mathsf{C}(\mathsf{I},\mathsf{J})$ and $\mathsf{C}(\mathsf{I}+1,\mathsf{J})$, since they use the innermost loop to traverse the *columns* of the matrix. The gradual increase in underlying miss-ratio is similar to the Jacobi kernel, and is largely from interference on the group dependence reuse.

The average errors, given in Table 1, support the validity of the models. For each experiment the average error when compared with simulation is less than fifteen per cent, and in all but three of the eighteen experiments the error is less than ten per cent. It is also encouraging that the models detect the worst

[9] Models for this type of interference have not been described in the paper, although rudimentary models are included in the implementation. As can be seen in the Jacobi data these models are not always entirely accurate.

cases very well, which is important for optimization purposes.

One of the motivations for this work was to minimize the time taken when evaluating a program fragment. As expected the analytical model is much quicker to compute than a simulation, typically several orders of magnitude, even with the smallest problem sizes. As the total number of array accesses grows the gulf widens: simulation time increasing proportionally to the number of accesses, the time needed to evaluate the analytical model staying mostly constant.

9 Conclusions

A hierarchical method of classifying cache interference has been presented, for both self and group dependent reuse of data. Analytical techniques of modeling each category of interference have been developed for array references in loop nestings and direct-mapped cache architectures. It has been shown that these techniques will give accurate results, comparable with those found by simulation, and that they can be implemented such that predictions can be made at a much faster rate than with simulation. More importantly, the prediction rate has been shown to be proportional to the number of array references in the program, rather than to the actual number of memory accesses as in a simulation.

It is envisaged that the benefits of the models—accuracy and speed of prediction—will allow their use in a wide range of situations, including those that are impractical with more traditional techniques. An important example of such a use will be run-time optimization of programs, using analytical models of the cache behavior of algorithms to drive the optimization process.

Areas that will be addressed in future work will include such optimization strategies, as well as extensions to the model itself. The extensions under consideration include modeling set-associative caches, and increasing the accuracy when modeling certain types of problems. It is also intended to use the techniques as part of a general purpose performance modeling system.

Acknowledgements. This work is funded in part by DARPA contract N66001-97-C-8530, awarded under the Performance Technology Initiative administered by NOSC.

References

[1] Agarwal, A., Horowitz, M., and Hennessy, J. An analytical cache model. *ACM Trans. Comput. Syst. 7*, 2 (May 1989), 184–215.

[2] Bodin, F., and Seznec, A. Skewed associativity improves program performance and enhances predictability. *IEEE Trans. Comput. 46*, 5 (May 1997), 530–544.

[3] Coleman, S., and McKinley, K. S. Tile size selection using cache organisation and data layout. In *Proceedings of the SIGPLAN '95 Con-*

ference on Programming Language Design and Implementation (June 1995), vol. 30, pp. 279–289.

[4] Fahringer, T. Automatic cache performance prediction in a parallelizing compiler. In *Proceedings of the AICA'93 — International Section* (Sept. 1993).

[5] Fricker, C., Temam, O., and Jalby, W. Influence of cross-interferences on blocked loops: A case study with matrix-vector multiply. *ACM Trans. Program. Lang. Syst. 17*, 4 (July 1995), 561–575.

[6] Gee, J. D., Hill, M. D., Pnevmatikatos, D. N., and Smith, A. J. Cache performance of the SPEC92 benchmark suite. *IEEE Micro 13*, 4 (Aug. 1993), 17–27.

[7] Hill, M. D. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California, Berkeley, 1987.

[8] Lam, M. S., Rothberg, E. E., and Wolf, M. E. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, California, 1991), pp. 63–74.

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

[10] McKinley, K. S., and Temam, O. A quantitative analysis of loop nest locality. In *Proceedings of the 7th Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, Oct. 1996), vol. 7.

[11] Sugumar, R. A., and Abrahams, S. G. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of ACM SIGMETRICS* (1993), pp. 24–35.

[12] Temam, O., Fricker, C., and Jalby, W. Impact of cache interference on usual numerical dense loop nests. *Proceedings of the IEEE 81*, 8 (Aug. 1993), 1103–1115.

[13] Temam, O., Fricker, C., and Jalby, W. Cache interference phenomena. In *Proceedings of ACM SIGMETRICS* (1994), pp. 261–271.

[14] Uhlig, R. A., and Mudge, T. N. Trace-driven memory simulation: A survey. *ACM Comput. Surv. 29*, 2 (June 1997), 129–170.

[15] Wolf, M. E., and Lam, M. S. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation* (June 1991), vol. 26, pp. 30–44.

A Notation and Symbols

Notation	*Meaning*
$x \bmod y$	remainder, i.e. $x - y\lfloor x/y \rfloor$
X_0	base address of array X
TRS_l	theoretical reuse set on loop l
ARS_l	actual reuse set on loop l
TIS_l	theoretical interference set on loop l
AIS_l	actual interference set on loop l
AIS_l^*	AIS of a translation group
$\|X\|$	cumulative size of set X in the cache
$X \cap Y$	intersection between X and Y
$\#X$	number of elements in set X
$\{x \mid P(x)\}$	set of all x such that $P(x)$
x^+	positive part of x, i.e. $\max(x, 0)$
$\overline{x}$	mean value of x
$R \prec R'$	R occurs before R' in the loop nest
$\vec{R}$	reuse dependence of R

Symbol	*Usage*
$\mathcal{C}$	size of the cache
$\mathcal{L}$	size of each cache line
$\mathcal{E}$	size of each array element
n	depth of the loop nesting
m	number of dimensions in an array
$\mathcal{N}_i$	number of iterations of loop i
j_i	loop variable of loop i
α_k	multiplier of dimension k of a reference
β_k	additive part in dimension k of a reference
γ_k	loop referred to by dimension k of a reference
j_{γ_k}	variable referred to by dimension k
$\mathcal{B}$	constant term of a linear form
$\mathcal{A}_i$	coefficient of loop variable j_i
R, R'	References
G	A translation group
l	loop on which reuse occurs
S	average size of intervals in the cache
B	number of cache lines in an interval
σ	average distance between intervals
σ_k	value of σ at level k of the subdivision
N	number of intervals
ϕ	position in the cache of the first interval
s	final subdivision level
$\tilde{\sigma}_k$	absolute value of σ_k in relation to σ_{k-1}
r	number of left over intervals

Symbol	*Usage*
n_s	number of intervals in each area
M_{spat}	spatial interference factor
P_{spat}	probability of spatial self-interference
C_{spat}	spatial compulsory miss-ratio
P_R	probability of spatial interference on R