Randomized Cache Placement for Eliminating Conflicts

Nigel Topham

Department of Computer Science, University of Edinburgh
JCMB, Kings Buildings, Edinburgh (UK)

e-mail:npt@dcs.ed.ac.uk

Antonio González

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
c/ Jordi Girona 1-3, 08034 Barcelona (Spain)

e-mail:antonio@ac.upc.es

Abstract

Applications with regular patterns of memory access can experience high levels of cache conflict misses. In SMP systems using SPMD programming models the levels of conflict misses can be increased significantly by the data transpositions required for parallelization. Furthermore, techniques such as blocking which are introduced within a single thread to improve locality, can result in yet more conflict misses. The tension between minimizing cache conflicts and the other transformations needed for efficient parallelization leads to complex optimization problems for parallelizing compilers.

This paper begins with a survey and quantitative evaluation of the existing proposals for conflict-resistant cache architectures. We show that the introduction of a pseudo-random element into the cache index function provides significant performance benefits. In effect one can eliminate repetitive conflict misses and produce a cache where miss ratio depends solely on working set behavior. We show empirically that one particular pseudo-random indexing function, when used in conjunction with address prediction, yields the best overall performance.

In many systems the processor clock period is closely linked to the critical path through the first-level cache. In this paper we consider the impact of pseudo-random cache indexing on processor cycle times and present practical solutions to some of the major implementation issues for this type of cache.

We present results from detailed simulations of a superscalar out-of-order processor executing the Spec95 benchmarks, as well as from cache simulations of individual loop kernels to illustrate specific effects. We present measurements of Instructions committed Per Cycle (IPC) when comparing the performance of different cache architectures on whole-program benchmarks such as the Spec95 suite. In addition, miss ratio measurements are presented for some loop kernels to highlight the extra conflicts introduced by loop transformations and the elimination of these misses by our proposed cache.

1 Introduction

If the upward trend in processor clock frequencies during the last ten years is extrapolated over the next ten years, we will see clock frequencies by a factor of twenty during that period [35]. However, based on the current 7% per annum reduction in DRAM access times [17], memory latency can be expected to reduce by only 50% in the next ten years. This potential ten-fold increase in the distance to main memory has serious implications for the design of future cache-based memory hierarchies as well as for the architecture of memory devices themselves.

There are many options for an architect to consider in the battle against memory latency. These can be partitioned into two broad categories - latency reduction and latency hiding. Latency reduction techniques typically rely on caches to exploit locality with the objective of reducing the latency of each individual memory reference. Latency hiding techniques exploit parallelism to overlap memory latency with other operations and thus “hide” it from a program’s critical path. Increasing transistor densities also offer the possibility of large quantities of integrated DRAM, with the potential to eliminate the costly off-chip processor-memory communication delays [25]. However, regardless of the techniques developed to reduce cache miss penalties and increase memory bandwidth, the fundamental effectiveness of the top-most level of cache will remain of critical importance.

This paper examines the degree to which future cache architectures can isolate their processor from the unpredictable miss ratios caused by cache conflicts. Our contribution is prefaced with a discussion of the causes of cache conflicts, not only those inherent in an application but also those which are introduced by the processes of optimization and program transformation.

We discuss the theory, and evaluate the practice, of using alternative index functions aimed at conflict avoidance. We demonstrate how such caches could be constructed and discuss some practical solutions to the problems associated with implementing unconventional indexing schemes.

In section 2 we present an overview of the causes of conflict misses. Then, section 3 presents quantitative data about the performance of previously proposed indexing schemes. The proposed pseudo-random indexing schemes are introduced in section 4 and their application to different cache architectures is analysed in section 5. Section 6 evaluates the potential of such indexing schemes to avoid conflict misses. In section 7 we discuss a number of implementation issues, such as the effect of using this novel indexing scheme on the processor cycle time. Then, in section 8, we present an experimental evaluation of how the proposed indexing scheme affects IPC (instructions committed per cycle). Finally, in section 9, we draw conclusions from this study.

2 The causes of cache conflicts

Each block of main memory can be placed in exactly one set of blocks in the cache. The chosen set is determined by the *indexing* function. Conventional caches typically extract a field of m bits from the address and use this to select one block from a set of 2^m. Whilst simple, and easy to implement, this indexing function is not robust. The principal weakness of this function is its susceptibility to repetitive conflict misses. For example, if C is the number of cache sets and B is the block size, then addresses A_1 and A_2 map to the same cache set if $|A_1/B|_C = |A_2/B|_C$. If A_1 and A_2 collide on the same cache set, then addresses $A_1 + k$ and $A_2 + k$ also collide in cache, for any integer k, except when

$$m_1 < B - |k|_B \le m_2 \qquad \text{(i)}$$

where

$$m_1 = \min(|A_1|_B, |A_2|_B) \qquad \text{(ii)}$$

and

$$m_2 = \max(|A_1|_B, |A_2|_B) \qquad \text{(iii)}$$

There are two common cases when this happens:

- when accessing a stream of addresses $\{A_0, A_1, \ldots, A_m\}$ if A_i collides with A_{i+k}, then there may be up to $(m-k)$ conflict misses in this stream.
- when accessing elements of two distinct arrays b_0 and b_1, if $b_0[i]$ collides with $b_1[j]$ then $b_0[i+k]$ collides with $b_1[j+k]$ except under the conditions outlined above.

w-way set-associativity can help to alleviate such conflicts. However, if a working set contains $p > w$ conflicts on some cache set, then associativity can only eliminate at most w of those conflicts.

One of the best ways to control locality in dense matrix computations with large data structures is to use a tiled (or "blocked") algorithm. This is effectively a re-ordering of the iteration space which increases temporal locality. There is a plethora of literature on the subject of tiling, recently focusing on the problem of "induced conflicts". Consider the sequence of word addresses shown in figure 1, from a square TxT tile obtained from an NxN matrix:

$$\begin{bmatrix} a & a+1 & \ldots & a+T-1 \\ a+N & a+N+1 & \ldots & a+N+T-1 \\ a+2N & a+2N+2 & \ldots & a+2N+T-1 \\ \ldots & \ldots & \ldots & \ldots \\ a+(T-1)N & \ldots & \ldots & a+T(N-1)+(N-1) \end{bmatrix}$$

Figure 1. Conflicting addresses in a tiled 2D array.

If, for example, the shaded addresses a and $a+2N+2$ differ by an integer multiple of the cache capacity they will collide in a conventional cache introducing a "self conflict". Worse still, if a and $a+2N+2$ collide then $a+k$ and $a+2N+2+k$ will also collide, for any k satisfying the inequalities in (i). Measurements by Lam, Rothberg and Wolf indicate that such self-conflicts can be a serious problem [22]. In practice, until now, this has meant that compilers that tile loop nests really ought to compute the maximal conflict-free tile size for given values of B, N and cache capacity C. Often this will be too small to make it worthwhile tiling a loop, or perhaps the value of N will not be known at compile time. Gosh *et al.* [15] present a framework for analyzing cache misses in perfectly-nested loops with affine references. They develop a generic technique for determining optimum tile sizes, and methods for determining array padding sizes to avoid conflicts. These methods require solutions to sets of linear Diophantine equations and depend upon there being sufficient information at compile time to find such solutions.

There are two fundamental difficulties with software methods for conflict avoidance. Firstly, when tiling one often finds that the maximal conflict-free tile size is too small to make tiling beneficial. Secondly, there are many constraints on the types of loop that can be analyzed and modified statically by software. For example, the array dimensions or base addresses may not be known at compile time. Perhaps the loops are not perfectly nested, or maybe they contain non-inlined subroutine calls which prevent analysis. Worse still, the array references may be non-affine, in which case none of the currently published techniques apply. In short, software methods can be extremely effective when loop nests are analyzable, but there are many factors that can prevent their use in practice.

An analogous situation can occur during auto-parallelization. In such cases the application data set is typically partitioned across P processors, and as a result the working set in each processor may include memory locations separated by integer multiples of P. If the resulting inter-address distances share common factors with C, the working set will not exercise all cache lines with equal probability and conflicts are likely. This places a heavy burden on the parallelization process in the form of additional constraints to be satisfied during data partitioning. Practitioners in this field would have their task simplified if the problem of induced cache conflicts could be removed.

3 The performance of conventional indexing schemes

Before attempting to evaluate the benefits of randomized cache indexing one must first define the systems against which one will compare results. In this section we present miss ratio measurements from the Spec95 benchmark suite for a variety of previously published (and some cases widely used) cache architectures using conventional indexing schemes.

The programs were compiled with the maximum optimization level and instrumented with the ATOM tool [34]. A data cache similar to the first-level cache of the Alpha 21164 microprocessor has been assumed: 8 Kilobytes capacity, 32 bytes per line, write-through and no write allocate. For each benchmark we have simulated the first billion (2^{30}) load operations. Because of the no-write-allocate feature, the performance metrics computed below refer only to load operations.

Table 1 shows the miss ratio for the following cache organizations: direct-mapped, two-way associative, four-way set-associative [32], hash-rehash [3], column-associative [4], victim cache with four victim lines [20], and two-way skewed-associative [30], [31]. Of these schemes, only the two-way skewed-associative cache uses an unconventional indexing scheme, as proposed by its author. For comparison, the miss ratio of a fully-associative cache is shown in the penultimate column. The miss ratio difference between a direct-mapped cache and that of a fully-associative cache is shown in the right-most column of table 1, and represents the direct-mapped conflict miss ratio [17]. In the case of 104.hydro2d and 141.apsi some organizations exhibit lower miss ratios than a fully-associative cache, due to sub-optimality of LRU replacement in a fully-associative cache for these particular programs. Effectively the direct-mapped conflict miss ratio represents the target reduction in miss ratio that we hope to achieve through improved indexing schemes. The other type of misses, compulsory and capacity, will remain unchanged by the use of randomized indexing schemes.

From the results in table 1, we can conclude that set associativity does reduce the miss ratio, as expected, although the improvement of a two-way set-associative cache over a direct-mapped cache is rather low. Comparing the direct-mapped and two-way set-associative cache with the fully-associative cache suggests that several benchmarks (e.g. 101.tomcatv, 102.swim, 125.turb3d, 146.wave) show significant clustering in the mapping of memory lines to cache lines under a conventional mapping scheme.

The hash-rehash cache has a miss ratio similar to that of a direct-mapped cache. Although both have similar access times, the hash-rehash scheme requires two cache probes for some hits. Hence, the direct-mapped cache will be more effective. This poor behavior of the hash-rehash cache was also observed by the originator of that scheme [4], who subsequently proposed a refined scheme, the column associative cache.

The column-associative cache provides a miss ratio similar to that of a two-way set-associative cache. Since the former has a lower access time but requires two cache probes to satisfy some hits,

	direct	2-way	4-way	hash-rehash	column-assoc.	victim	2-way skewed	fully-assoc.	d-m conflicts
101.tomcatv	53.8	48.1	29.5	51.4	47.0	26.6	22.1	12.5	41.3
102.swim	56.2	59.1	57.1	57.6	53.7	33.7	15.1	7.9	48.3
103.su2cor	11.0	9.1	9.0	11.1	9.3	9.5	9.6	8.9	2.1
104.hydro2d	17.6	17.1	17.3	17.6	17.2	17.0	17.1	17.5	0.1
107.mgrid	3.8	3.6	3.5	6.1	4.2	3.7	4.1	3.5	0.3
110.applu	7.6	6.4	6.0	7.8	6.5	6.9	6.7	5.9	1.7
125.turb3d	7.5	6.5	5.3	7.7	6.4	7.0	5.4	2.8	4.7
141.apsi	15.5	13.3	11.3	18.0	13.4	10.7	11.5	12.5	3.0
145.fpppp	8.5	2.7	2.1	5.9	2.7	7.5	2.2	1.7	6.8
146.wave	31.8	31.7	23.0	35.4	30.7	20.1	16.8	13.9	17.9
099.go	13.4	8.2	6.3	11.9	8.6	10.9	7.5	4.8	8.6
124.m88ksim	4.0	1.6	0.7	5.0	2.4	3.2	1.5	0.7	3.3
126.gcc	10.6	7.2	6.0	9.5	7.3	8.6	6.6	5.3	5.3
129.compress	17.1	15.8	15.5	17.6	16.3	16.2	14.3	13.0	4.1
130.li	8.6	5.4	4.3	7.9	5.5	7.2	4.9	3.8	4.8
132.ijpeg	4.1	3.3	3.1	5.2	3.1	2.3	1.9	1.2	2.9
134.perl	10.7	7.3	6.1	10.4	7.5	9.3	6.9	5.2	5.5
147.vortex	5.3	2.7	1.5	7.3	2.7	3.8	1.8	1.4	3.9
Average fp	**21.32**	**19.76**	**16.42**	**21.87**	**19.11**	**14.27**	**11.05**	**8.71**	**12.61**
Average int	**9.22**	**6.44**	**5.44**	**9.34**	**6.67**	**7.67**	**5.66**	**4.42**	**4.80**
Average	**15.95**	**13.84**	**11.54**	**16.30**	**13.58**	**11.34**	**8.66**	**6.80**	**9.14**

Table 1: Miss ratios for the original schemes, with a fully-associative cache for comparison.

any choice between these two organizations should take into account implementation parameters such as access time and miss penalty. The victim cache removes many conflict misses and outperforms a four-way set-associative cache. Finally, the two-way skewed-associative cache offers the lowest miss ratio, which is significantly lower than that of a four-way set-associative cache. The results for the skewed-associative cache are more positive than those observed in [30], where a miss ratio similar to a four-way associative cache was claimed, although using a different workload.

4 Alternative indexing functions

The aim of this paper is to show how alternative cache organizations can exhibit some degree of conflict resistance. The task of finding an appropriate block placement mechanism is somewhat analogous to the problem of finding a suitable hash function for a hash table. For large secondary or tertiary caches it may be possible to use the virtual address mapping to adjust the location of pages in cache, as suggested by Bershad *et al.*, thus avoiding conflicts dynamically, For small first-level caches this effect can only be achieved by using an alternative indexing function.

Perhaps the most well-known alternative indexing scheme is the class of bitwise XOR functions proposed for the skewed associative cache [30]. This yields two or more indexing functions derived by XORing two m -bit fields from an address to produce an m -bit cache index.

In the field of interleaved memories it is well known that bank conflicts can be reduced by using bank selection functions other than the simple modulo-power-of-two. Lawrie and Vora proposed a scheme using prime-modulus functions [23], Harper and Jump [16], and Sohi [33] proposed skewing functions. The use of XOR functions in parallel memory systems was proposed by Frailong *et al.* [8], and other pseudo-random functions were proposed by Raghavan & Hayes [26] and Rau *et al.* [27], [28]. These schemes each yield a more or less uniform distribution of requests to banks, with varying degrees of theoretical predictability and implementation cost. In principle each of these schemes could be used to construct a conflict-resistant cache by using them as the indexing function. However, when considering conflict resistance in cache architectures two factors are critical. Firstly, the chosen indexing function must have a logically simple implementation, and secondly we would like to be able to guarantee good behavior on all regular address patterns - even those that are pathological under a conventional placement function.

In the commercial domain the IBM 3033 [18] and the Amdahl 470 [1] made use of XOR-mapping functions in order to index the TLB. The first generation HP Precision Architecture processors [10] also used a similar technique. In that system, the 11-bit TLB index was obtained by the exclusive OR of two 9-bit fields, one from the virtual page number and the other from the space ID, appended to two other bits from the space ID.

The use of XOR-mapping schemes in order to obtain a pseudo-random placement has been suggested by other authors. For example, Smith [32] compared a pseudo-random placement against a set-associative placement. He concluded that random indexing had a small advantage in most cases, but that the advantages were not significant. In this paper we show that for certain workloads and cache organizations, the advantage can be very large.

Hashing the process ID with the address bits in order to index the cache was evaluated in a multiprogrammed environment by Agarwal in [2]. Results were provided for just one trace, which showed that this scheme could reduce the miss ratio.

The use of multiple, distinct XOR-mapping functions was proposed by Seznec in the skewed-associative cache [30] [31]. A two-way skewed-associative cache consists of two banks of the same size that are accessed simultaneously with two different hashing functions. Not only does the associativity help to reduce conflicts but the skewed indexing functions help to prevent repetitive conflicts from occurring.

In this paper we consider two types of XOR-based indexing functions; those chosen in an *ad hoc* way based on common intuitive notions of how such schemes behave, and a systematic scheme proposed by Rau [28] which describes a method for constructing permutation functions based on polynomial arithmetic. The former class of functions computes a cache index by performing a bitwise XOR of two fields of an address. We refer to this type of schemes as *bitwise XOR mapping,* and describe it section 4.1. The family of mapping functions proposed in [30] belong to this category. We refer to Rau's scheme simply as *polynomial indexing*, and describe how it works in section 4.2.

4.1 Bitwise XOR indexing

Following the notation introduced in [30], a family of XOR indexing functions is defined as follows. Assume that the cache memory consists of 2^l lines of 2^b bytes. A memory address $A = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ comprising the fields $A = (A_3, A_2, A_1, A_0)$ such that $A_0 = \langle a_{b-1}, \ldots, a_0 \rangle$, $A_1 = \langle a_{l+b-2}, \ldots, a_b \rangle$, $A_2 = \langle a_{2l+b-3}, \ldots, a_{b+b-1} \rangle$ and $A_3 = \langle a_{n-1}, \ldots, a_{2l+b-2} \rangle$. Let $\oplus$ denote the bitwise exclusive OR, and let • denote the bitwise AND operation. Let T be any $l-1$ bit number (a good choice for T would be 1010...10), and let $\bar{T} = 2^l - 1 - T$. The family of twin XOR-based indexing functions can then be defined as:

$$\begin{aligned} f_0^T &: \{0 \ldots 2^n - 1\} \to \{0 \ldots 2^{l-1} - 1\} \\ &A = (A_3, A_2, A_1, A_0) \to ((A_2 \bullet T) \oplus A_1, A_0) \\ f_1^T &: \{0 \ldots 2^n - 1\} \to \{0 \ldots 2^{l-1} - 1\} \\ &A = (A_3, A_2, A_1, A_0) \to \left(\left(A_2 \bullet \bar{T}\right) \oplus A_1, A_0\right) \end{aligned} \tag{iv}$$

XOR-based index functions behave in a way similar to a fully-associative cache, but with some restrictions. For instance, in the two-way skewed-associative cache, the set of all addresses that are mapped into the same line of bank 0 are distributed over all the lines in bank 1. Thus, it is similar to having all the lines of bank 1 as alternative locations for a given line in bank 0. However, each particular memory address can be placed in exactly two cache locations (one in bank 0, and the other in bank 1).

4.2 Polynomial indexing

A polynomial mapping function is best described by first considering an unsigned integer address A in terms of its binary representation, as shown for example in equation (iv)

$$A = a_{n-1} 2^{n-1} + a_{n-2} 2^{n-2} + \ldots + a_0 \tag{v}$$

The binary representations of A can be interpreted as a polynomial defined over the field GF(2), thus:

$$A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_0 \qquad \text{(vi)}$$

The binary representation of the cache index R is defined by the GF(2) polynomial $R(x)$ of order less than l such that $A(x) = V(x)P(x) + R(x)$. Effectively $R(x)$ is $A(x)$ modulo $P(x)$, where $P(x)$ is an irreducible polynomial of order l and $P(x)$ is such that $x^i \; mod \; P(x)$ generates all polynomials of order lower than l. The polynomials that fulfil the previous requirements are called *I-Poly* polynomials. Rau shows how the computation of $R(x)$ can be accomplished by the vector-matrix product of the address and an $n \times l$ matrix H of single-bit coefficients [28] derived from $P(x)$. In GF(2), this product is computed by a network of AND and XOR gates, and if the H-matrix is constant the AND gates can be omitted and the mapping then requires just l XOR gates with fan-in from 2 to n.

In practice we may wish to reduce the number of input address bits to the polynomial mapping function by ignoring some of the upper bits in A. This does not seriously degrade the quality of the mapping function and permits us to compute:

$$\left(a_{v-1}x^{v-1} + \ldots + a_1x^1 + a_0x^0\right) \bmod P(x) \qquad \text{(vii)}$$

for some $v < n$.

4.2.1 Characteristics of I-poly functions

I-poly polynomial functions have pseudo-random permutation properties similar to those of prime integer modulus functions. But whereas a prime integer modulus function would be prohibitively complex to implement, the I-poly polynomial function has very low complexity; suitable even for computing a cache index.

These mapping functions have been studied previously in the context of stride-insensitive interleaved memories (see [27] and [28]), and have certain provable characteristics which are of significant value in the context of cache indices. For example, all strides of the form 2^k produce address sequences that are free from conflicts. This is a fundamental result for polynomial indexing; if the addresses of a 2^k-strided sequence are partitioned into M-long sub-sequences, where M is the number of cache blocks, we can guarantee that there are no cache conflicts within each sub-sequence. Any conflicts between sub-sequences are due to capacity problems and can only be solved by larger caches or tiling of the iteration space.

The stride-insensitivity of the I-Poly index function can be seen in figure 2 which shows the behavior of four cache configurations, identical except in their indexing functions. All have 8KB capacity, 32 byte block size, and two-way associativity. They were each driven from an address trace representing repeated accesses to a vector of 64 8-byte elements in which the elements were

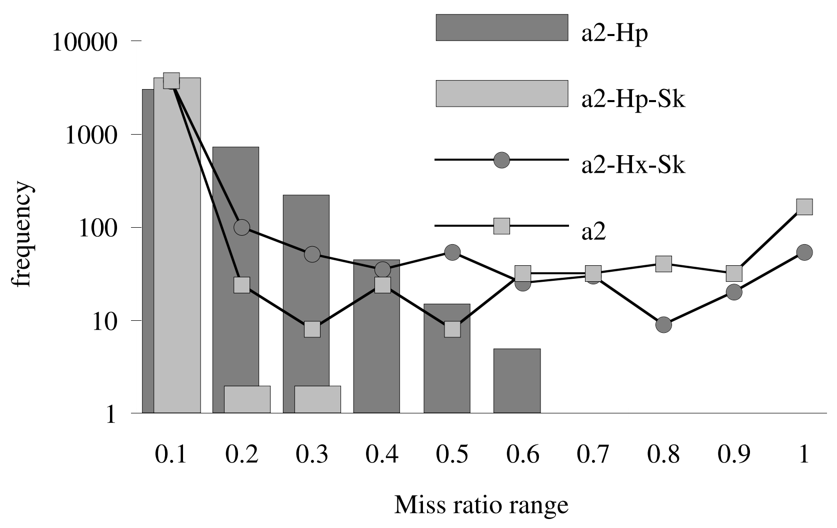

Figure 2. Frequency distribution of miss ratios for conventional and pseudo-random indexing schemes. Columns represent I-Poly indexing and lines represent conventional and skewed-associative indexing.

separated by stride S. With no conflicts such a sequence would use at most half of the 128 sets in the cache. The experiment was repeated for all strides in the range $1 \le S < 4096$ to determine how many strides exhibited bad behavior for each indexing function. The experiment compares three different indexing schemes; conventional modulo power-of-2 (labelled a2), the XOR function proposed in [30] for the skewed-associative cache (a2-Hx-Sk) and two I-Poly functions. The I-Poly scheme was simulated both with and without skewed index functions (a2-Hp and a2-Hp-Sk respectively).

For all schemes the majority of strides yield low miss ratios. However, both the conventional and the skewed XOR functions display pathological behavior (miss ratio > 50%) on more than 6% of all strides. The I-Poly scheme with skewing does not exhibit significant cache conflicts for any of the strides in the range 1 to 4096, suggesting a higher degree of conflict resistance than one can obtain through conventional set-associativity or other (non polynomial) XOR-based index functions.

5 Candidate cache architectures

5.1 Direct-mapped

The direct-mapped cache is the least conflict-resistant of all possible cache architectures and as such serves as a useful baseline for determining an upper bound on cache conflicts in each application. It is also simple to implement and has the fastest hit time of all candidate architectures considered in this paper. The conflict-resistance of a direct-mapped cache can be enhanced by replacing the conventional indexing function with either a bitwise XOR or an I-poly index function. However, this does not increase the perceived associativity, so in general we can expect direct-mapped caches to retain higher miss ratios than some of the schemes presented below which do exhibit associativity.

A bitwise XOR mapping function for a direct-mapped cache can be defined using the notation presented in section 4.1 thus:

$$\begin{aligned} f \,:\, \{0\ldots 2^n - 1\} &\rightarrow \{0\ldots 2^l - 1\} \\ A = (A_3, A_2, A_1, A_0) &\rightarrow (A_2 \oplus A_1, A_0) \end{aligned} \qquad \text{(viii)}$$

where A_0 is b-bit long and A_1 and A_2 are l-bit long.

Similarly, I-poly indexing can be implemented in a direct-mapped cache by constructing an I-poly permutation based on an order-l irreducible polynomial $P(x)$. An example of this is explained in section 6.1.

5.2 Hash-rehash and column-associative

The hash-rehash and column-associative caches are interesting from the viewpoint of conflict avoidance. They have access-time characteristics similar to a direct-mapped cache but also a degree of pseudo-associativity - each address can map to one of two locations in the cache, but initially only one is probed.

The bitwise XOR functions proposed for the skewed-associative cache [30] can also be used for a hash-rehash cache and a column associative cache. For these, as for the skewed-associative cache, we define two distinct mapping functions f_0 and f_1. The first probe uses f_0 and, if required, the second probe uses f_1. These functions are as defined in section 4.1, with a binary value of $T = 10101010$.

In [14] the average cache miss ratios are presented for a number of cache organizations, including the hash-rehash and column associative caches using bitwise XOR functions. On average the XOR-mapping functions worked well for the column-associative cache, provided some slight modifications to the rules for swapping data between the columns are made, and when a pseudo-LRU replacement policy is used.

5.3 Victim cache

The victim cache uses a small fully-associative buffer to cache the lines recently evicted from the first-level direct-mapped cache. One could envisage using a randomized indexing function with the direct-mapped portion of the victim cache organization. Again, this was evaluated in [14], where it was reported that the bitwise XOR function makes the average miss ratio of the victim cache very close to that of the two-way skewed-associative cache. The XOR mapping produced a slight increase in miss ratio for those benchmarks with very few conflict misses. The same behavior was observed for a direct-mapped cache and can be explained by the random, but infrequent, introduction of new conflict misses. Overall, the victim cache organization was unable to eliminate conflicts in programs with pathological conflict behavior, and is consequently not considered further in this paper.

5.4 Two-way set-associative cache

It is to be expected that a two-way set-associative cache will be capable of eliminating many random conflicts. However, a conventionally-indexed set-associative cache will not be able to eliminate pathological conflict behavior as it has limited associativity and a naive indexing function.

The performance of a two-way set-associative cache can be improved by simply replacing the index function, whilst retaining all other characteristics. A suitable bitwise XOR replacement function would be g:

$$\begin{aligned} g : \{0 \ldots 2^n - 1\} &\rightarrow \{0 \ldots 2^{l-1} - 1\} \\ A = (A_3, A_2, A_1, A_0) &\rightarrow (A_2 \oplus A_1, A_0) \end{aligned} \tag{ix}$$

where A_0 contains b bits and A_1 and A_2 contain $(l-1)$ bits.

The same mapping function is used to access both banks, as with a conventional set-associative cache. Conventional LRU replacement is used, as the indexing function has no impact on replacement for this cache organization.

Applying a bitwise XOR mapping scheme to a two-way set-associative cache reduces the average miss ratio in the SPECfp95 benchmarks [14] by more than half, eliminating the majority of the conflict misses. The average miss ratio is just 1.1 times that of a fully-associative cache, whereas the miss ratio of a conventionally-indexed two-way set-associative cache is 2.3 times that of a fully-associative cache. For two programs the two-way XOR cache has a lower miss ratio than a fully-associative cache. This is again due to the sub-optimality of LRU replacement in the fully-associative cache, and is a common anomaly in programs with negligible conflict misses.

An I-poly indexing scheme can also be used in a two-way set-associative cache. As with the direct-mapped cache a single polynomial $P(x)$ of order $l-1$ must be chosen to construct the I-poly function. The resulting miss ratio is slightly better than that observed with a bitwise XOR function.

5.5 Two-way skewed-associative cache

The final candidate cache organization is the two-way skewed-associative cache proposed originally by Seznec [30]. In its original form it used the two bitwise XOR indexing functions f_0 and f_1 defined in section 4.1. It is also possible to use I-poly indexing functions with a skewed-associative cache, as proposed in [14] and [36]. In this case two distinct order $l-1$ polynomials, $P_0(x)$ and $P_1(x)$ must be chosen and used to construct two distinct indexing functions.

6 Miss ratio analysis

In this section we present simulation results for the various candidate cache organizations. Our aim is three-fold. Firstly, to determine the most effective randomization function, secondly to assess the degree to which the candidate organizations are able to eliminate conflict misses in a typical suite of scientific benchmarks, and thirdly to look specifically at the ability of the most promising cache organizations to eliminate induced conflicts in tiled loop kernels.

6.1 Comparison of randomization functions

The performance of bitwise XOR and I-poly mapping has been evaluated for the column associative, the two-way set-associative and the two-way skewed-associative organizations through simulation of the SPECfp95 benchmarks. This suite contains three programs which exhibit very high levels of conflict miss ratios when cache capacities are below 32 Kbytes, as well as other programs with almost no conflict misses.

Table 2 compares the total miss ratios resulting from the mapping function based on the bitwise XOR of two bit strings (*XOR*) with that obtained using polynomial mapping functions (*Poly*). In all cases, an LRU replacement is assumed. The miss ratio of a fully-associative cache is also shown for comparison.

To perform a fair comparison we applied the randomization schemes using the same number of bits of the original address as input to all the mapping functions; in all the cases this is 19 bits (14 without considering the bits that indicate the displacement inside the cache line). For the I-poly mapping functions, we chose the irreducible polynomials that require the fewest number of XOR entries for its implementation. We refer to a polynomial by the value obtained after substituting *x* by 2 (e.g., polynomial 19 is x^4+x+1). The four chosen polynomials are: P_1=505, P_2=301, P_3=131, P_4=137. For the column-associative cache, P_1 and P_2 define the mapping of the two indexing functions used by this organization. P_3 corresponds to the single function utilized by the two-way set-associative cache. Finally, P_3 and P_4 define the two different mapping functions used by two-way skewed-associative cache. Each mapping function requires 7 or 8 XOR gates with fan-in from 2 to 5 each.

miss ratio	column-associative		2-way set associative		2-way skewed assoc.		fully-assoc.
	XOR	Poly	XOR	Poly	XOR	Poly	
101.tomcatv	13.8	12.8	17.0	14.8	20.0	12.6	12.5
102.swim	8.3	7.7	7.9	7.9	12.3	7.5	7.9
103.su2cor	9.1	9.1	9.6	9.9	9.1	9.4	8.9
104.hydro2d	17.1	17.2	17.2	17.1	17.1	17.1	17.5
107.mgrid	4.0	4.2	3.7	3.8	3.9	4.1	3.5
110.applu	6.6	6.5	6.9	6.9	6.3	6.4	5.9
125.turb3d	5.5	6.0	4.6	4.8	4.9	4.2	2.8
141.apsi	10.6	11.2	11.4	11.4	10.5	10.6	12.5
145.fpppp	4.0	2.7	2.7	2.8	2.2	2.3	1.7
146.wave	14.7	13.8	14.4	14.2	16.3	13.7	13.9
Average	9.36	9.12	9.54	9.37	10.24	8.78	8.71

Table 2: Miss ratios for a column associative cache, a two-way associative cache and a two-way skewed-associative cache for the two XOR-mapping schemes: bitwise XOR (XOR) and polynomial mapping (Poly).

We can conclude from table 2 that the I-poly mapping provides a noticeable advantage over the bitwise XOR scheme in cases where conflict miss ratios are high. For the other programs, the reduction in miss ratio achieved by any scheme is very small, since the miss ratio of the original mapping was already very close to that of a fully-associative cache. Overall, the skewed-associative cache using polynomial mapping and a pure LRU replacement achieves a miss ratio practically identical to that of a fully-associative cache (it is just 0.8% higher). Given the advantage of an I-poly function over a bitwise XOR function, all subsequent simulations presented in this paper use the I-poly indexing scheme.

6.2 Conflict avoidance of I-poly indexing functions

Having demonstrated in the previous section that I-poly indexing functions yield the highest degree of conflict avoidance of the two schemes considered, our subsequent experiments are confined to organizations using I-poly indexing functions.

The performance of both the integer and floating-point SPEC95 programs has been evaluated for direct-mapped, column-associative, two-way set-associative and two-way skewed-associative organizations using I-poly indexing functions. In all cases a single-level cache is assumed. The miss ratios of these configurations are shown in table 8, where for comparison the miss ratios of conventionally-indexed fully-associative and direct-mapped caches are also shown.

	I-Poly indexed caches					conventionally-indexed caches		
	column-associative		2-way assoc.	2-way skewed-assoc.		fully- assoc.		direct-mapped
	Swap+ pseudo-LRU	No Swap + LRU		pseudo-LRU	LRU	LRU	Rand.	
101.tomcatv	17.2	12.8	14.8	13.1	12.6	12.5	12.7	53.8
102.swim	7.7	7.7	7.9	7.8	7.5	7.9	7.8	56.2
103.su2cor	10.5	9.1	9.9	9.4	9.4	8.9	9.6	11.0
104.hydro2d	17.6	17.2	17.1	17.0	17.1	17.5	17.5	17.6
107.mgrid	5.1	4.2	3.8	4.5	4.1	3.5	5.0	3.8
110.applu	7.3	6.5	6.9	6.8	6.4	5.9	7.0	7.6
125.turb3d	8.1	6.0	4.8	4.5	4.2	2.8	4.9	7.5
141.apsi	12.2	11.2	11.4	11.0	10.6	12.5	11.2	15.5
145.fpppp	4.0	2.7	2.8	2.1	2.3	1.7	2.7	8.5
146.wave	14.6	13.8	14.2	13.9	13.7	13.9	13.7	31.8
099.go	9.6	6.6	8.6	7.5	6.7	4.8	7.0	13.4
124.m88ksim	2.6	1.2	1.9	1.0	0.8	0.7	0.8	4.0
126.gcc	8.2	6.3	7.2	6.7	6.1	5.3	6.7	10.6
129.compress	14.5	13.5	13.7	13.9	13.4	13.0	14.5	17.1
130.li	5.5	4.5	6.1	4.9	4.5	3.8	4.9	8.6
132.ijpeg	1.8	1.3	1.7	1.5	1.4	1.2	1.5	4.1
134.perl	8.5	6.7	8.8	7.1	6.4	5.2	6.9	10.7
147.vortex	2.7	1.7	2.0	1.8	1.6	1.4	1.9	5.3
Average fp	**10.43**	**9.12**	**9.37**	**9.01**	**8.78**	**8.71**	**9.22**	**21.32**
Average int	**6.68**	**5.22**	**6.26**	**5.55**	**5.09**	**4.42**	**5.52**	**9.22**
Average	**8.77**	**7.39**	**7.99**	**7.47**	**7.14**	**6.80**	**7.58**	**15.95**

Table 3: Miss ratios of I-Poly indexing on SPEC95 benchmarks for a selection of L1 configurations. For the column-associative organization, the first column correspond to the case when swapping is implemented and a 1-bit pseudo-LRU replacement is used, whereas the second column assumes no swapping and exact LRU. For the skewed-associative organization, the first column corresponds to 1-bit pseudo-LRU whereas the second assumes exact LRU. For comparison the best and worst case conventionally-indexed caches are shown in the three right-most columns.

The best result is obtained with a 2-way skewed-associative cache with I-poly index functions. In this case the conflict miss ratio is reduced from 4.8% to 0.67% for SPECint, and from 12.61% to 0.07% for SPECfp. The I-Poly cache eliminates more conflicts than a fully-associative cache with random replacement. This holds systematically for all benchmarks.

For the FP benchmarks, all organizations using I-Poly indexing eliminate the majority of conflict misses, yielding miss ratios within a few percent of a fully-associative cache. Notice also that the performance improvement is dominated by three programs (101.tomcatv, 102.swim and 146.wave). The remaining seven do not contain a significant number of conflict misses, and as a consequence provide little scope for improvement. The conflict miss ratios for direct-mapped, conventional set-associative, and skewed-associative I-Poly caches for those three "bad" programs are shown in table 4.

	conventional indexing			Seznec's 2-way skewed	I-Poly 2-way skewed
	direct	2-way	4-way		
101.tomcatv	41.3	35.6	17.0	9.6	0.1
102.swim	48.3	51.2	49.2	7.2	-0.4
146.wave	17.9	17.8	9.1	2.9	-0.2
Average	35.8	34.9	25.1	6.57	-0.17

Table 4: Conflict miss ratios for selected "bad" programs illustrating the ability of I-Poly indexing to eliminate conflict misses. Negative values in the I-Poly column indicate a slightly advantageous replacement policy in the 2-way I-Poly cache than the fully-associative cache.

These three "bad" programs exhibit pathological conflict miss ratios under conventional indexing schemes. Studies by Olukotun *et al.* have shown that the data cache miss ratio in tomcatv wastes 56% and 40% of available IPC in 6-way and 2-way superscalar processors respectively [24]. Whereas Seznec's 2-way skewed-associative scheme yields a noticable reduction in miss ratio, a 2-way I-Poly cache with distinct index functions appears to eliminate all conflicts.

Table 5 presents miss ratios at both L1 and L2 under conventional and I-Poly indexing for tomcatv, using a single (i.e. non-skewed) I-Poly index function at L1 and a conventional 2-way set-associative cache at L2, for a selection of L1 capacities. Both levels of cache use copy-back and write-allocate policies. Conventional cache miss ratios are similar to those in column 3 of table 1, and I-Poly miss ratios are similar to those in column 5 of table 2. Differences are due to the slightly different cache configuration in this experiment.

L1 capacity (KBytes)	I-Poly L1		Conventional L1	
	L1 miss	L2 miss	L1 miss	L2 miss
1	21.34	45.98	50.78	19.32
2	17.27	56.83	50.67	19.36
4	16.71	58.71	50.58	19.40
8	15.53	63.18	50.55	19.41
16	15.32	64.05	35.27	27.82
32	13.41	73.15	13.56	72.36
64	11.46	85.65	11.41	86.00

Table 5: Local miss ratios for **tomcatv** under conventional 2-way set-associative and 2-way I-Poly indexing for various level-1 cache capacities. Level-2 cache capacity fixed at 1 MByte.and conventional indexing. Global L2 miss ratio was constant at 9.8%

The use of I-Poly indexing virtually eliminates conflict misses, producing a clear advantage over the normal mode of indexing for L1 capacities of 1 - 16 KB. The average speed-up of individual memory references due to I-Poly placement is 1.5 for an 8 KB L1 cache assuming 1-cycle L1 hit time, 6-cycle L2 hit time and 20-cycle L2 miss time. At cache sizes above 32 KB a negligible difference in miss ratio is produced by I-Poly indexing.

6.3 Conflict reduction in tiled loops

We saw in section 2 how tiling can introduce extra cache conflicts, and explained that eliminating these conflicts through software is not always possible or desirable. Now that we have alternative indexing functions that exhibit conflict avoidance properties we can use these to avoid induced conflicts in tiled loop nests. We evaluated the effectiveness of I-Poly placement on tiled loops by simulating the cache behavior of a variety of tiled loop kernels. Here we present a small sample of results to illustrate the general outcome. Consider figure 3, which shows the miss ratio observed in two tiled matrix multiplication kernels where the original matrices were square and of dimensions 171 and 256 respectively. The tile size was varied from 2x2 up to 16x16 to show the effect of self conflicts occurring in caches that are direct-mapped (a1), 2-way set-associative (a2), fully-associative (fa) and skewed 2-way I-Poly (Hp-Sk). The tiled working set divided by cache capacity measures the fraction of the cache occupied by a single tile. In all cases L1 capacity is 8 KBytes, with 32-byte lines.

For $N = 171$ the miss ratio initially falls for all caches as tile size increases. This is due to increasing spatial locality, up to the point where self conflicts begin to occur in the conventionally-indexed direct-mapped and two-way set-associative caches. The fully-associative cache suffers no self-conflicts and its miss ratio decreases monotonically to less than 1% at 50% loading. The behavior of the skewed 2-way I-Poly cache tracks the fully-associative cache closely. The qualitative difference between the I-Poly cache and a conventional two-way cache is clearly visible.

For $N = 256$ the product array and the multiplicand array are positioned in memory so that cross-conflicts occur in addition to self-conflicts. Hence the direct-mapped and 2-way set associative caches experience little spatial locality. However, the I-Poly cache is able to eliminate cross-conflicts as well as self-conflicts, and it again tracks the fully-associative cache.

Further proof of the ability of I-Poly caches to eliminate self conflicts can be seen by plotting miss ratio alongside the maximal conflict-free tile size, as predicted by the algorithm of Lam, Rothberg and Wolf for computing tile sizes [22]. This is shown in figure 4, where the columns represent tile size measurements and the connected points represent miss ratio measurements. The pale gray columns indicate the theoretical maximum conflict-free square tile size for the given array dimension and cache capacity. The darker gray area behind highlights all array sizes where this value

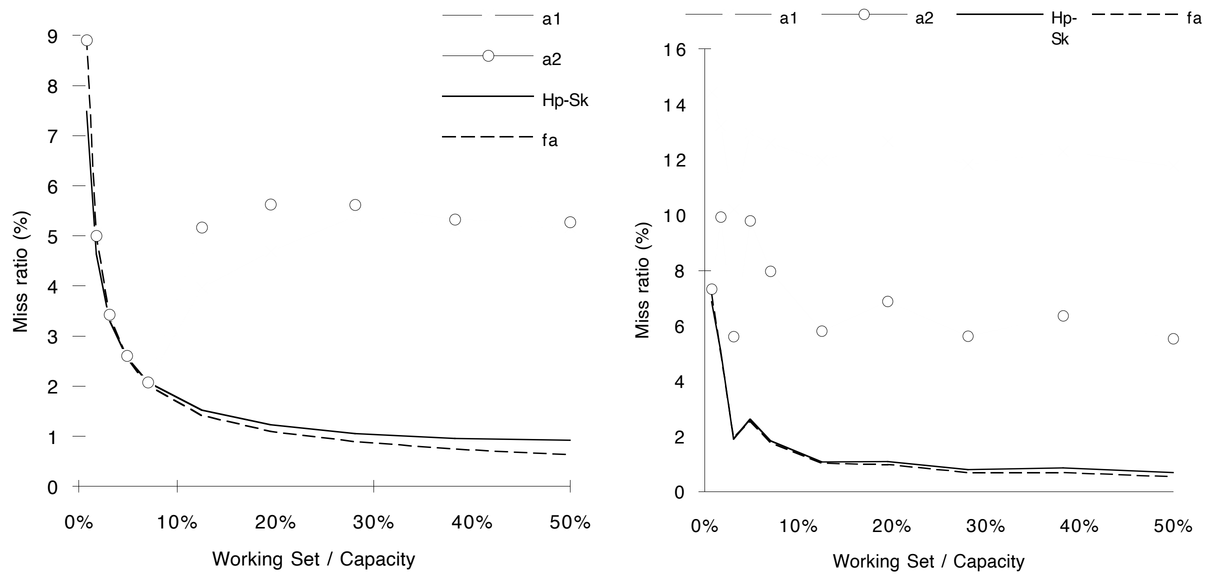

Figure 3. Graphs showing miss ratio versus cache loading (working set / capacity) for tiled matrix multiply kernel. Array dimensions are $N = 171$ (left-most) and $N = 256$ (right-most).

falls below the actual 16x16 tile size used in this experiment. Any array dimensions for which the dark gray is visible should coincide with self-conflict misses in the simulations. We find that the larger the dark gray column for a given array dimension, the larger the miss ratio in the conventional caches. However, for each case measured we find that the two-way polynomial cache eliminates all of the induced conflicts that would otherwise occur.

We see that in some cases a conventional two-way set-associative cache is also able to eliminate conflicts, but that there are other cases where a conventional two-way set-associative cache has a worse miss ratio than a direct-mapped cache. This is a well-known anomaly, and is due to the less than ideal behavior of LRU replacement for some access sequences. For comparison the miss ratio of a fully-associative cache is also shown. We see only small differences between the fully-associative cache and the I-poly two-way skewed-associative cache - and this is true regardless of array dimension. This is a key result; it demonstrates that with an I-Poly cache the optimum tile size is *independent* of array dimension, and can be determined solely by computing the working set of a tiled loop kernel.

7 Implementation Issues

The logic of the GF(2) polynomial modulus operation presented in section 4.2 defines a class of hash functions which compute the cache placement of an address by combining subsets of the address bits

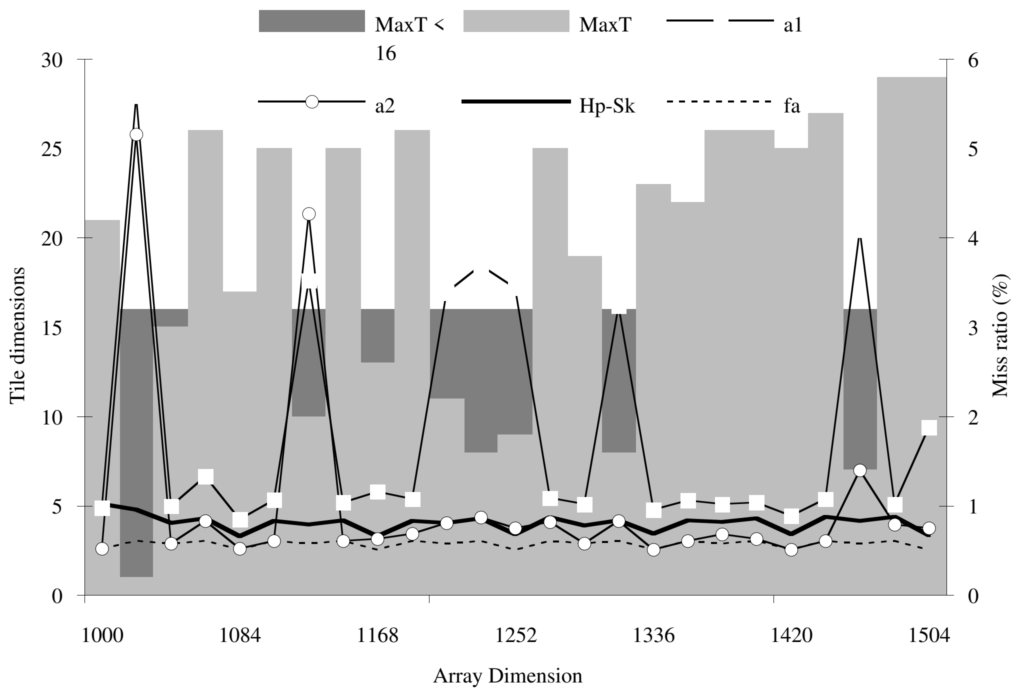

Figure 4. Miss ratios and conflict-free tile dimensions for matrix multiplication as a function of array dimension. All simulations use a 16x16 tile.

using XOR gates. This means that, for example, bit 0 of the cache index may be computed as the exclusive-OR of bits 0, 11, 14, and 19 of the original address. The choice of polynomial determines which bits are included in each set. The implementation of such a function for a cache with an 8-bit index would require just eight XOR gates with fan-in of 3 or 4.

Whilst this appears remarkably simple, there is more to consider than just the placement function. Firstly, the function itself uses address bits beyond the normal limit imposed by typical minimum page size restriction. Secondly, the use of pseudo-random placement in a multi-level memory hierarchy has implications for the maintenance of Inclusion. Here we briefly examine these two issues and show how the virtual-real two-level cache hierarchy proposed by Wang *et al.* [37] provides a clean solution to both problems. Finally, the impact of XOR gates on the critical path of address computation is analyzed, and a scheme based on address prediction is proposed to overcome the penalties caused by extensions to the critical path.

7.1 Overcoming page size restrictions

Typical operating systems permit pages to be as small as 4 Kbytes. In a conventional cache this places a limit on the first-level cache size if address translation is to proceed in parallel with tag lookup. Similarly, any novel cache indexing scheme which uses address bits beyond the minimum page size boundary cannot use a virtually-indexed physically-tagged cache. From the alternative options available one might consider:

1. Performing address translation prior to tag lookup (i.e. use physical indices)
2. Enabling I-Poly indexing only when data pages are known to be large enough
3. Using a virtually-indexed virtually-tagged level-1 cache
4. Indexing conventionally, but use a polynomial rehash on a level-1 miss.

Option 1 is attractive if an existing processor pipeline performs address translation at least one stage prior to tag lookup. This might be the case in a processor which is able to hide memory latency through dynamic execution or multi-threading, for example. However, in many systems, performing address translation prior to tag lookup will either extend the critical path through a critical pipeline stage or introduce an extra cycle of untolerated latency via an additional pipeline stage.

Option 2 could be attractive in high performance systems where large data sets and large physical memories are the norm. In such circumstances processes may typically have data pages of 256 Kbytes or more. The O/S would need to track the page sizes of segments currently in use by a process (and its kernel) and enable polynomial cache indexing at the first-level cache if all segments' page sizes were above a certain threshold. This would provide more unmapped bits to the hash function when possible, but revert to conventional indexing when this is not possible. For example, if the threshold was 256 Kbytes and the cache was 8 Kbytes two-way associative, one could implement a polynomial function combining 13 unmapped physical address bits to produce 7 cache index bits. This would be sufficient to produce good conflict-free behavior. Provided the level-1 cache is flushed when the indexing function is changed, there is no reason why the indexing function needs to remain constant.

The third option is not currently popular, primarily because of potential difficulties with aliases in the virtual address space as well as the difficulty of shooting down a level-1 virtual cache line when a physically-addressed invalidation operation is received from another processor. However, the two-level virtual-real cache hierarchy proposed by Wang *et al.* in [37] provides an interesting way of implementing a virtually-tagged L1 cache, thus exposing more address bits to the indexing function without incurring address translation delays. We consider this to be the most promising option for implementing an I-poly cache; it enables more address bits to be used in the index function and also provides a mechanism for maintaining Inclusion in the presence of holes (discussed in section 7.2).

The fourth option would be appropriate for a physically-tagged direct-mapped cache. It is similar in principle to the hash-rehash [3] and the column-associative caches [4]. The idea is to make an initial probe with a conventional integer-modulus indexing function, using only unmapped address bits. If this probe does not hit we probe again, but at a different index. By the time the second probe begins, the full physical address is available and can be used in a polynomial hashing function to compute the index of the second probe.

Addresses which can be co-resident under a conventional index function will not collide on the first probe. Conversely, sets of addresses which do collide under a conventional indexing function collide under the second probe with negligible probability 2^{-m}, due to the pseudo-random distribution of the polynomial hashing function. This provides pseudo-full associativity in what is effectively a direct-mapped cache. The hit time of such a cache on the first probe would be as good as any direct-mapped physically-indexed cache. However, the average hit time is lengthened slightly due the occasional need for a second probe. We have investigated this style of cache and devised a scheme for swapping cache lines between their "conventional" modulo-indexed location and their "alternative" polynomially-indexed location. This leads to a typical probability of around 90% that a hit is detected at the first probe. However, the slight increase in average hit time due to occasional double probes means that a column-associative cache is only attractive when miss penalties are comparatively large.

7.2 Requirements for Inclusion

Coherent cache architectures normally require that the property of Inclusion is maintained between all levels of the memory hierarchy. Thus, if L_k represents the set of data present in cache at level k, the property of Inclusion demands that $L_i \subseteq L_{i+1}$ for $1 \le i < M$ in an M-level memory hierarchy. Whenever this property is maintained a snooping bus protocol need only compare addresses of global write operations with the tags of the lowest level of private cache.

The line at index i_2 in the L2 cache is replaced when a line at index i_1 in the L1 cache is replaced with data at address A if A is not already present in L2. If line i_2 contains valid data we must be sure that after replacement its data is not still present in L1. In a conventionally-indexed cache this is not an issue because it is relatively easy to guarantee that the data at L2 index i_2 is always located at L1 index i_1, thus ensuring that L1 replacement will automatically preserve Inclusion. In a pseudo-randomly indexed cache there is in general no way to make this guarantee. Instead, the cache replacement protocols must explicitly enforce Inclusion by invalidating data at L1 when required.

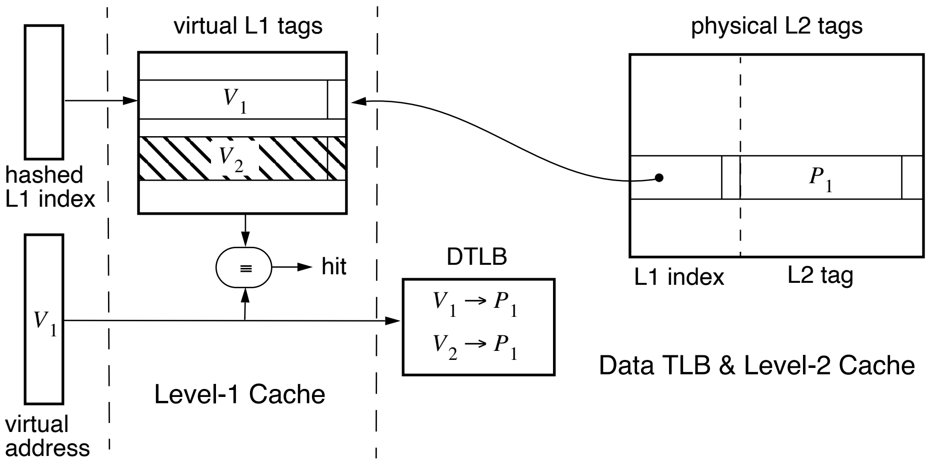

Figure 5. Organization of a two-level I-Poly virtually-indexed cache. The principal feature is the L1 index within each L2 tag. This implements an L1 directory to support inclusion, coherency and anti-aliasing. The L1 index in L2 prevents V_1 and V_2 from co-existing in L1.

7.2.1 A common solution for Coherency and Inclusion

A processor with a physically-tagged second-level cache can have an inclusive, coherent, virtually-tagged level-1 cache provided two rules are obeyed.

1. There can be at most one virtual alias of a given physical address in the L1 cache at a time.
2. The L2 cache must be able to locate any virtual L1 copy of each physical location present in L2.

Both rules can be implemented by appending a valid L1 index to each L2 tag as shown in figure 5. In this way it is possible to invalidate a virtual address in L1 when its alias is loaded from L2 into L1. Similarly, the L1 index stored with the L2 tag can be used to finger a particular L1 line for invalidation when an L2 line is invalidated either by an external coherency action or through the process of L2 replacement. In terms of L2 tag space this would require, for example, nine extra bits per L2 tag assuming an L1 cache with 256 tags.

If the L2 tag contains a valid ‘L1 index’ field, the L1 line at that index is invalidated whenever the L2 tag is over-written or invalidated.

When an L1 line is replaced the L2 cache should be informed that the line previously held in L1 is now no longer there, and that the ‘L1 index’ field should be invalidated for the corresponding L2 tag. This minimises unnecessary L1 invalidation, but is not a correctness requirement.

In a COMA system there will be occasional global update operations. These invalidate lines at L2, which in turn invalidates lines at L1 pointed to by the L1 index field. At no stage is there a requirement for reverse address translation. Inclusion is guaranteed by this two-level virtual-real cache, but leads to the creation of holes at the upper level of the cache, in turn leading to the possibility of additional cache misses.

7.3 Performance implication of holes

In a two-level virtual-real cache hierarchy there are three causes of holes at L1; these are:

1. Replacements at L2
2. Removal of virtual aliases at L1
3. Invalidations due to external coherency actions

It is probable that the frequency of item 2 occurring will be low; for this kind of hole to cause a performance problem a process must issue interleaved accesses to two segments at distinct virtual addresses which map to the same physical address. We preserve a consistent copy of the data at these virtual addresses by ensuring that at most one such alias may be present in L1 at any instant. This does not prevent the physical copy from residing undisturbed at L2; it simply increases the traffic between L1 and L2 when accesses to virtual aliases are interleaved.

Invalidations from external coherency actions occur regardless of the cache architecture so we do not consider them further in this analysis. The events that are of primary importance are invalidations at L1 due to the maintenance of Inclusion between L1 and L2. It is important to quantify their frequency and the effect they have on hit ratio at L1.

Recall that the index function at L2 is based on a physical address whereas the index function at L1 uses a virtual address. Also, the number of bits included in the index function and the function itself will be different in both cases. As these functions are pseudo-random there will be no correlation between the indices at L1 and L2 for each particular datum. For example, assuming direct-mapped caches, when a line is replaced at L2 the data being replaced will also exist in L1 with probability P_r

$$P_r = \frac{2^{m_1}}{2^{m_2}} = 2^{(m_1 - m_2)} \qquad \text{(x)}$$

where m_1 and m_2 are the number of bits in the indices at L1 and L2 respectively.

If the data being replaced at L2 does exist in L1, it is possible that the L1 index is coincidentally equal to the index of the data being brought into L1 (as the L2 replacement is actually caused by an L1 replacement). If this occurs a hole will not be created after all. Thus the probability that the elimination of a line at L1 to preserve inclusion will result in a hole is given by P_d

$$P_d = \frac{2^{m_1} - 1}{2^{m_1}} \qquad \text{(xi)}$$

The net probability that a miss at L2 will cause a hole to appear at L1 is P_H, given by the product of P_d and P_r, thus:

$$P_H = \frac{2^{m_1} - 1}{2^{m_2}} \qquad \text{(xii)}$$

When the size ratio between L1 and L2 is large the value of P_H is small. For example, an 8KB L1 cache and a 256KB L2 cache with 32 byte lines yield $P_H = 0.031$. Slightly more than 3% of L2 misses will result in the creation of a hole.

The expected increase in compulsory miss ratio at L1 can be modelled by the product of P_H and the L2 miss ratio. When compared with simulated miss ratios we found that this approximation is accurate for L2:L1 cache size ratios of 16 or above. For instance simulations of the whole Spec95 suite with an 8 KB two-way skewed-associative I-Poly L1 cache backed by a 1 Mb conventionally-indexed two-way set-associative L2 cache showed that the effect of holes on L1 miss ratio is negligible. The percentage of L2 misses that created a hole averaged less than 0.1% and was never greater than 1.2% for any program.

The two-level virtual-real cache described in [37] implements a protocol between the L1 and L2 cache which effectively provides a mechanism for ensuring that inclusion is maintained, that coherence can be maintained without reverse address translation, and in our case that holes can be created at level-1 when required by the inclusion property.

The use of pseudo-random index functions means that some holes will be created at L1, but simulations and simple probabilistic models both predict that their impact will be negligible.

7.4 Effect of polynomial mapping on critical path

A cache memory access in a conventional organization normally computes its effective address by adding two registers or a register plus a displacement. I-poly indexing implies additional circuitry to compute the index from the effective address. This circuitry consists of several XOR gates that operate in parallel and therefore the total delay is just the delay of one gate. Each XOR gate has a number of inputs that depend on the particular polynomial being used. For the experiments reported in this paper the number of inputs is never higher than 5. Therefore, the delay due to the XOR gates will be low compared with the delay of a complete pipeline stage.

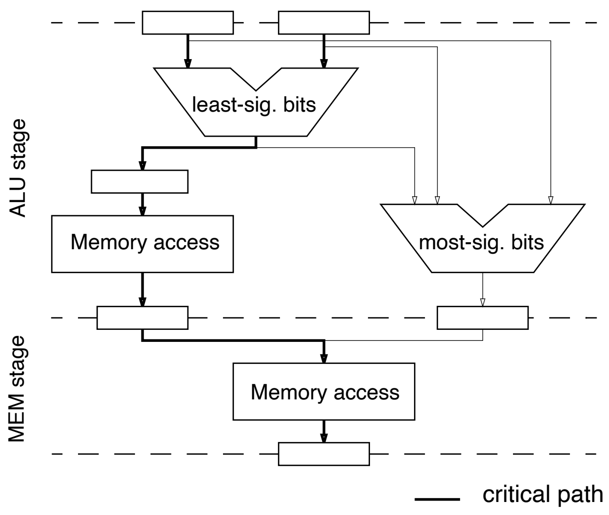

Figure 6. A pipeline that overlaps part of the address computation with the memory access.

Depending on the particular design, it may happen that this additional delay can be hidden. For instance, if the memory access does not begin until the complete effective address has been computed, the XOR delay can be hidden since the address is computed from right to left and the XOR gates use only the least-significant bits of the address (19 in the experiments reported in this paper). Notice that this is true even for carry look-ahead adders (CLA). A CLA with look-ahead blocks of size b bits computes first the b least-significant bits, which are available after a delay of approximately one look-ahead block. After a three-block delay the b^2 least-significant bits are available. In general, the b^i least-significant bits have a delay of approximately $2i - 1$ blocks. For instance, for 64-bit addresses and a binary CLA, the 19 bits required by the I-poly functions used in the experiments of this paper have a delay of about 9 blocks whereas the whole address computation requires 11 block-delays. Once the 19 least-significant bits have been computed, it is reasonable to assume that the XOR gate delay is shorter than the time required to compute the remaining bits.

However, since the cache access time usually determines the pipeline cycle, the fact that the least-significant bits are available early is sometimes exploited by designers in order to shorten the latency of memory instructions by overlapping part of the cache access (which requires only the least-significant bits) with the computation of the most significant address bits. This approach results in a pipeline with a structure similar to that shown in figure 6. Notice that this organization requires a pipelined memory (in the example we have assumed a two-stage pipelined memory). In this case, the polynomial mapping may cause some additional delay to the critical path. We will show later that even if the additional delay induces a one cycle penalty in the cache access time, the polynomial mapping provides a significant overall performance improvement. An additional delay in a load instruction may have a negative impact on the performance of the processor because the issue of dependent instructions may be delayed accordingly. On the other hand, this delay has a negligible effect, if any, on store instructions since these instructions are issued to memory when they are committed in order to have precise exceptions, and therefore the XOR functions can usually be performed while the instruction is waiting in the store buffer. Besides, only load instructions may depend on stores but these dependencies are resolved in current microprocessors (e.g. PA8000 [19]) by forwarding. This technique compares the effective address of load and store instructions in order to check a possible match but the cache index, which involves the use of the XOR gates, is not required by this operation.

Memory address prediction can be also used to avoid the penalty introduced by the XOR delay when it lengthens the critical path. The effective address of memory references has been shown to be highly predictable. For instance, in [13] it has been shown that the addresses of about 75% of the dynamically executed memory instructions of the Spec95 suite can be predicted with a simple scheme based on a table that keeps track of the last address seen by a given instruction and its last stride. We propose to use a similar scheme to predict early in the pipeline the line that is likely to be accessed by a given load instruction. In particular, the scheme works as follows.

The processor incorporates a table indexed by the instruction address. Each entry stores the last address and the predicted stride for some recently executed load instruction. In the fetch stage, this table is accessed with the program counter. In the decode stage, the predicted address is computed and the XOR functions are performed to compute the predicted cache line. Notice that this can be done in just one cycle since the XOR can be performed in parallel with the computation of the most-significant bits as discussed above, and the time to perform an integer addition is not higher than one cycle in the vast majority of processors. When the instruction is subsequently issued to the memory unit it uses the predicted line number to access the cache in parallel with the actual address and line computation. If the predicted line turns out to be incorrect, the cache access is repeated again with the actual address. Otherwise, the data provided by the speculative access can be loaded into the destination register.

The scheme to predict the effective address early in the pipeline has been previously used for other purposes. In [11], a Load Target Buffer is presented, which predicts effective address adding a stride to the previous address. In [5] and [6] a Fast Address Calculation is performed by computing load addresses early in the pipeline without using history information. In those proposals the memory access is overlapped with the non-speculative effective address calculation in order to reduce the cache access time, though none of them execute speculatively the subsequent instructions that depend on the predicted load.

A number of previous papers have proposed the use of a memory address prediction scheme in order to execute memory instructions speculatively, as well as instructions dependent upon them [12], [13] and [29]. In the case of a miss-speculation, a recovery mechanism similar to that used by branch prediction schemes is utilized to squash the miss-speculated instructions.

8 Effect of I-Poly indexing on superscalar IPC

In order to verify the impact of polynomial mapping on a realistic microprocessor architecture we have developed a parametric simulator of an out-of-order execution processor. A four-way superscalar processor has been simulated. table 7 shows the different functional units and their latency considered for this experiment. The size of the reorder buffer is 32 entries. There are two separate physical register files (FP and Integer), each one having 64 physical registers. The processor has a lockup-free data cache [21] that allows 8 outstanding misses to different cache lines. The cache size is either 8 KB or 16 KB and is 2-way set-associative with 32-byte line size. The cache is write-through and no-write-allocate. The hit time of the cache is two cycles and the miss penalty is 20 cycles. An infinite L2 cache is assumed and a 64-bit data bus between L1 and L2 is considered (i.e., a line transaction occupies the bus during four cycles). There are two memory ports and dependencies thorough memory are speculated using a mechanism similar to the ARB of the Multiscalar [9] and PA8000 [19]. A branch history table with 2K entries and 2-bit saturating counters is used for branch prediction.

Functional Unit	Latency	Repeat rate
1 Simple Integer	1	1
1 Complex Integer	9 multiply 67 divide	1 67
2 Effective Address	1	1
1 Simple FP	4	1
1 FP Multiplication	4	1
1 FP Divide and SQR	16 divide 35 SQR	16 35

Table 6 Functional units and instruction latency.

The memory address prediction scheme has been implemented by means of a direct-mapped table with 1K entries and without tags in order to reduce cost at the expense of more interferences in the table. Each entry contains the last effective address of the last load instruction that used this entry and the last observed stride. In addition, each entry contains a 2-bit saturating counter that assigns confidence to the prediction. Only when the most-significant bit of the counter is set is the prediction considered to be correct. The address field is updated for each new reference regardless of the prediction, whereas the stride field is only updated when the counter goes below 10_2 (i.e. after two consecutive mispredictions).

Table 7 shows the IPC and the miss ratio for different configurations. The baseline configuration

	Conventional indexing					I-poly indexing			
	16kb		8 KB			8KB			
			IPC		miss	Xor no CP		Xor in CP	
								no pred.	with pred.
	IPC	miss	no pred	with pred		IPC	miss	IPC	IPC
go	1.00	5.45	0.87	0.88	10.87	0.87	10.60	0.83	0.84
m88ksim	1.56	1.41	1.53	1.53	2.62	1.52	2.89	1.49	1.51
gcc	1.16	5.63	1.04	1.05	10.01	1.03	10.77	0.98	0.99
compress	1.13	12.96	1.12	1.13	13.63	1.11	14.17	1.07	1.10
li	1.40	4.72	1.30	1.32	8.01	1.33	7.10	1.26	1.31
ijpeg	1.31	0.94	1.28	1.28	3.72	1.29	2.17	1.28	1.30
perl	1.45	4.52	1.26	1.27	9.47	1.24	10.26	1.19	1.21
vortex	1.39	4.97	1.27	1.28	8.37	1.30	7.87	1.25	1.27
tomcatv	1.18	35.14	1.03	1.04	54.45	1.33	19.67	1.30	1.36
swim	1.30	29.56	1.06	1.08	66.62	1.53	8.85	1.49	1.57
su2cor	1.28	13.74	1.24	1.26	14.69	1.24	14.66	1.21	1.25
hydro2d	1.14	15.40	1.13	1.15	17.23	1.13	17.22	1.11	1.15
applu	1.63	5.54	1.61	1.63	6.16	1.57	6.84	1.55	1.59
mgrid	1.51	4.91	1.50	1.53	5.05	1.50	5.31	1.46	1.52
turb3d	1.85	4.67	1.80	1.82	6.05	1.81	5.38	1.78	1.82
apsi	1.13	10.03	1.08	1.09	15.19	1.08	13.36	1.07	1.09
fpppp	2.14	1.09	2.00	2.00	2.66	1.98	2.47	1.93	1.94
wave5	1.37	27.72	1.26	1.28	42.76	1.51	14.67	1.48	1.54
Int average	1.29	5.07	1.19	1.20	8.34	1.20	8.23	1.15	1.17
Fp average	1.42	14.78	1.34	1.35	23.09	1.44	10.84	1.41	1.46
Combined average	1.36	10.47	1.27	1.28	16.53	1.33	9.68	1.29	1.33

Table 7 IPC and load miss ratio for different cache configuration. Miss ratios are averaged with arithmetic mean, and IPC rates are averaged with geometric means.

is an 8 KB cache with conventional indexing and no address prediction (4th column). The average IPC of this configuration is 1.27 and the average miss ratio (6th column) is 16.53[1]. When I-poly indexing is used the average miss ratio goes down to 9.68 (8th column). If the XOR gates are not in the critical path this implies an increase in the IPC up to 1.33 (7th column). On the other hand, if the XOR gates are in the critical path and we assume a one cycle penalty in the cache access time, the

1. For each benchmark we simulated 100M instructions after skipping the first 2000M.

resulting IPC is 1.29 (9th column). However, the use of the memory address prediction scheme when the XOR gates are in the critical path (10th column) provides the same overall performance as a cache with the XOR gates not in the critical path (7th column). Thus, the main conclusion of this study is that the memory address prediction scheme can offset the penalty introduced by the additional delay of the XOR gates when they are in the critical path. Finally, table 7 also shows the performance of a 16 KB 2-way set-associative cache (2nd and 3rd columns). Notice that the addition of I-poly indexing to an 8 KB cache yields over 60% of the IPC increase that can be obtained by doubling the cache size.

The absolute differences are low, but this is because the benefit of I-poly indexing is perceived by a small subset of the benchmark programs. In the Spec95 benchmark suite there are many benchmarks that exhibit a relatively low conflict miss ratio. In fact the Spec95 conflict miss ratio of a 2-way associative cache is less than 4% for all programs except tomcatv, swim and wave5. If we perform independent analyses on the benchmarks with high conflict miss ratios, versus those with low conflict miss ratios, we can observe the ability of polynomial mapping to reduce the miss ratio and significantly boost the performance of the problem cases. This is shown in table 8, which contains the results for the three programs with high conflict miss ratios together with their averages and the averages of the remaining fifteen programs with lower conflict miss ratios. In this breakdown

	Conventional indexing					I-poly indexing			
	16kb		8 KB			8 KB			
						Xor no CP		Xor in CP	
			IPC		miss			no pred.	with pred.
	IPC	miss	no pred.	with pred.		IPC	miss	IPC	IPC
tomcatv	1.18	35.14	1.03	1.04	54.45	1.33	19.67	1.30	1.36
swim	1.30	29.56	1.06	1.06	66.62	1.53	8.85	1.49	1.57
wave5	1.37	27.72	1.26	1.28	42.76	1.51	14.67	1.48	1.54
Average-bad	1.28	30.80	1.11	1.13	54.61	1.46	14.40	1.42	1.49
Average-good	1.38	6.40	1.30	1.32	8.91	1.30	8.74	1.27	1.30

Table 8 IPC and load miss ratio for different cache configurations for the selected bad programs. Miss ratios are averaged using an arithmetic mean, whereas IPC rates are averaged using a geometric mean. Final row shows averages for the 15 programs with low conflict miss ratios.

one can see that the polynomial mapping provides a significant improvement in performance for the bad programs even if the XOR gates are in the critical path and the memory address prediction scheme is not used (27% increase in IPC). When memory address prediction is used the IPC is 33% higher than that of a conventional cache of the same capacity and 16% higher than that of a conventional cache with twice the capacity. Notice that the polynomial mapping scheme with prediction is even better than the organization with the XOR gates not in the critical path but without prediction. This is due to the fact that the memory address prediction scheme reduces by one cycle the effective cache hit time when the predictions are correct, since the address computation is overlapped with the cache access (the computed address is used to verify that the prediction was correct). However, the main benefits observed in table 8 come from the reduction in conflict misses. To isolate the different effects we have also simulated an organization with the memory address prediction scheme and conventional indexing for an 8 KB cache (column 5). If we compare this IPC with that in column 4 of table 3, we see that the benefits of the memory address prediction scheme due to the reduction of the hit time are almost negligible. This confirms that the improvement observed in the I-poly indexing scheme with address prediction derives from the reduction in conflict misses.

The averages for the fifteen programs which exhibit low levels of conflict misses (labelled "average-good") show a small (1.7%) deterioration in average IPC when I-poly indexing is used and the XOR gates are in the critical path. This is due to a slight increase in the average hit time rather than an overall increase in miss ratio (which on average falls by 2%). For these programs the reduction in aggregated miss penalty does not outweigh the slight extension in critical path length.

9 Conclusions

In this paper we have discussed the problem of cache conflict misses and surveyed the options for reducing or eliminating those conflicts. We have described pseudo-random indexing schemes based on bitwise XOR functions and boolean polynomial modulus functions. We have shown the latter to be robust enough to virtually eliminate the repetitive cache conflicts caused by the "bad strides" that are both inherent in some Spec95 benchmarks and also introduced into an application by the tiling of loop nests.

We have discussed the major implementation issues that arise from the use of such novel indexing schemes. For example, I-poly indexing uses more address bits than a conventional cache to compute the cache index. Also, the use of different indexing functions at L1 and L2 results in the occasional creation of a hole at L1. We have shown how both of these problems can be solved using a two-level virtual-real cache hierarchy. Finally, we have proposed a memory address prediction scheme to avoid the penalty due to the potential delay in the critical path introduced by the pseudo-random indexing function.

Detailed simulations of an o-o-o superscalar processor have demonstrated that programs with significant numbers of conflict misses in a conventional 8 KB 2-way set-associative cache perceive IPC improvements of 33% (with address prediction) or 27% (without address prediction). This is up to 16% higher than the IPC improvements obtained simply by doubling the cache capacity. Furthermore, from the programs we analyzed, those that do not experience significant conflict misses on average see only a 1.7% reduction in IPC when I-poly indexing appears on the critical path for computing the effective address, and address prediction is used. If the index function does not appear on the critical path no deterioration in overall performance is experienced by those programs. The small potential reduction in IPC for some programs may appear to detract from the benefit of using I-poly indexing; one could argue that an expert programmer could restructure the application to avoid cache conflicts. This of course assumes the programmer is able to identify and locate the source of conflicts, and also represents a highly machine-specific optimization.

We believe the key contribution of pseudo-random indexing is the resulting predictability of cache behavior. In our experiments we see that I-poly indexing reduces the standard deviation of miss ratios across Spec95 from 18.49 to 5.16. The use of caches in real-time systems is often problematic when it cannot be guaranteed that pathological miss ratios will not occur. If conflict misses are eliminated, the miss ratio depends solely on compulsory and capacity misses, which in general are easier to predict and control. Systems which incorporate a pseudo-random cache could be particularly useful in the real-time domain. Conflict resistance could also be beneficial in cache-based scientific computing where expert programmers and restructuring compilers use iteration-space tiling to manage data locality.

References

[1] Amdahl Corp., *470V/6 Machine Reference Manual*, 1976

[2] A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming,* Kluwer Academic Publishers, 1989, pp. 120-122.

[3] A. Agarwal, J. Hennessy and M. Horowitz, "Cache Performance of Operating Systems and Multiprogramming", *ACM Trans. on Computer Systems,* vol. 6, Nov. 1988, pp. 393-431.

[4] A. Agarwal and S.D. Pudar, "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches", in *Proc. Int. Symp. on Computer Architecture,* 1993, pp. 179-190.

[5] T.M. Austin, D.N.Pnevmastikatos G.S. Sohi, "Streamlining Data Cache Access with Fast Address Calculation", in *Proc of the Int. Symp. on Computer Architecture*, pp. 369-380, 1995.

[6] T.M. Austin, G.S. Sohi, "Zero-Cycle Loads: Microarchitecture Support for Reducing Load Latency", in *Proc. of Int. Symp. on Microarchitecture,* pp 82-92, 1995.

[7] B.N. Bershad, D. Lee, T.H. Romer and J.B. Chen, “Avoiding Cache Conflict Misses Dynamically in Large Direct-Mapped Caches”, in *Proc. ASPLOS VI*, pp.158-170, 1994.

[8] J.M. Frailong, W. Jalby and J. Lenfant, “XOR-Schemes: A Flexible Data Organization in Parallel Memories”, in *Proc. Int. Conf. on Parallel Processing*, pp. 276-283, Aug. 1985.

[9] M. Franklin and G.S. Sohi, “ARB: A Hardware Mechanism for Dynamic Reordering of Memory References”, *IEEE Transactions on Computers*, 45(6), pp. 552-571, May 1996.

[10] D.A. Fotland *et al*., “Hardware Design of the First HP Precision Architecture Computers”, *Hewlet-Packard Journal,* 38(3), March 1987, pp. 4-17.

[11] M.Golden and T.N. Mudge, “Hardware Support for Hiding Cache Latency”, Technical report # CSE-TR-152-93. University of Michigan, 1993.

[12] J. González and A. González, “Memory Address Prediction for Data Speculation”, in *Proc. of EUROPAR 97*, pp. 1084-1091, 1997, also available as technical report # UPC-DAC-1996-50, http://www.ac.upc.es, Oct. 1996.

[13] J. González and A. González, “Speculative Execution via Address Prediction and Data Prefetching”, in *Proc of 11th. ACM Int. Conf. on Supercomputing*, Vienna (Austria), pp. 196-203, 1997, also available as technical report # UPC-DAC-1997-2, http://www.ac.upc.es, Jan. 1997.

[14] A. González, Mateo Valero, Nigel Topham and Joan M. Parcerisa, “Eliminating Cache Conflict Misses Through XOR-based Placement Functions”, in *Proc of 11th. ACM Int. Conf. on Supercomputing*, Vienna (Austria), pp, 76-83, July 1997.

[15] S. Gosh, M. Martonosi and S. Malik, “Cache Miss Equations: An Analytic Representation of Cache Misses”, in *Proc. ICS ‘97*, Vienna, pp, 317-324, July 1997.

[16] D.T. Harper III and J.R. Jump, “Vector Access Performance in Parallel Memories Using a Skewed Storage Scheme”, *IEEE Trans. Comp*., Vol. TC-36, No 12, pp. 1440-1449, Dec. 1987.

[17] J.L. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann Publish., 1996

[18] .IBM, *3033 Processor Complex, Theory of Operation/Diagrams Manual-Processor Storage Control Function*, vol. 4, IBM, Poughkeepsie, N.Y., 1978

[19] D. Hunt, “Advanced Performance Features of the 64-bit PA-8000”, in *Proc. of the CompCon’95,* pp. 123-128, 1995.

[20] N. P. Jouppi, “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers”, in *Proc. Int. Symp. on Computer Architecture,* 1990, pp. 364-373.

[21] D. Kroft, “Lockup-free instruction fetch/prefetch cache organization”, in *Proc. 8th International Symposium on Computer Architecture* (1981) pp. 81-87

[22] M.S. Lam, E.E. Rothberg and M.E. Wolf, “The Cache Performance and Optimization of Blocked Algorithms”, in *Proc. ASPLOS-IV,* April 1991, pp. 63-74 (also SIGPLAN Notices 26).

[23] D.H. Lawrie and C.R. Vora, “The Prime Memory System for Array Access”, *IEEE Trans. Comp*., Vol. TC-31, No. 5, pp. 435-442, May 1982.

[24] K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, “The Case for a Single-Chip Multiprocessor”, in *Proc. ASPLOS-VII*, October 1996.

[25] D. Patterson *et al*., “A Case for Intelligent RAM”, *IEEE Micro*, Mar/Apr. 1997, pp. 34-44.

[26] R. Raghavan and J.P. Hayes, “On Randomly Interleaved Memories”, in *Proc. Supercomputing ‘90*, pp. 49-58.

[27] B.R. Rau, M.S. Schlansker and D.W.L Yen, “The Cydra 5 Stride-Insensitive Memory System”, In *Proc Int. Conf. on Parallel Processing,* 1989, pp. 242-246.

[28] B.R. Rau, “Pseudo-Randomly Interleaved Memories”, in *Proc. Int. Symp. on Computer Architecture,* 1991, pp. 74-83.

[29] Y. Sazeides, S. Vassiliadis and J.E. Smith, “The Performance Potential of Data Dependence Speculation & Collapsing”, in *Proc. of Int. Symp. on Microarchitecture,* pp. *238-257* December 1996.

[30] A. Seznec, “A Case for Two-way Skewed-associative Caches”, in *Proc. Int. Symp. on Computer Architecture,* 1993, pp. 169-178.

[31] A. Seznec and F. Bodin, “Skewed-associative Caches”, In *Proc. Int. Conf. on Parallel Architectures and Languages (PARLE),* 1993, pp. 305-316.

[32] A. J. Smith, “Cache Memories”, *ACM Computing Surveys,* vol. 14, no. 4, Sept. 1982, pp. 473-530.

[33] G.S. Sohi, “Logical Data Skewing Schemes for Interleaved Memories in Vector Processors”, Computer Sciences Technical Report #753, U. Wisconsin-Madison, Sept. 1988.

[34] A. Srivastava and A. Eustace, “ATOM: A System for Building Customized Program Analysis Tools”, in *Proc. SIGPLAN Conf. on Programming Language Design and Implementation,* 1994.

[35] *The National Technology Roadmap for Semiconductors*, Semiconductor Industry Association, San Jose, Calif., 1994.

[36] N.P. Topham, A. González and J. González, “The Design and Performance of a Conflict-avoiding Cache”, in *Proc. 30th Int. IEEE/ACM Symp. on Microarchitecture,* pp. 71-80, Dec. 1997.

[37] W-H Wang, J-L Baer and H.M. Levy, “Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy”, in *Proc. Int. Symp. on Computer Architecture*, 1989.