The Design and Performance of a Conflict-avoiding Cache

Nigel Topham,[†] Antonio González[*] and José González[*]

[†] Department of Computer Science
University of Edinburgh
JCMB, Kings Buildings, Edinburgh (UK)
Email:npt@dcs.ed.ac.uk

[*] Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
c/ Jordi Girona 1-3, 08034 Barcelona (Spain)
Email:{antonio,joseg}@ac.upc.es

Abstract

High performance architectures depend heavily on efficient multi-level memory hierarchies to minimize the cost of accessing data. This dependence will increase with the expected increases in relative distance to main memory. There have been a number of published proposals for cache conflict-avoidance schemes. In this paper we investigate the design and performance of conflict-avoiding cache architectures based on polynomial modulus functions, which earlier research has shown to be highly effective at reducing conflict miss ratios. We examine a number of practical implementation issues and present experimental evidence to support the claim that pseudo-randomly indexed caches are both effective in performance terms and practical from an implementation viewpoint.

1 Introduction

On current projections the next 10 years could see CPU clock frequencies increase by a factor of twenty whereas DRAM row-address-strobe delays are projected to decrease by only a factor of two. 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 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.

This paper addresses the issue of latency reduction and the degree to which future cache architectures can isolate their processor from increasing memory latency. We discuss the theory, and evaluate the practice, of using a particular class of conflict-avoidance indexing functions. We demonstrate how such a cache could be constructed and provide practical solutions to some previously unreported problems, as well as some known problems, associated with unconventional indexing schemes. The key contribution of this paper is to explore the design space of conflict-resistant caches and evaluate their performance on programs both with and without high levels of conflict misses.

In section 2 we present an overview of the causes of conflict misses and summarise previous techniques that have been proposed to minimize their effect on performance. We propose a method of cache indexing which has demonstrably lower miss ratios than alternative schemes, and summarise the known characteristics of this method. In section 3 we discuss a number of implementation issues, such as the effect of using this novel indexing scheme on the processor cycle time. We present an experimental evaluation of the proposed indexing scheme in section 4. Our results show how the IPC (instructions committed per cycle) of an out-of-order superscalar processor can be improved through the use of our proposed indexing scheme. Finally, in section 5, we draw conclusions from this study.

2 The problem of cache conflicts

Whenever a block of main memory is brought into cache a decision must be made on which block, or set of blocks, in the cache will be candidates for storing that data. This is referred to as the placement policy. 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

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

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 $b_1[j+k]$.

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. The following section proposes a remedy to the problem of cache conflicts by defining an improved method of block placement.

2.1 Conflict-resistant cache placement functions

The objective of a conflict-resistant placement function is to avoid the repetitive conflicts described above. This is analogous to finding a suitable hash function for a hash table. Perhaps the most well-known alternative to conventional cache indexing is the skewed associative cache [21]. This involves 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 [16], Harper and Jump [11], and Sohi [24] proposed skewing functions. The use of XOR functions was proposed by Frailong *et al*. [5], and pseudo-random functions were proposed by Raghavan & Hayes [17] and Rau *et al*. [18], [19]. 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 placement 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 both respects the irreducible polynomial modulus (I-Poly) permutation function proposed by Rau [19] is an ideal candidate.

The I-Poly scheme effectively defines families of pseudo-random hash functions which are implemented using exclusive-OR gates. They also have some useful behavioral characteristics which we discuss later. In [10] the miss ratio of the I-Poly indexing scheme is evaluated extensively in the context of cache indexing, and is compared with a number of different cache organizations including; direct-mapped, set-associative, victim, hash-rehash, column-associative and skewed-associative. The results of that study suggest that the I-Poly function is particularly robust. For example, on Spec95 an 8Kb two-way associative cache has an average miss ratio of 13.84%. An I-poly cache of identical capacity and associativity reduces that miss ratio to 7.14%, which compares well against a fully-associative cache which has a miss ratio of 6.80%.

2.1.1 Polynomial-modulus cache placement

To define the most general form of conflict resistant cache indexing scheme let the placement of a block of data from an n-bit address A, in each of w ways of a w-way associative cache with $M = 2^m$ sets, be determined by the set of indices $\{i_1, i_2, \ldots, i_w\}$. In I-Poly indexing each i_k is computed by the function $h_v(A, P_k) \in (0, M-1)$, for $m < v < n$. In this scheme P_k is a member of a set of w, possibly distinct, integer values $\{P_1, P_2, \ldots, P_w\}$, in the range $(M, 2M-1)$. If we choose to use distinct values for each P_i the cache will be skewed, though skewing is not an obligatory feature of this scheme. Each index function is defined as follows. Consider the integers A and P in terms of their binary representations, as shown for example in equation (iv)

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

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

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

Similarly P can be interpreted as the polynomial $P(x)$. For best performance $P(x)$ will be an irreducible polynomial, though it need not be so.

Each cache index $h_v(A, P)$ is also defined over GF(2), and is given by the polynomial $R(x)$ of order less than m computed by the following modulus operation

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

Effectively $R(x)$ is the polynomial modulus function

$A(x) \bmod P(x)$ ignoring the higher order $n-v$ terms of $A(x)$. Each bit of the index can be computed using an XOR tree, if $P(x)$ is constant, or an AND-XOR tree if one requires a configurable index function. For best performance v should be as close as possible to $n-1$, though it may be as small as m for this scheme to be distinct from conventional block placement. Examples of how this function can be applied to cache indexing can be found in [10].

2.1.2 Polynomial placement characteristics

The class of polynomial hash functions described above have been studied previously in the context of stride-insensitive interleaved memories (see [18] and [19]). These functions 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 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 1 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 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 [21] 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.

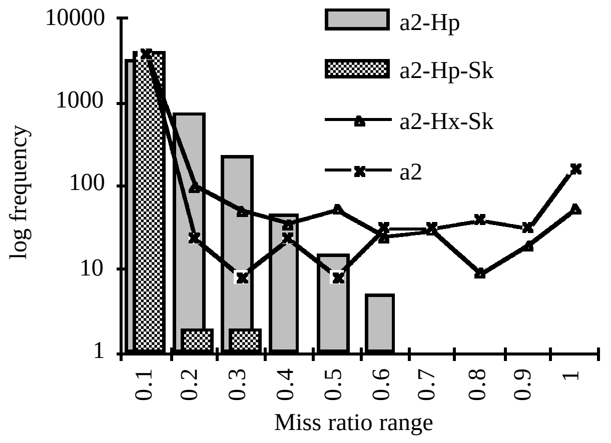

Figure 1. Frequency distribution of miss ratios for conventional and pseudo-random indexing schemes. Columns represent I-Poly indexing and lines represent conventional and skewed-associative indexing.

3 Implementation Issues

The logic of the polynomial modulus operation in GF(2) defines a class of hash functions which compute the cache placement of an address by combining subsets of the address bits 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.* [25] 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.

3.1 Overcoming page size restrictions

Typical operating systems permit pages to be as small as 4Kbytes. 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 256Kbytes 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 is 256Kbytes and the cache is 8Kbytes 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 [25] 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 3.2).

The fourth option would be appropriate for a physically-tagged direct-mapped cache. It is similar in principle to the hash-rehash [1] and the column-associative caches [2]. 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 a kind of 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. Space restrictions prevent further coverage of this option.

3.2 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 $1 \leq i < M$ 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.

A 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. This is guaranteed by the 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.

3.3 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

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

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

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

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:

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

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 8Kb two-way skewed 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 [25] 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.

3.4 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.

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 2. 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

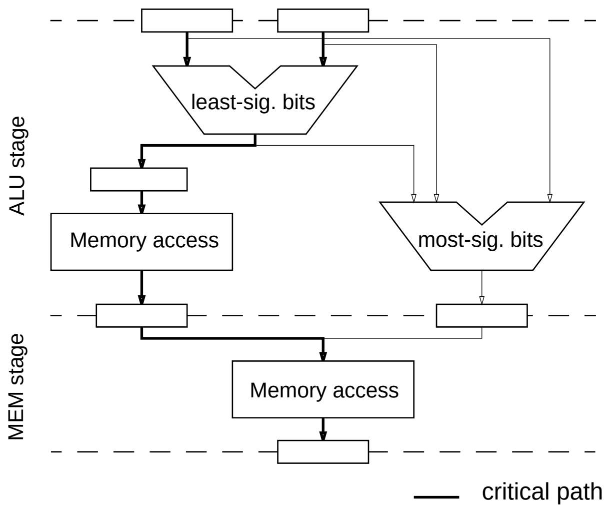

Figure 2. A pipeline that overlaps part of the address computation with the memory access.

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 [12]) 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 [9] it has been shown that the address 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 [7], a Load Target Buffer is presented, which predicts effective address adding a stride to the previous address. In [3] and [4] 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 [8], [9] and [20]. 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.

4 Experimental Evaluation

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 1 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 [14] that allows 8 outstanding misses to different cache lines. The cache size is either 8Kb 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 [6] and PA8000 [12]. A branch history table with 2K entries and 2-bit saturating counters is used for branch prediction.

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 interference 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.

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 1: Functional units and instruction latency.

Table 2 shows the IPC and the miss ratio for different configurations. The baseline configuration is an 8 Kb cache with I-poly 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 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 2 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 8Kb cache yields over 60% of the IPC increase that can be obtained by doubling the cache size.

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

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

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 3, 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 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 3 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 8Kb 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

	Conventional indexing					Polynomial mapping			
	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
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 3: 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.

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.

5 Conclusions

In this paper we have described a pseudo-random indexing scheme which is robust enough to eliminate repetitive cache conflicts. We have discussed the main 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. 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 I-poly indexing function.

Detailed simulations of an o-o-o superscalar processor have demonstrated that programs with significant numbers of conflict misses in a conventional 8Kb 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.

We believe the key contribution of I-poly 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 an I-poly 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. Tiling often introduces additional conflict misses which depend on array dimensions as well as stride. An I-poly cache would, for example, eliminate the need to compute conflict-free tile dimensions.

Acknowledgements

This work was supported by UK EPSRC grant K19723, British Council grant 1016 and the Spanish Ministry of Education (Acción Integrada Hispano-Británica 202 and CYCT TIC 429/95). The authors would like to thank the anonymous referees for their constructive comments.

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

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

[7] M.Golden and T.N. Mudge, "Hardware Support fot Hiding Cache Latency", Technical report # CSE-TR-152-93. University of Michigan, 1993.

[8] 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.

[9] 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.

[10] A. González, Mateo Valero, Nigel Topham and Joan M. Parcerisa, "Eliminating Cache Conflict Misses Through XOR-based Placement Functions", in *Proc. ICS '97*, Vienna, pp, 76-83, July 1997.

[11] 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.

[12] D. Hunt, "Advanced Performance Features of the 64-bit PA-8000", in *Proc. of the CompCon'95,* pp. 123-128, 1995.

[13] 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.

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

[15] 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).

[16] 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.

[17] R. Raghavan and J.P. Hayes, "On Randomly Interleaved Memories", in *Proc. Supercomputing '90*, pp. 49-58.

[18] 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.

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

[20] 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.

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

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

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

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

[25] 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.