CORE

Metadata, citation and similar papers at core.ac.uk

Provided by Warwick Research Archives Portal Repository

THE UNIVERSITY OF WARWICK

Original citation:
Harper, J. S., Kerbyson, D. J. and Nudd, G. R. (1998) Analytical modeling of set-associative cache behaviour. University of Warwick. Department of Computer Science. (Department of Computer Science Research Report). (Unpublished) CS-RR-349

Permanent WRAP url:
http://wrap.warwick.ac.uk/61062

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

A note on versions:
The version presented in WRAP is the published version or, version of record, and may be cited as it appears here.For more information, please contact the WRAP Team at: publications@warwick.ac.uk

warwick**publications**wrap

highlight your research

http://wrap.warwick.ac.uk/

Analytical Modeling of Set-Associative Cache Behavior

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

September 16, 1998

Abstract

Cache behavior is complex and inherently unstable, yet is a critical factor affecting program performance. A method of evaluating cache performance is required, both to give quantitative predictions of miss-ratio, and information to guide optimization of cache use.

Traditional cache simulation gives accurate predictions of miss-ratio, but little to direct optimization. Also, the simulation time is usually far greater than the program execution time. Several analytical models have been developed, but concentrate mainly on direct-mapped caches, often for specific types of algorithm, or to give qualitative predictions.

In this work novel analytical models of cache phenomena are presented, applicable to numerical codes consisting mostly of array operations in looping constructs. Set-associative caches are considered, through an extensive hierarchy of cache reuse and interference effects, including numerous forms of temporal and spatial locality. Models of each effect are given, which, when combined, predict the overall miss-ratio. An advantage is that the models also indicate sources of cache interference.

The accuracy of the models is validated through example program fragments. The predicted miss-ratios are compared with simulations, and shown typically to be within fifteen percent. The evaluation time of the models is shown to be independent of the problem size, generally several orders of magnitude faster than simulation.

1 Introduction

Cache performance is one of the most critical factors affecting the performance of software, and with memory latency continuing to increase in respect to processor clock speeds utilizing the cache to its full potential is more and more essential. Yet cache behavior is extremely difficult to analyze, reflecting its unstable nature in which small program modifications can lead to disproportionate changes in cache miss ratio [2, 12]. A method of evaluating cache performance is required, both to give quantitative predictions of miss ratio, and information to guide optimization of cache use.

[*] `john@dcs.warwick.ac.uk`

Traditionally cache performance evaluation has mostly used simulation, emulating the cache effect of every memory access through software. Although the results will be accurate, the time needed to obtain them is prohibitive, typically many times greater than the total execution time of the program being simulated [13]. Another possibility is to measure the number of cache misses incurred, using the performance monitoring features of modern microprocessors. This can also give accurate results, and in reasonable time, but introduces the restriction that only cache architectures for which actual hardware is available can be evaluated.

To try and overcome these problems several analytical models of cache behavior have been developed. One such technique is to extract parameters from an address trace and combine them with parameters defining the cache to derive a model of cache behavior [1]. This method is able to accurately predict the general trends in behavior, but lacks the fine detail that is needed to model the instability noted above. Analytical models combined with heuristics have also been used to guide optimizing compilers in their choice of source code transformations [14, 4, 10]. The models developed however are usually unsuitable for more general performance evaluation, since they often aim for qualitative, rather than quantitative, predictions. Another area in which analytical models have been employed has been in studying the cache performance of particular types of algorithm, especially in the analysis of blocked algorithms [9, 3, 5].

Attempts have been made at creating general purpose models that are both accurate and expressive, with some success [12, 6, 7], but in all cases limited to describing direct-mapped caches. In this work we present novel analytical techniques for predicting the cache performance of a large class of loop nestings, for the general case of set-associative caches (i.e. with direct-mapped as the case with associativity one). All forms of cache reuse and interference are considered leading to accurate, yet rapidly evaluated, models. These benefits and others are demonstrated through the examination of several example code fragments. The work has a wide range of possible applications, from aiding software development to on the fly performance prediction and management. We also plan to integrate the model with an existing system for analyzing the performance of parallel systems [11].

The paper is organized as follows: the next section outlines the problem being addressed, and the classification of the cache phenomena being modeled. Section 3 describes in detail how the effect of array references on the cache is represented, and how this representation can be efficiently computed. In Sections 4, 5, and 6, the different types of cache reuse are considered, in terms of the representation developed in Section 3. Finally, Section 7 presents experimental data showing how the models compare with simulation, followed by a discussion of these results and our conclusions in Sections 8 and 9.

```
DO $j_1$ = 0, $\mathcal{N}_1$ - 1
  DO $j_2$ = 0, $\mathcal{N}_2$ - 1
    DO $j_3$ = 0, $\mathcal{N}_3$ - 1
      ...
    ENDDO
  ENDDO
ENDDO
```

Figure 1: General form of considered loop constructs

2 Overview of methodology

2.1 Concepts

The models presented in this work consider the cache behavior of array references accessed by regular looping constructs. The general form of a loop nesting is shown in Figure 1; the loops are numbered from 1 to n, outer-to-innermost respectively, and are assumed to be normalized such that they count upward from zero in steps of one. The number of iterations performed by a loop at a level k is labeled $\mathcal{N}_k$, and the variable used to index arrays by this loop is labeled j_k.

Array references considered are of the form:

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

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

Such array references can be rearranged into the form of a linear expression, giving the address of the element accessed for a particular combination of values of $j_1 \ldots j_n$, the general form being

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

where $\mathcal{B}$ and $\mathcal{A}_1 \ldots \mathcal{A}_n$ are constants. The base address of the array and the β_k values combine to form $\mathcal{B}$; the $\mathcal{A}$ values are derived from the loop multipliers α_k and the dimensions of the array. Without loss of generality we assume that array indices are in the Fortran style, and that all values are in terms of array elements.

The concept of an iteration space is also important. The loop bounds $\mathcal{N}_1 \ldots \mathcal{N}_n$ represent the full n-dimensional iteration space of the array reference being examined. By limiting the range of certain loops the iteration space can also be restricted. For example by only allowing j_1 to have the value 0, only a single iteration of the outermost loop is specified. When modeling cache behavior this restriction of iterations is a natural way to consider some problems. However, in this work we will only need to restrict the upper bound of loops, for a loop k this can be handled by "binding" a temporary value to $\mathcal{N}_k$.

Given two array references R_1 and R_2, if their linear forms are identical except $\mathcal{B}_1 \neq \mathcal{B}_2$, then they are said to be *in translation*. This means that the access patterns of both references are identical, but offset by $|\mathcal{B}_1 - \mathcal{B}_2|$ elements in the address space. References in translation with one another are said to be in the same *translation group*.

2.2 Evaluation strategy

The function of the cache is to provide data reuse, that is, to enable memory regions that have recently been accessed to be subsequently accessed with a much smaller latency. Two basic forms of reuse exist: *self dependence reuse* in which an array reference repeatedly accesses the same elements of an array, and *group dependence reuse* in which elements are repeatedly accessed that were most recently used by a *different* array reference.

When considering an array reference R, its *reuse dependence* $\vec{\mathsf{R}}$ is defined as the reference from which its reuse arises. When $\mathsf{R} = \vec{\mathsf{R}}$ it is a self dependence, conversely, when $\mathsf{R} \neq \vec{\mathsf{R}}$ it is a group dependence. Since it is possible for more than two references to access the same data elements, when identifying dependences $\vec{\mathsf{R}}$ is defined as the reference with the smallest reuse distance from R, related to the number of loop iterations occurring between $\vec{\mathsf{R}}$ accessing an element, and R reusing it.

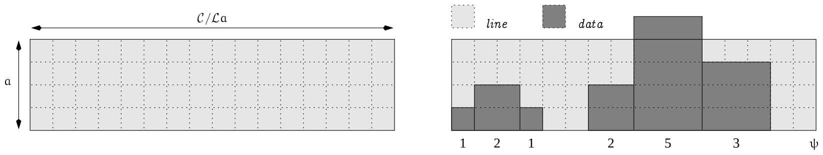
Unlike the most well known system for classifying cache misses, the "three C's" model (compulsory, capacity and conflict misses [8]), the method presented by this paper uses a two part model. Compulsory misses are defined as before, but capacity and conflict misses are considered a single category—*interference misses*—since the underlying cause is the same, reusable data being ejected from the cache.

To predict the number of cache misses suffered by an array reference interference is divided into several types, dependent on their source. *Self interference* occurs when the reference obstructs its own reuse, *internal cross interference* occurs due to references in the same translation group, and *external cross interference* is caused by references in different translation groups. Sections 5 and 6 describe how these effects are modeled for self and group dependences respectively.

A distinction is also made between the temporal and spatial components of a reference's miss ratio. The spatial miss ratio is defined as the average number of cache misses needed to load a single cache line of data; all three types of interference contribute to this ratio, and are modeled in Section 4. The spatial miss ratio is applied to the predicted number of temporal misses to give the total number of cache misses for a reference. Repeating this procedure for all references in the loop nesting gives the prediction for the entire code fragment.

3 Modeling cache footprints

A common requirement when modeling interference is to identify the effect on the cache of accessing all references in a single translation group, for a specified



(a) Cache representation

(b) Example of ψ

Figure 2: Examples of cache layout and set overlap ψ.

iteration space. Once the effect on the cache is known, it can be used to predict the effect on the reuse of the references being examined, and from this the resulting number of cache misses can be predicted.

A cache of size $\mathcal{C}$, with $\mathcal{L}$ elements in a line, and associativity a can be considered as a rectangular grid, $\mathcal{C}/\mathcal{L}a$ cache lines wide and a lines deep, as shown in Figure 2(a). Each vertical column represents a single "set" of the cache, each containing a lines. A mapping function determines which set a memory element is stored in; the usual mapping function, which this paper examines, is simply $x \bmod (\mathcal{C}/\mathcal{L}a)$, where x is the address of the element in question. The line in the set that is actually used depends on the replacement strategy employed. In this paper only the "least-recently-used" strategy is considered, replacing the line in the set that was last accessed the earliest.

Given this view of the cache, the effect of a translation group on the cache can also be visualized. For each of the $\mathcal{C}/\mathcal{L}a$ sets a certain number of the lines contain data accessed by the translation group. This number can be thought of as the "overlap" of each set, and is labeled ψ. Figure 2(b) shows an example for a small 4-way set-associative cache ($\mathcal{C} = 64\mathcal{L}$, $a = 4$). Data elements loaded into the cache are darkly shaded, with the value of ψ for each set shown underneath.

To identify interference, cache footprints such as this are compared with the footprints of the data being reused, either set-by-set or statistically as a whole. The method of detecting interference is simple: it occurs wherever the combined overlap of the footprints is greater than the level of associativity.

The model represents each cache footprint as a sequence of regions, each region having a constant value of ψ, the overlap. As well as ψ, two other parameters define each region, its first element of the region (i.e. a value between zero and $(\mathcal{C}/a) - 1$), and the number of elements in the region. Considering the example footprint in Figure 2(b), it is clear that it is defined by the following sequence of regions $(start, size, \psi)$,

$$(0, \mathcal{L}, 1), (\mathcal{L}, 2\mathcal{L}, 2), (3\mathcal{L}, \mathcal{L}, 1), (6\mathcal{L}, 2\mathcal{L}, 2), (8\mathcal{L}, 3\mathcal{L}, 5), (11\mathcal{L}, 3\mathcal{L}, 3). \tag{1}$$

In the rest of this section of the paper, we show how footprints of this form can be calculated efficiently for individual translation groups.

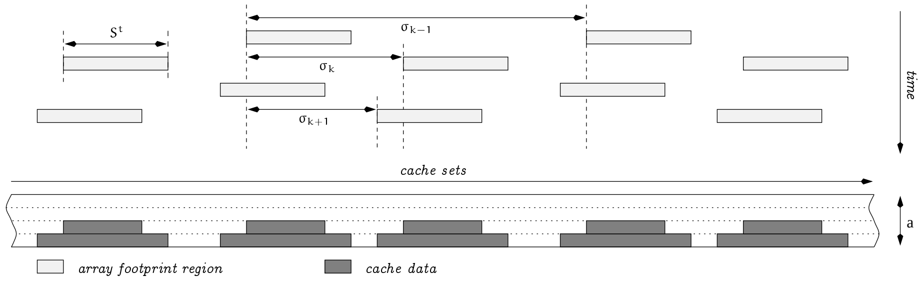


Figure 3: Example of mapping array footprint regions into the cache.

3.1 Finding the cache footprint of a single reference

An accurate method of mapping a regular data footprint into a direct-mapped cache has previously been presented in detail [7, 12]. As such we only consider the problem briefly, extending the method given in [7] (which is descended from [12]) to set-associative caches.

Given an array reference we wish to find the cache footprint of the data it accesses for a particular iteration space, the form of which is defined by the values $\mathcal{N}_1 \ldots \mathcal{N}_n$. The structure of these array elements is defined by the reference itself, the array dimensions, and the iteration space. For the majority of array references encountered, the array footprint can be expressed using four parameters: the address of the first element ϕ^t, the number of elements in each contiguous region S^t, the distance between the start of two such regions σ^t, and finally the number of regions N^t.

After identifying these four parameters, the array footprint they describe is mapped into the cache to give the cache footprint of the reference in question. The cache footprint is defined by parameters similar to those describing the array footprint: the interval between regions σ, the number of regions N, and the position of the first[1] region ϕ. Two parameters define the structure of the data elements in each region, the level of overlap ψ, as defined in Section 3, and S, the number of elements in the region divided by the overlap. Considering Figure 2, ψ and S can be thought of as the average "height" and "width" of each region in the footprint.

To find the parameters defining the cache footprint we use a recursive method of dividing the cache into regular *areas*. 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 3 for part of a cache. A recurrence relation defines the sequence of σ_k values, representing how the array footprint regions map into the cache,

$$\sigma_0 = \mathcal{C}/a, \qquad \sigma_1 = \sigma^t, \qquad \sigma_k = \sigma_{k-1} - \sigma_{k-2} \bmod \sigma_{k-1}, \qquad \text{for } k \geq 0. \quad (2)$$

The sequence is truncated at a level s, where either all N^t regions map into the cache without overlapping, or overlapping occurs between regions. To detect

[1] the "first" region is not the one nearest cache set zero, but she first region in the sequence of N, this sequence may cross the cache boundary.

overlap from either end of an area of size σ_{k-1} a value $\tilde{\sigma}_k$ is introduced, the smallest distance between two regions in the area. If $\tilde{\sigma}_k < S^t$ overlapping occurs on level k, where,

$$\tilde{\sigma}_k = \min\left(\sigma_k, \sigma_{k-1} - \sigma_k\right). \tag{3}$$

At level s, the cache has been divided into σ_0/σ_{s-1} areas of size σ_{s-1}; in each there are a certain number of footprint regions of size S^t, each a distance $\tilde{\sigma}_s$ from the previous. There are r areas that contain $n_s + 1$ regions, and $(\sigma_0/\sigma_{s-1}) - r$ areas containing n_s,

$$n_s = \left\lfloor \frac{N^t}{\sigma_0/\sigma_{s-1}} \right\rfloor, \qquad \text{and} \qquad r = \left\lfloor N^t - n_s\left(\sigma_0/\sigma_{s-1}\right)\right\rfloor. \tag{4}$$

In the simplest case, when $s = 1$, i.e. the array footprint didn't wrap around the end of the cache (no overlapping), $\sigma = \sigma_s$ and $N = N^t$. In the general case when $s > 1$, the distance between each area $\sigma = \sigma_{s-1}$, and the total number of areas $N = \lfloor \sigma_0/\sigma_{s-1} \rfloor$. The position of the first region can also be found, $\phi = \phi^t \bmod (\mathcal{C}/a)$.

The overlap of a single area is found by dividing the total number of elements in it by the distance from the start of the first region to the end of the last. The average level of overlap ψ is found by combining the overlap for both types of area,

$$\psi = \frac{(n_s S^t/f_l(n_s))(\sigma_0/\sigma_{s-1} - r) + ((n_s+1)S^t/f_l(n_s+1))r}{N}, \tag{5}$$

where $f_l(x) = S^t + (x-1)^+ \tilde{\sigma}_s$, the distance from the start of the first region to the end of the last.[2]

To find S for a single area, the size of the "gaps" between regions is subtracted from the distance from the start of the first region to the end of the last region. As when finding ψ the values for both types of area are combined,

$$S = \frac{f_i(n_s)(\sigma_0/\sigma_{s-1} - r) + f_i(n_s+1)r}{N}$$
$$f_i(x) = f_l(x) - (x-1)^+(\tilde{\sigma}_s - S^t)^+$$

The function $f_i(x)$ gives the value of S for an area containing x regions, each $\tilde{\sigma}_s$ from the previous.

3.2 Combining individual cache footprints

Using the techniques presented in the previous section, the cache footprint of a reference for a defined iteration space can be identified. This gives the information necessary to predict how that reference interacts with the footprints of other references, thus allowing interference to be detected.

Generally, however, there are more than two references in a loop nesting, and therefore interference on a reference can originate from more than one source.

[2] Note that $x^+ = \max(x, 0)$.

As well as modeling the interference from each reference in the loop nesting, it is also important that interference on a single element only be counted once. Simply comparing the cache footprint of every reference in the loop with the footprint of the reference being examined will not meet this requirement.

As noted in Section 2.1, it is possibly to classify the array references in a loop nesting into translation groups, all members of a group have exactly the same access pattern, the only difference being that the patterns are offset from one another in the address space. This allows the references in a translation group to be combined into a single cache footprint—it is this meta-footprint that is used to identify interference.

The problem can be stated as follows: given q references in translation: $R_1 \dots R_q$, it is necessary to find the cumulative cache footprint of these references, assuming that the array footprint of the references is defined by the parameters S^t, N^t and σ^t, and the values $\phi^t_1 \dots \phi^t_q$. The combined cache footprint is defined as a sequence of regions defined by triples, (ϕ, S, ψ^*), the position of the region, the size in elements, and the level of overlap, as shown in (1).

3.2.1 Finding the one-dimensional footprint

Examining the calculations in Section 3.1 shows that the only parameter of the cache footprint depending on ϕ^t is ϕ, the position of the first region, defined as $\phi = \phi^t \bmod (\mathcal{C}/a)$. It follows therefore that all references $R_1 \dots R_q$ share the same cache footprint, but with individual values of ϕ: $(\phi^t_1 \bmod (\mathcal{C}/a)) \dots (\phi^t_q \bmod (\mathcal{C}/a))$.

This property is easy to visualize, form a cylinder from the rectangular representation of the cache in Figure 2(a), such that the first and last sets are adjacent to one another. The surface area of the cylinder represents the cache. If we project a cache footprint onto the cylinder, such that it starts at the first element of the cache (i.e. $\phi = 0$), by rotating the footprint ϕ^t positions[3] around the circumference of the cylinder we have the actual cache footprint. This simplifies the problem of finding the combined cache footprint, instead of computing q footprints and merging them, it is only necessary to compute one footprint, then consider rotated copies.

Generating the position of every footprint region. From the definition of ϕ given above, the start and end points of each region in the cache footprint of each reference can be enumerated. The region starting positions for reference R_i are defined by the series,

$$\begin{aligned}&\left[(\phi^t_i + k\sigma) \bmod (\mathcal{C}/a) \mid k = 0 \dots N-1\right] \qquad (6)\\ &\quad \equiv \phi^t_i \bmod (\mathcal{C}/a),\ (\phi^t_i + \sigma) \bmod (\mathcal{C}/a),\ \dots,\ (\phi^t_i + (N-1)\sigma) \bmod (\mathcal{C}/a).\end{aligned}$$

and the position of the end of each region by,

$$\left[(\phi^t_i + S + k\sigma) \bmod (\mathcal{C}/a) \mid k = 0 \dots N-1\right].$$

[3] Or rather $\phi^t \bmod (\mathcal{C}/a)$ since the circumference of the cylinder is $\mathcal{C}/a$.

One possible method of merging all q footprints would be to enumerate the start and end positions of each reference, and then sort them into smallest-first order. Fortunately, there is a much more efficient method. Each rotated footprint can only cross the boundary between cache position $\mathcal{C}/a$ and position zero once. This allows the start and end positions of each region to be generated in numerical order, by generating the points after the cache boundary, followed by the points before the cache boundary.

First the starting points of each region in the footprint of reference i are considered. The first region (when counting from zero) to start after position $\mathcal{C}/a$, is,

$$\mathit{start\_after}_i = \left\lceil \frac{\mathcal{C}/a - (\phi_i^t \bmod (\mathcal{C}/a))}{\sigma} \right\rceil .$$

The list of starting positions in (6) can now be split in two and recombined, so that the list of positions in ascending order is,

$$\begin{aligned}&\big[(\phi_i^t + k\sigma) \bmod (\mathcal{C}/a) \mid k = \mathit{start\_after}_i \ldots (N-1)\big]\\ &\quad + \!\!+ \ \big[(\phi_i^t + k\sigma) \bmod (\mathcal{C}/a) \mid k = 0 \ldots (\mathit{start\_after}_i - 1)\big],\end{aligned}$$

assuming that the $+\!\!+$ operator concatenates two lists.

A similar method can be used to generate the end points of each region in the footprint of reference R_i. The first end point after cache position $\mathcal{C}/a$ is,

$$\mathit{end\_after}_i = \left\lceil \frac{\mathcal{C}/a - ((S + \phi_i^t) \bmod (\mathcal{C}/a))}{\sigma} \right\rceil ,$$

and the list of end points in numerical order is,

$$\begin{aligned}&\big[(S + \phi_i^t + k\sigma) \bmod (\mathcal{C}/a) \mid k = \mathit{end\_after}_i \ldots (N-1)\big]\\ &\quad + \!\!+ \ \big[(S + \phi_i^t + k\sigma) \bmod (\mathcal{C}/a) \mid k = 0 \ldots (\mathit{end\_after}_i - 1)\big].\end{aligned}$$

Merging the q footprints. Given q lists of region start positions, and q lists of end positions as defined in the previous section it is straightforward to construct a new list of regions, such that no two regions overlap. The end product of this process is a sequence of triples, each of the form $(\phi, S, [\nu_1 \ldots \nu_q])$. The two values ϕ and S define the position and size of the region; $\mathbf{v}$ is a bit-vector such that $\nu_i = 1$ if the region is a subset of reference i's individual footprint. It can be seen that ψ^* is a consequence of $\mathbf{v}$, since the level of overlapping in a region is directly related to the references being accessed in that region.

The merging process is straightforward since the lists of region boundaries are known to be in ascending numerical order. A working value of $\mathbf{v}$ is maintained, initially set to reflect the references whose footprints wrap around from the end to the start of the cache. While there are elements left in any of the $2q$ lists, the list with the smallest first element is found. This element is deleted, and a footprint region is created from the previously found point to the current point, with the current value of $\mathbf{v}$. Assuming that the list refers to reference R_k; if it is the list of start points then ν_k is set to one, otherwise it is set to zero.

3.2.2 Finding the cumulative overlap of a region

After merging the reference's footprints as in the previous section the structure of the translation group's cache footprint is almost complete. Instead of the (ϕ, S, ψ^*) representation that is required, it is in the form $(\phi, S, [\nu_1 \ldots \nu_q])$.The problem then, is to calculate ψ^* given vector $\mathbf{v}$.

The average level of overlap of a reference's cache footprint has already been calculated as ψ, in (5). Using the same logic as in Section 3.2.1 all references in the translation group must have the same value of ψ.

A natural method of finding ψ^* is to simply multiply ψ by the number of bits in $\mathbf{v}$ that are set, i.e. the number of references in the region. On considering how caches work, it can be seen that this method is only guaranteed to work when no two references access the same array. If two or more references do access the same array, there is the possibility that there could be an intersection between the two sets of array elements accessed. If such an intersection occurs, these elements will only be stored in the cache *once*, not twice as predicted if we take 2ψ as the overlap of the two references combined.

This feature means that the amount of *sharing* between any two references must be examined. We define this by a ratio, ranging from zero, if they have no elements in common, to one, when all elements are accessed by both references. This ratio, *sharing*(R_x, R_y) for two references R_x and R_y, is calculated from the array footprint of the translation group—the parameters S^t, σ^t, N^t, and ϕ^t defined in Section 3.1.

Calculating ***sharing*****(R_x, R_y).** The definition of *sharing*(R_x, R_y) consists of two expressions: the degree of sharing between the two array footprints when considered as two contiguous regions, and the degree of sharing between the individual regions inside the footprints. The distinction between these two concepts is shown in Figure 4 for the two references R_x and R_y, first as single regions, then as a sequence of regions.

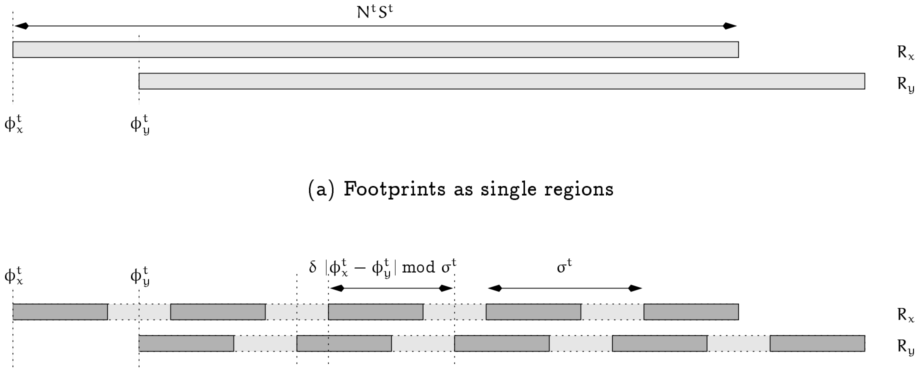
Considering the footprints as two single regions (Figure 4(a)) it can be seen that the distance between the two regions is $|\phi_x^t - \phi_y^t|$, subtracting this value from the total extent of the region $N^t\sigma^t$ gives the total number of shared elements. Hence the ratio of shared elements is $(N^t\sigma^t - |\phi_x^t - \phi_y^t|)^+/(N^t\sigma^t)$.

The level of sharing between two regions of the footprint (Figure 4(b)) is found in a similar manner. The distance between two possibly overlapping regions is $|\phi_x^t - \phi_y^t| \bmod \sigma^t$. Since overlapping could occur in either direction the smallest possible distance between overlapping regions δ is defined as,

$$\delta = \min\left(\left|\phi_x^t - \phi_y^t\right| \bmod \sigma^t, \sigma^t - \left(\left|\phi_x^t - \phi_y^t\right| \bmod \sigma^t\right)\right).$$

If $\delta \geq S^t$ then there is no sharing, otherwise $S^t - \delta$ elements are shared between the two regions. Then the ratio defining the level of sharing between the two regions is $(S^t - \delta)^+/S^t$.

Multiplying the two sharing ratios, that for the footprints as a whole and that for two regions, gives the overall ratio of shared elements between the two



(a) Footprints as single regions

(b) Footprints as multiple regions

Figure 4: Array footprint sharing

footprints, i.e.

$$\mathit{sharing}(R_x, R_y) = \left(\frac{\left(N^t\sigma^t - \left|\phi_x^t - \phi_y^t\right|\right)^+}{N^t\sigma^t} \right) \left(\frac{(S^t - \delta)^+}{S^t} \right). \tag{7}$$

Finding ψ^* of a region. The $\mathit{sharing}(R_x, R_y)$ function defined in (7) allows the combined level of overlap between two references to be found. For example if $\psi_{R_x \cup R_y}$ is the level of overlap occurring when R_x and R_y access the same region of the cache, ψ_{R_x} and ψ_{R_y} are the overlaps of the individual references, and $\psi_{R_x \cap R_y}$ is the overlap shared between R_x and R_y, then,

$$\begin{aligned} \psi_{R_x \cup R_y} &= \psi_{R_x} + \psi_{R_y} - \psi_{R_x \cap R_y}, \\ &= \psi\left(2 - \mathit{sharing}(R_x, R_y)\right). \end{aligned} \tag{8}$$

The second line of this equation follows since only references in translation are merged in this way, and the intersection is directly related to how many elements the two references share (as an average across the entire cache).

To find ψ^*, the average level of overlap across all references $\{R_i \mid \nu_i = 1\}$, it is necessary to extend the union operator shown above to include an arbitrary number of references. Considering (8) it's evident that there is a similarity between finding the combined overlap and the number of elements in a union of sets. That is, (8) is analogous to

$$|S_1 \cup S_2| = |S_1| + |S_2| - |S_1 \cap S_2|. \tag{9}$$

The general form of this expression for the number of elements in a union is,

$$\begin{aligned} |S_1 \cup \cdots \cup S_n| = {} & \sum\nolimits_1 |S_i| - \sum\nolimits_2 |S_i \cap S_j| \\ & + \sum\nolimits_3 |S_i \cap S_j \cap S_k| - \cdots \pm |S_1 \cap S_2 \cap \cdots \cap S_n|. \end{aligned}$$

where the $\sum_i$ symbol stands for the summation of all i-element combinations of $S_1 \ldots S_n$. The expression $|S_1 \cup \cdots \cup S_n|$ is analogous to $\psi_{R_1 \cup \cdots \cup R_n}$ in exactly the same way that (9) is analogous to to (8), and therefore

$$\begin{aligned} \psi_{R_1 \cup \cdots \cup R_n} &= n\psi - \sum\nolimits_2 \psi_{R_i \cap R_j} \\ &\quad + \sum\nolimits_3 \psi_{R_i \cap R_j \cap R_k} - \cdots \pm \psi_{R_1 \cap R_2 \cap \cdots \cap R_n}. \end{aligned} \tag{10}$$

It is still necessary to define the average overlap of an intersection between an arbitrary number of references. A two-reference intersection was shown in (8), this can be extended to an arbitrary number of references,

$$\psi_{R_1 \cap \cdots \cap R_n} = \psi^n \prod\nolimits_2 \mathit{sharing}(R_i, R_j)\,, \tag{11}$$

where the symbol $\prod_2$ stands for the product of all two-element combinations of R_i and R_j.

Now it is possible to find ψ^*, the average overlap of a cache footprint region containing references defined by the vector $\mathbf{v}$. Computing (10) for the references included in the region, i.e. the set $\{R_i \mid v_i = 1\}$, gives ψ^*.

3.2.3 Notes on optimizing the calculation of ψ^*

The method shown in the previous paragraphs is obviously highly combinatorial in nature. When the bit vector $\mathbf{v}$ contains n ones, the number of multiplications required is,

$$\mathrm{op}(n) = 1 + \sum_{k=2}^{n} \binom{n}{k}\binom{n}{2},$$

this grows rapidly, making computing ψ^* slow for relatively small values of n (for example $\mathrm{op}(10) \approx 4.5 \times 10^4$, and $\mathrm{op}(15) \approx 3.4 \times 10^6$). Since one of the main reasons for using analytical methods is their increased speed this is clearly undesirable. Fortunately two straightforward modifications push the combinatorial barrier back some distance.

Firstly, the value of ψ^* does not have to be completely evaluated at the boundary of each footprint region. Considering the identity,

$$\begin{aligned} |S_1 \cup \cdots \cup S_n \cup S_{n+1}| &= |S_1 \cup \cdots \cup S_n| + |S_{n+1}| - \sum\nolimits_1 |S_i \cap S_{n+1}| \\ &\quad + \sum\nolimits_2 |S_i \cap S_j \cap S_{n+1}| - \cdots \pm |S_1 \cap \cdots \cap S_n \cap S_{n+1}|\,, \end{aligned}$$

shows that ψ^* can be adaptively calculated from the previous region's value when a single reference enters or leaves the union. This approximately halves the number of multiplications required.

Secondly, since one of the constraints of the model is that an array may not overlap any other arrays, there can be no sharing of data elements between references accessing different arrays. This means that only a *subset* of vector $\mathbf{v}$ need be examined when computing ψ^*—those where $v_i = 1$ *and* where reference R_i accesses the same array as that accessed by the array reference whose state changed at the region boundary. Depending upon the distribution of array references to arrays, this modification can decrease the complexity of the ψ^* calculation by orders of magnitude.

4 Modeling spatial interference

As noted in Section 2 the temporal and spatial cache effects of an array reference are modeled separately. Spatial reuse occurs when more than one element in a cache line is accessed before the data is ejected from the cache. For a reference R the innermost loop on which spatial reuse may occur is labeled l_s, where

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

The *spatial miss ratio* of a reference, labeled M_s, is defined such that multiplying it by the predicted number of temporal misses suffered by a reference predicts the actual number of cache misses occurring. This ratio encapsulates all spatial effects on the reference, and is found by combining four more specific miss ratios: the compulsory miss ratio C_s, the self interference miss ratio S_s, the internal cross interference miss ratio I_s, and finally the external cross interference ratio E_s,

$$M_s = \min\left(1, \max(C_s, S_s) + I_s + E_s\right).$$

The value of C_s for a particular reference follows directly from the array footprint of the reference defined over all loops $1 \ldots n$. It is the ratio between the number of cache lines in each footprint region and the number of referenced elements within each region.

When studying the level of interference affecting a spatial reuse dependence it is necessary to examine what happens between each iteration of loop l_s. Figure 5 illustrates this for self interference. The left hand side of the figure shows a square matrix Y being accessed by the array reference $Y(2j_1, j_2)$; on the right is shown how this maps into the cache, both over time and for a complete iteration of loop j_1 (assuming a 4-way associative cache). The elements that may interfere with $Y(6,0)$ reusing the data loaded into the cache by $Y(4,0)$ are shaded. The three types of spatial interference are considered in the following sections.

4.1 Calculating spatial self interference

As shown in Figure 5 the reference being modeled can obstruct its own spatial reuse; this happens when the number of data elements accessed on a single iteration of loop l_s that map to a particular set in the cache is greater than the level of associativity. To analyze this mapping process the recurrence shown in (2) is used, but with slightly different array footprint parameters. The distance between each footprint region σ^t is defined by the distance between elements accessed on successive iterations of loop l_s (see Figure 5), and the size of each

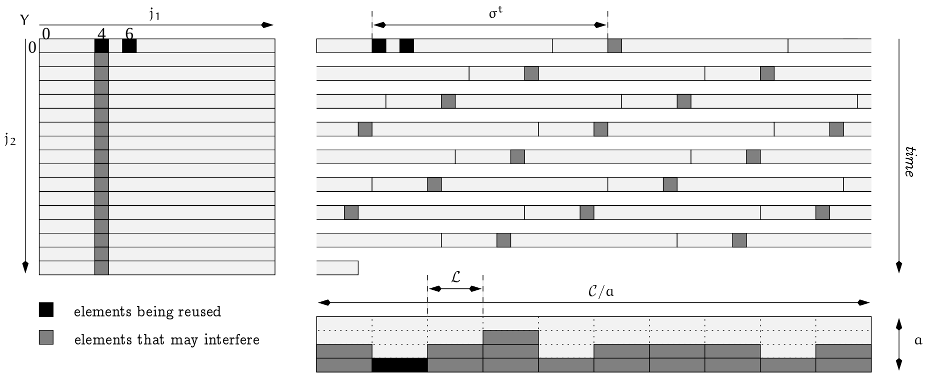


Figure 5: Example of spatial reuse from $Y(2j_1, j_2)$

footprint region is defined as the size of a cache line $\mathcal{L}$ to ensure that interference between lines is detected.

As in Section 3.1, the result of the mapping process is that the cache is divided into σ_0/σ_{s-1} areas of size σ_{s-1}; each with a certain number of footprint regions, each a distance $\tilde{\sigma}_s$ from the previous. There are r areas that contain $n_s + 1$ regions, and $(\sigma_0/\sigma_{s-1}) - r$ areas containing n_s (Section 3.1).

By examining each of the two types of area separately, calculating the value of S_s in each, and combining the two values, it is possible to predict the overall level of self interference,

$$S_s = 1 - \left(f_s(n_s) \frac{(\lfloor \sigma_0/\sigma_{s-1} \rfloor - r)(n_s)}{N^t} + f_s(n_s + 1) \frac{(r)(n_s + 1)}{N^t} \right)$$

where the function $f_s(x)$ gives the probability that an element in an area of size σ_{s-1}, containing x elements, does *not* suffer from spatial interference.

Defining $f_s(x)$**.** It is immediately possible to identify two special cases,

1. if $\tilde{\sigma}_s = 0$ then all elements in the area occupy the same cache set; if the number of elements x is greater than the level of associativity interference occurs, thus

$$f_s(x) = \begin{cases} 0 & \text{if } x \leq a \\ 1 & \text{if } x > a. \end{cases} \qquad \text{when } \tilde{\sigma}_s = 0.$$

2. if there is only one element per set and no overflow between neighboring areas, then reuse must be total,

$$f_s(x) = 1, \qquad \text{when } \tilde{\sigma}_s \geq \mathcal{L} \text{ and } x\tilde{\sigma}_s < \sigma_{s-1}.$$

In the general case the solution is not so straightforward, the main complication being the possibility that the distance from the first to the last element in the area (i.e. $x\tilde{\sigma}_s$) is greater than the size of the area itself, and therefore the elements "wrap-around" the end of the area, possibly interfering with those at the start.

To handle this a hybrid analytical-simulation technique is used: each of the x elements in the area has $\mathcal{L}$ different positions in a cache line where it might occur, each position is analyzed for whether reuse can occur or not, leading to the overall probability of reuse for that element. Repeating for the other $x-1$ elements, and combining all the individual probabilities gives the value of $f_s(x)$.

For an element y from $0 \ldots x-1$, it is possible to list the positions in the cache of the elements surrounding it,

$$\begin{aligned}
\mathit{points}(y) &= \mathit{before}(y) +\!\!+\ \mathit{after}(y) \\
\mathit{before}(y) &= [k\tilde{\sigma}_s \mid k = 0 \ldots (y-1)] \\
\mathit{after}(y) &= [k\tilde{\sigma}_s + \delta \mid k = (y+1) \ldots (x-1)] \\
\delta &= \begin{cases} -\mathit{stride} & \text{if } \sigma_s > 0 \\ +\mathit{stride} & \text{if } \sigma_s < 0 \end{cases}
\end{aligned}$$

where the *stride* of a reference is the distance between elements accessed on successive iterations of the spatial reuse loop l_s.

The essence of the problem is now as follows. From $\mathit{points}(y)$, deduce the number of points that occur in the cache line-sized region $z \ldots (z+\mathcal{L})$, given that the points wrap around to zero at position σ_{s-1}. A generalized form of the series defined above is

$$[kA + B \mid k = 0 \ldots (C-1)],$$

with,

$$\begin{aligned}
\mathit{before}(y) &\Rightarrow & A &= \tilde{\sigma}_s, & B &= 0, & C &= y, \\
\mathit{after}(y) &\Rightarrow & A &= \tilde{\sigma}_s, & B &= (y+1)\tilde{\sigma}_s + \delta, & C &= (x-y)-1.
\end{aligned}$$

For this general series the number of points within an interval $z \ldots (z+\mathcal{L})$, including the wrap around effect, is given by,

$$\mathit{incl}(z) = \sum_{i=\left\lceil \frac{B}{\sigma_{s-1}} \right\rceil}^{\left\lceil \frac{(C-1)A+B}{\sigma_{s-1}} \right\rceil} \left(\min\left(C, \left\lceil \frac{z_i - B}{A} \right\rceil^+ \right) - \min\left(C, \left\lceil \frac{z_i + \mathcal{L} - B}{A} \right\rceil^+ \right) \right)$$
$$\text{where } z_i = z + i\sigma_{s-1}.$$

Thus to find the total number of elements within a particular cache-line sized interval the above expression is evaluated for both $\mathit{before}(y)$ and $\mathit{after}(y)$, so that the total number of elements in a particular interval $z \ldots (z+\mathcal{L})$ is

$$\mathit{incl}(z, \mathit{before}(y)) + 1 + \mathit{incl}(z, \mathit{after}(y)).$$

If this value, the number of elements in a particular line, is greater than the level of associativity a, then self interference occurs; by averaging over the $\mathcal{L}-1$ possible positions for the start of a line containing the interval y, the probability of reuse can be found. By repeating this process for the $x-1$ other elements in the area the overall probability, and hence S_s, can be calculated.

4.2 Internal spatial cross interference

As well as being caused by the reference itself, spatial interference may also arise due to the other references in the same translation group. When the number of data elements mapping to a particular cache set, on a single iteration of loop l_s, is greater than the level of associativity a, interference will occur. This phenomena is often referred to as "ping-pong" interference, and may affect performance massively since it is possible for *all* spatial reuse by the reference to be prevented.

When considering a reference R, ping-pong interference is detected by calculating the cache footprint of all references in the translation group, for a single iteration of the spatial reuse loop (i.e. let $\mathcal{N}_{1\dots l}=1$). Considering only the regions where $\psi^* > a$, if any are less than $\mathcal{L}$ elements from the position of the first element accessed by R, i.e. $\phi_R \bmod (\mathcal{C}/a)$, then ping pong interference occurs.

Assuming that the closest footprint region before $\phi_R \bmod (\mathcal{C}/a)$ is δ_b positions away, and the closest region after R is δ_a positions away, then the miss ratio due to internal interference is defined as follows,

$$I_s = \min\left(1, \left(1-\frac{\delta_a-1}{\mathcal{L}}\right)^+ + \left(1-\frac{\delta_b}{\mathcal{L}}\right)^+\right).$$

4.3 External spatial interference

After considering the interference from the reference's own translation group, interference from the other translation groups—external interference—must be modeled. Each group is examined in turn, the overall miss ratio due to external interference E_s being the sum of each group's individual external interference ratio.

For a reference R, with spatial reuse on loop l_s, the probability P_R, that accessing a random data element will find an element in a set containing data spatially reused by R, is defined by,

$$P_R = \frac{N_R S_R}{\mathcal{C}/a},$$

where N_R and S_R are the number and size of regions in reference R's cache footprint on loop l_s respectively (see Section 3.1).

Restricting the iteration space to a single iteration of loop l_s (i.e. let $\mathcal{N}_{l_s}=1$), the cache footprint of each translation group (of which R is not a member) is examined. By counting the number of elements in these footprints that could cause spatial interference on R, and multiplying by P_R, a prediction of the number of misses is made.

If the average level of overlap for the translation group containing R is ψ_R, and the footprint of each other translation group is represented by a sequence of (ϕ, S, ψ^*) triples, then an individual footprint region can possibly interfere with R only if $\psi_R + \psi^* > a$. Also, if interference does occur, the number of cache misses for that set can not be greater than the actual number of elements in the set. This leads to the definition of the following function giving the "miss overlap",

$$\psi\text{-miss}\left(\psi_R, \psi^*\right) = \min\left(\psi_R, \left(\psi_R + \psi^* - a\right)^+\right). \tag{12}$$

Mapping this function, multiplied by the size of each region, over the cache footprint of each translation group gives the total number of elements accessed by the group that might cause a cache miss[4]. Multiplying this value by P_R, and dividing by the total number of iterations made by loop l_s, gives the external miss ratio for a single translation group G,

$$E_s(G) = \frac{P_R \sum_* \left(\psi\text{-miss}\left(\psi_R, \psi^*\right) \times S\right)}{\mathcal{N}_{l_s+1} \times \cdots \times \mathcal{N}_n},$$

where the symbol $\sum_*$ stands for the summation across all of the translation group's cache footprint regions (ϕ, S, ψ^*). By summing $E_s(G)$ over all translation groups G, such that $R \notin G$, the overall value of E_s is found.

5 The cache behavior of a self dependence

As noted in Section 2, a self dependence occurs when an array reference accesses particular data elements more than once. This happens when one or more of the loop variables $j_1 \ldots j_n$ are not used by the reference. For example, the array reference $A(j_3, j_1)$ does not use j_2, and therefore all iterations of loop 2 access exactly the same set of elements, namely $\{A(0, j_1) \ldots A(\mathcal{N}_3 - 1, j_1)\}$. The innermost loop on which reuse occurs is defined as loop l, where $l = \{\max k \mid \mathcal{A}_k = 0\}$.

In theory, each time loop l is entered the first iteration would load the referenced elements into the cache, and subsequent iterations reuse them. That the first iteration of loop l *must* load the elements gives the number of compulsory misses,

$$M_s \prod_{k=1}^{l-1} \mathcal{N}_k \prod_{k=l+1}^{n} \mathcal{N}_k, \tag{13}$$

that is: the spatial miss ratio, multiplied by the number of times loop l is entered, multiplied by the number of unique elements referenced.

But the cache capacity is limited—it may not be possible to hold all elements referenced by loop l in the cache at once. This factor is not only dependent on whether the size of the cache is greater than the number of elements, as with spatial reuse the accessed elements may map into the cache in such a way as to prevent reuse. Although using a cache with high associativity can prevent

[4] When the referenced array is significantly smaller than the number of sets in the cache, only footprint regions that actually overlap with the array are considered.

interference in certain cases, as the number of elements accessed increases the problem may return.

5.1 Self interference

Self interference on a reference is modeled by mapping the array footprint of the elements accessed by a single iteration of loop l into the cache, removing those elements that fall in sets with overlap greater than the level of associativity. Subtracting the number of elements left from the original number of elements gives the number of cache misses per iteration.

We use the same mapping process as shown in Section 3.1, with one important modification, the function $f_i(x)$ is replaced by $f_r(x)$ (and the way in which ψ is calculated is changed to reflect this). Whereas $f_i(x)$ gave the number of sets that could interfere in an area containing x regions, $f_r(x)$ gives the number that can be reused, i.e. those where $\psi \leq a$. Given $f_r(x)$ the number of reusable elements in the footprint follows as $NS\psi$, and therefore the total number of cache misses due to self interference is

$$M_s \left(\prod_{k=1}^{l-1} \mathcal{N}_k \right) (\mathcal{N}_l - 1) \left(\left(\prod_{k=l+1}^{n} \mathcal{N}_k \right) - NS\psi \right)$$

—the number of times loop l is entered multipled by the number of cache misses each iteration (excluding when $j_l = 0$, which is handled by the compulsory miss calculation shown in (13)).

The definition of function $f_r(x)$ uses a similar method to that shown in Section 4.1 for calculating spatial self interference. The structure of the cache section being examined was described in Section 3.1; an area of size σ_{s-1} containing x regions of size S^t, each at an interval $\tilde{\sigma}_s$ from the previous. The first region is located at the beginning of the area, and the regions wrap around the end of the area (i.e. the position in the area of region k is actually $(k\tilde{\sigma}_s) \bmod \sigma_{s-1}$).

For an area with this structure, the function $f_r(x)$ must calculate the number of positions in which the level of overlap is less than or equal to the level of associativity, i.e. where no interference occurs. For a single position z in the area, the level of overlap (i.e. the number of regions crossing this point) is given by the number of regions beginning before this point minus the number of regions ending before it. To include the wrapping effect this expression is summed over all possible "wrap arounds" in which a region appears, i.e.,

$$\mathit{overlap}(z) = \sum_{i=0}^{\left\lceil \frac{f_l(x)}{\sigma_{s-1}} \right\rceil - 1} \min\left(x, \left\lfloor \frac{z_i}{\tilde{\sigma}_s} \right\rfloor + 1\right) - \left\lceil \frac{(z_i - S^t)^+}{\tilde{\sigma}_s} \right\rceil \qquad (14)$$
$$\text{where } z_i = z + i\sigma_{s-1}.$$

A possible definition $f_r(x)$ would be to test every position in the area, i.e. $z = 0 \ldots (\sigma_{s-1} - 1)$, and count the number of times that $\mathit{overlap}(z) \leq a$. Fortunately there is a more efficient method: since there are only x footprint regions, the value of $\mathit{overlap}(z)$ can only change a maximum of $2x$ times (at the start and end of each region). Using a similar method to when finding the one-dimensional footprint of a translation group (see Section 3.2.1), these $2x$ positions are enumerated in ascending order, and the atomic regions they define are examined.

Finally, the definition of ψ in (5) includes positions in the area where reuse cannot occur (since it is still relevant when calculating interference). However, when looking at the reuse of a footprint it is necessary for ψ to be the average overlap of the positions in the footprint where reuse *does* occur. This can be calculated while computing the value of $f_r(x)$.

5.2 Internal cross interference

After examining the level of self interference on a self dependent reference the cache footprint of the data not subject to self interference is known; characterized by the parameters S, σ, N and ψ. It is still uncertain whether or not these regions of the cache can be reused since data accessed by the other array references in the loop nesting may map to the same cache sets, possibly preventing reuse.

Interference from other references in the same translation group is considered first. The cache footprint of these references is identified (using the techniques shown in Section 3) and then compared region by region with the footprint of the data not subject to self interference. Interference can only occur wherever the two footprints overlap, and only when the combined level of overlap is greater than the level of associativity, that is when $\psi + \psi^* > a$. Assuming that two footprint regions overlap for *size* positions, then the number of misses occurring on each iteration of loop l is

$$\textit{size} \times \psi\text{-miss}\,(\psi, \psi^*) \times M_s.$$

The summation of this expression over all sections of the cache where two footprint regions overlap gives the total number of cache misses on each iteration of the reuse loop; multiplying by $\mathcal{N}_1 \ldots \mathcal{N}_l$ gives the total number of misses.

To increase the accuracy of the next stage—predicting the level of external interference—the values of NS and ψ (the number of reusable positions and average overlap) are adjusted to take account of internal interference. The number of reusable positions after considering internal interference NS' is the combined size of all regions where interference doesn't occur, and the adjusted overlap ψ' is the average value of $\psi + \psi^*$ across all these regions.

5.3 External interference

The final source of temporal interference on a self dependence to be considered is external cross interference. This is interference arising from references in other translation groups to the reference being examined. Unlike when modeling internal cross interference, it is not possible to simply compare the two cache footprints (the reference's possibly reusable data, and the footprint of the interfering translation group) exactly because they are not in translation. The footprints are "moving" through the cache in different ways and hence incomparable. Instead, a statistical method is used, based on the dimensions of the two footprints—the total size and the average overlap.

Similarly to when modeling external interference on spatial dependences (see Section 4.3) each external translation group is considered in turn. The number of footprint positions that could possibly cause interference are found by summation over the cache footprint of the group. To find the average number of cache misses this quantity is multiplied by the size of the reusable footprint and divided by the number of possible positions,

$$\text{external misses} = M_s \frac{NS' \left(\sum_{*} \left(S \times \psi\text{-miss}\left(\psi', \psi^{*}\right) \right) \right)}{\mathcal{C}/a}.$$

This gives the number of misses on each iteration of loop l caused by a particular translation group. Summing this expression over all external groups and multiplying by the total number of iterations of loop l gives the actual number of cache misses due to external interference.

6 Modeling group dependences

A group dependence occurs when an array reference reuses data that was most recently accessed by another reference in the same translation group. For a reference R the reference that it is dependent upon is denoted $\vec{R}$; Section 2.2 has described how dependences are identified.

The definition of the spatial miss ratio given in Section 4 must be altered slightly to model group dependences, it must also include any spatial group reuse occurring. This is when R is in the same cache line as $\vec{R}$ a certain number of times per every $\mathcal{L}$ elements accessed. If the constant distance between the two references, $\mathcal{B}_{\vec{R}} - \mathcal{B}_R$, is less than the size of a cache line, then this is the number of times that R must load an element itself per cache line. Therefore the actual spatial miss ratio is defined by,

$$M_s' = M_s \frac{\mathcal{L} - \left(\mathcal{B}_{\vec{R}} - \mathcal{B}_R\right)}{\mathcal{L}}.$$

The number of compulsory misses is defined by the number of elements accessed only by R, not by $\vec{R}$, multiplied by the spatial miss ratio. Since the *sharing*(R_x, R_y) function defined in (7) gives the ratio of elements shared between R_x and R_y, we have that

$$\text{compulsory misses} = M_s' \left(\prod_{k=1}^{n} \mathcal{N}_k \right) \left(1 - \textit{sharing}\left(R, \vec{R}\right) \right).$$

For a reference R, the innermost loop on which group reuse occurs is defined as

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

where m is the number of dimensions in the array being accessed. To identify cross interference on a group dependence it is only necessary to examine the period between $\vec{R}$ accessing an arbitrary element and R reusing it. This is defined as δ_g iterations of loop l_g:

$$\delta_g = \beta_k(\vec{R}) - \beta_k(R),$$

with k the innermost dimension of the array where the β_k constants of the two references differ.

Consider for example, the case when $R = A(j_2, j_1)$ and $\vec{R} = A(j_2, j_1 + 2)$. Here $l_g = 1$, and $\delta_g = 2$, that is, after $\vec{R}$ accesses element $A(j_2, 2)$, two iterations of loop 1 pass before R accesses the same element. Interference occurs if the element has been ejected from the cache during these two iterations.

6.1 Internal interference

Internal cross interference is found by examining the cache footprint of the translation group of R for the first δ_g iterations of loop l_g, i.e. the iteration space with $\mathcal{N}_1 \ldots \mathcal{N}_{l_g-1} = 1$ and $\mathcal{N}_{l_g} = \delta_g$. For each region in the footprint that contains data accessed by R the probability of interference is calculated, the maximum probability across the whole footprint is then the actual probability of internal interference. For a footprint region with average overlap ψ^*, this probability is defined as,

$$P_i(\psi^*) = \min(1, \psi^* - a)^+,$$

i.e. for interference to definitely occur $\psi > a + 1$, while if $\psi < a$ interference definitely doesn't occur; there is a gradient between these two certainties.

The number of cache misses is defined as the number of elements that could theoretically be reused, multiplied by the maximum value of $P_i(\psi^*)$ and the spatial miss ratio,

$$\text{int. misses} = M_s' \left(\prod_{k=1}^{l_g} \mathcal{N}_k \right) \mathit{sharing}\left(R, \vec{R}\right) \left(\max_* P_i(\psi^*)\right).$$

6.2 External interference

When the maximum value of P_i is less than 1, and therefore internal interference is not total, external cross interference must also be considered. Again the iteration space is defined as δ_g iterations of loop l_g, but this time the cache footprints of the translation groups that R is not a member of are examined.

For each such group, the number of cache misses caused is found by counting the number of positions in its footprint where interference may occur, and applying the same probabilistic method used when predicting external interference on a self dependence (see Section 5.3). Assuming that the cache footprint of the translation group containing R has an average overlap of ψ' in the regions containing data accessed by R (this can be calculated while finding internal interference), then a footprint region with overlap ψ^* may possibly cause interference if $\psi' + \psi^* > a$. The actual number of misses per translation group is defined as

$$\begin{aligned}\text{ext. misses} = M_s' &\left(\sum_{\psi' + \psi^* > a} \left(S \times \psi\text{-miss}\left(\psi^*, 1\right)\right) \right) \\ &\times \left(\mathcal{C}/a\right) \left(1 - \max P_i\right) \left(\prod_{k=1}^{l_g - 1} \mathcal{N}_k \right) \left(\mathcal{N}_{l_g} - \delta_g\right).\end{aligned}$$

7 Example results

To demonstrate the validity and benefits of the techniques described, this section presents experimental results obtained using an implementation of the model. Code fragments are expressed in a simple language which allows the details of the arrays being accessed, the loop structures, and the array references themselves to be specified. Here three examples typical of nested computations are shown, chosen for their contrasting characteristics to ensure that all parts of the cache model are exercised. Each manipulates matrices of double precision values, arranged in a single contiguous block of memory. They are:

1. A matrix-multiply, consisting of three nested loops, containing four array references in total. Each reference allows temporal reuse to occur within one of the loops, one reference may be subject to considerable spatial interference. The Fortran code is shown in Figure 6(a).

2. A "Stencil" operation, from [10]. This kernel shows group dependence reuse, and doesn't always access memory sequentially. See Figure 6(b).

3. 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, and contains significantly more array references than the others. The matrices IVX and IVY contain 32-bit integers. See Figure 6(c).

Each example kernel has been evaluated for a range of cache parameters, comparing the predicted miss ratio against that given by standard simulation techniques[5]. The average percentage errors are shown in Table 1.

The results for $\mathcal{C} = 16384, \mathcal{L} = 16$, and for $\mathcal{C} = 32768, \mathcal{L} = 16$ are shown in Figure 7 for the three example kernels. Miss ratio and absolute error are plotted against the width and height of the matrices. Also shown, in Table 2, are the range of times taken to evaluate each problem on a 167MHz SUN ULTRA-1 workstation, for a single cache configuration.

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

```
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-2
  DO J = 0, N-2
    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-2
  DO I = 1, N-2
    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 6: Example kernels

8 Discussion

The experimental data presented in the previous section shows that the predictions made by the model are generally very accurate: the majority of average errors are within ten percent, with all but three of the fifty four examples having average errors of less than fifteen percent. When combined with the increased speed of prediction we believe that the analytical approach is more practical than simulation when examining the individual kernels of an application.

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 by several orders of magnitude, even with the smallest problem sizes. As the number of memory references grows the gulf widens: simulation time increasing proportionally to the number of accesses, the time needed to evaluate the analytical model staying mostly constant. The Jacobi example is the slowest to evaluate analytically because it has eighteen array references to evaluate, compared to Stencil's six and the matrix multiply's four. Even so, the combinatorial effects that might have been feared are not a problem.

It is also clear from the miss ratio plots that using set-associative caches

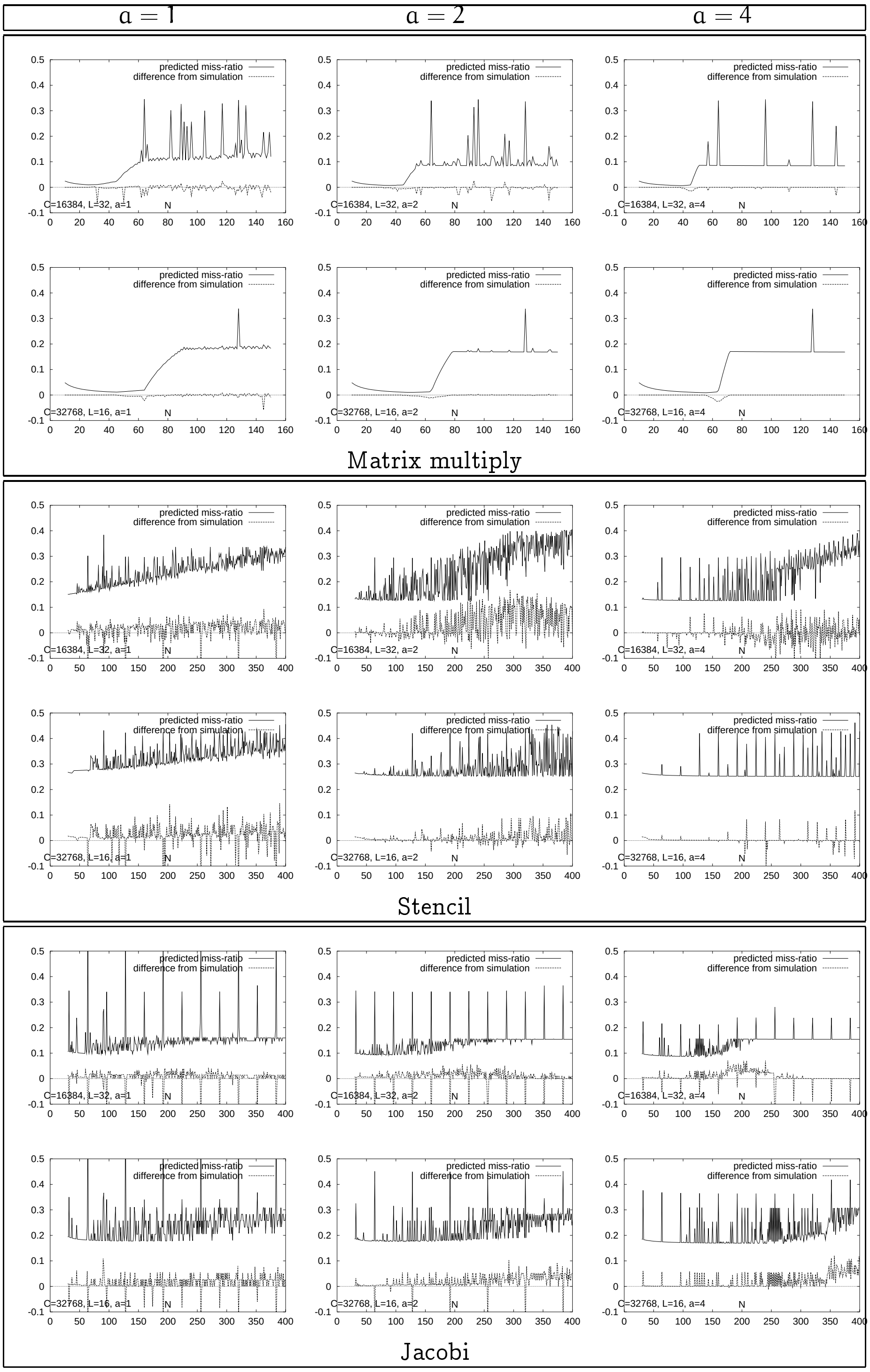


Figure 7: Predicted miss ratios and absolute errors for $\mathcal{C} = 16384, \mathcal{L} = 32$, and $\mathcal{C} = 32768, \mathcal{L} = 16$ configurations.

		$\mathcal{C}$	8192		16384		32768	
Experiment	a	$\mathcal{L}$	16	32	16	32	16	32
	1		5.79	9.36	4.99	6.97	4.99	7.19
Matrix	2		3.85	7.02	4.07	6.22	4.89	6.12
Multiply	4		2.42	4.89	3.29	3.51	3.90	3.97
	1		12.1	11.2	12.9	12.3	10.1	11.8
Stencil	2		16.1	17.9	12.5	19.2	4.78	9.60
	4		10.9	13.4	6.17	8.78	1.73	3.64
	1		7.48	9.57	10.3	11.8	9.56	10.8
Jacobi	2		6.04	7.75	11.1	12.5	10.7	11.8
	4		4.72	6.99	9.16	10.5	9.34	10.2

Table 1: Average percentage errors of example predictions when compared with simulated results.

	Analytical Model			Simulation		
Experiment	Min.	Max.	Mean	Min.	Max.	Mean
Matrix mult.	0.00093	0.018	0.0014	0.0058	18.20	4.93
2D Jacobi	0.0095	0.019	0.010	0.019	4.21	1.50
Stencil	0.0016	0.018	0.0029	0.0079	1.45	0.52

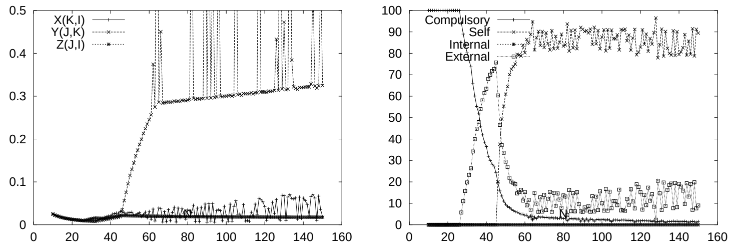
Table 2: Calculation times for $\mathcal{C} = 16384, \mathcal{L} = 32, a = 2$ experiments (seconds.)

does not avoid the problem of cache interference. Even for a 4-way associative cache there are still large variations in miss ratio, especially in the Stencil and Jacobi kernels, i.e. as the number of array references increases. By using using well known techniques such as padding array dimensions and controlling base addresses, guided by an analytical model such as presented here, the variations can be reduced to decrease the miss ratio.

A benefit of using analytical models that has not yet been mentioned is the extra information available through using analytical models. When trying to lower the number of cache misses in a program it is important to know both where and why the cache misses occur. Due to the structure of the method presented in this paper both requirements can be met simply by examining the outputs of the component models. For example, with the matrix multiply kernel we can examine both the miss ratio of each reference (Figure 8(a)), and the miss ratio due to each type of interference (Figure 8(b)). These show that the vast majority of the misses are due to reference `Y(J,K)`, and that between 80 and 90 percent of the interference is self interference (in this case spatial self interference, due to array `Y` being accessed non-sequentially).

9 Conclusions

A hierarchical method of classifying cache interference has been presented, for both self and group dependent reuse of data, considering both temporal and



(a) Reference miss ratios

(b) % miss ratio by type

Figure 8: Examining the Matrix multiply, $\mathcal{C} = 16348, \mathcal{L} = 32, a = 1$.

spatial forms. Analytical techniques of modeling each category of interference have been developed for array references in loop nestings. It has been shown that these techniques 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 dependent on the number of array references in the program, rather than the actual number of memory accesses (as with 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 include such optimization strategies, as well as extensions to the model itself. It is also intended to use the techniques as part of a general purpose performance modeling system [11].

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] Anant Agarwal, Mark Horowitz, and John Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, May 1989.

[2] François Bodin and André Seznec. Skewed associativity improves program performance and enhances predictability. *IEEE Transactions on Computers*, 46(5):530–544, May 1997.

[3] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organisation and data layout. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, volume 30, pages 279–289, June 1995.

[4] Thomas Fahringer. Automatic cache performance prediction in a parallelizing compiler. In *Proceedings of the AICA'93 — International Section*, September 1993.

[5] Christine Fricker, Olivier Temam, and William Jalby. Influence of cross-interferences on blocked loops: A case study with matrix-vector multiply. *ACM Transactions on Programming Languages and Systems*, 17(4):561–575, July 1995.

[6] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 11th ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[7] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. Predicting the cache miss ratio of loop-nested array references. Research Report CS-RR-336, Department of Computer Science, University of Warwick, Coventry, UK, December 1997.

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

[9] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, California, 1991.

[10] Kathryn S. McKinley and Olivier Temam. A quantitative analysis of loop nest locality. In *Proceedings of the 7th Conference on Architectural Support for Programming Languages and Operating Systems*, volume 7, Cambridge, MA, October 1996.

[11] E. Papaefstathiou, D. J. Kerbyson, G. R. Nudd, and T. J. Atherton. An overview of the CHIP^{3}S performance toolset for parallel systems. In *Proc. of the ISCA Int. Conf. on Parallel and Distributed Computing Systems*, page 527, Orlando, September 1995.

[12] Olivier Temam, Christine Fricker, and William Jalby. Cache interference phenomena. In *Proceedings of ACM SIGMETRICS*, pages 261–271, 1994.

[13] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):129–170, June 1997.

[14] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 30–44, June 1991.