Evaluation of the Performance of Polynomial Set Index Functions

Hans Vandierendonck and Koen De Bosschere
Dept. of Electronics and Information Systems
Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
E-mail: {hvdieren,kdb}@elis.rug.ac.be

Abstract

Randomising set index functions, randomisation functions for short, can significantly reduce conflict misses in data caches by placing cache blocks in a conflict-free manner. XOR-based functions are a broad class of functions that generally exhibit few conflict misses. Topham and González claimed that the sub-class of functions based on division of polynomials over $GF(2)$ contains those functions that achieve the best conflict-free mapping. Furthermore, they claim that an even smaller class of functions, namely those based on irreducible polynomials, perform better still. This paper investigates these claims in a novel way, by evaluating many randomisation functions from the different classes. The minimum, maximum and average miss rates are used to compare the different classes of randomisation functions. To avoid computing miss rates for all of these functions, we estimate the miss rates using a semi-analytical technique. The evaluation shows that polynomial randomisation functions are not the best possible choice, although they perform relatively well. Irreducible polynomials offer no measurable benefits over reducible polynomials, while they may be a lot more complex to implement.

1 Introduction

Randomising set index functions can remove a significant amount of conflict misses in data caches [11, 15] by spreading the cache blocks uniformly over all sets. Instead of selecting a slice of bits from the block address to index a cache, a randomising set index function computes a more complex function of the address, e.g.: using exclusive or's (XOR). We will use the shorter term *randomisation function* when we mean randomising set index function.

Although some work has shown the benefits of randomisation functions, architects are left with the implausible task of selecting just one randomisation function to implement in a processor. Few papers provide help on this task. In [6, 7, 14, 15], the class of functions based on division of polynomials over $GF(2)$, originally proposed for interleaved memories in vector processors [10], is promoted for use as set index functions for data caches. The performance of these functions in the presence of stride-based access patterns can be shown to be predictable. Except for a few functions in [7], no comparisons between these functions, called *polynomial randomisation functions*, and other XOR-based randomisation functions are made. Furthermore, it is claimed that polynomials that are irreducible, which means that they cannot be written as the product of other polynomials, will lead to even less conflict misses, although no experimental evidence is presented to support this claim [6, 7, 14, 15].

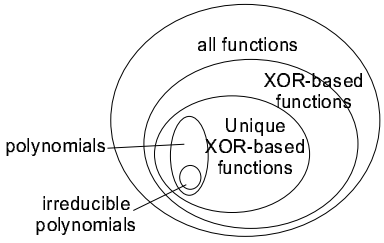


Figure 1. Different classes of randomisation functions.

The space of randomisation functions, together with some sub-classes of functions are shown in Figure 1. Functions outside the class of XOR-based functions include those using multiplication and division of integers [13, 11]. Because different XOR-based functions can induce the same number of misses in all circumstances (see below), the class of "unique XOR-based functions" retain only one of those functions. We limit our evaluation effort to these functions. Specific sub-classes are the polynomial and irreducible polynomial functions. This paper investigates the following questions: (i) do polynomial randomisation functions perform better than the general class of XOR-based

functions and (ii) do irreducible polynomial randomisation functions perform better than functions based on reducible polynomials? These questions will be answered both in the general case (for any functions) and in the particular case (for a specific function).

Performing exhaustive evaluation of the whole space of XOR-based randomisation functions would be too time-consuming. Therefore, we use a novel technique to explore the design space. We randomly generate a huge number of randomisation functions and estimate the miss rate of each function using a semi-analytical technique [16]. Because of the statistical nature of this, we can cover a wide range of randomisation functions including functions with both high and low miss rates. This statistical exploration gives us minimum, maximum and average miss rates for each class of randomisation functions. These numbers allow us to compare the relative performance of the different classes.

The remainder of this paper is organised as follows. We discuss related work first. In section 3 we review the different types of randomisation functions studied and show some of their mathematical properties that we use in our evaluation. The semi-analytical technique to estimate miss rates is reviewed in section 4. The evaluation results are presented in section 5 and section 6 concludes.

2 Related Work

In vector processors many conflicts can occur when accessing a multi-bank memory. XOR-based interleaving functions were shown to reduce the number of conflicts and reduce the access time of interleaved memories [5, 8, 10]. The polynomial mapping functions that were identified by Rau [10] for interleaved memories, were adopted by Topham *et al.* to reduce conflicts in data caches [6, 7, 14, 15]. They showed that randomisation significantly reduces conflicts in small caches, including direct mapped, set-associative, skewed-associative and hash-rehash caches [7]. Furthermore, they showed that the delay of the XOR-gates can be effectively hidden with address prediction [15].

Earlier work assumed functions involving multiplication of integers. Schlansker *et al.* [11] showed that randomised set index functions reduce miss rates for cyclic sweep patterns that are sent directly to a 32-way set-associative level-2 cache. In a theoretical study Smith concluded that the set index function has a high potential for reducing the miss rate [12], although the multiplication function tested in [13] did not perform significantly better than bit selection mapping.

Better algorithms and data structures can also help to reduce conflict misses. Lam *et al.* [9] analysed blocking algorithms and showed that the size of blocks and the data layout have a high impact on conflict misses. Cache misses for pointer-based data structures can be removed by clustering and colouring of individual structures [4] and by structure splitting and field reordering [3]. These optimisations target both conflict and capacity misses. In [2], the compiler uses profiling information to improve data placement and avoid conflict misses.

Clearly, randomisation and clever software techniques have different benefits and limitations. Randomisation works relatively well for all programs and requires no intervention of the programmer. On the other hand, a non-randomised cache may exhibit close-to-optimal miss rates when running carefully designed algorithms that are backed up by an aggressive compiler. But this approach requires the commitment of the programmer to define the data structures correctly and select the appropriate algorithms. This may put too much responsibility on the programmer. Furthermore, we believe that it is possible to combine randomisation and software techniques. When a compiler lays out code for a conventionally indexed cache, it takes into account the number of sets and the set index function. It is possible to amend these algorithms to take into account a randomising set index function.

The semi-analytical technique for estimating the miss rate of randomisation functions was proposed and evaluated in [16]. This paper applies the technique to investigate properties of randomisation functions.

3 Models of Set Index Functions

XOR-based set index functions can be modelled by means of (i) a matrix representation (ii) division of polynomials or (iii) using null spaces. Each of these models is reviewed and their relations are discussed.

3.1 The Matrix Representation

The matrix representation can be used to represent any randomisation function that can be computed using solely XOR-gates operating on the address bits. In this representation, an n-bit block address $\mathbf{a}$ is represented by a bit vector[1] $a_{n-1}a_{n-2}\ldots a_0$, with a_{n-1} the most significant bit and a_0 the least significant bit. A randomising set index function mapping n to m bits is represented as a binary matrix H with n rows and m columns:

$$H = \begin{bmatrix} h_{n-1,m-1} & h_{n-1,m-2} & \cdots & h_{n-1,0} \\ h_{n-2,m-1} & h_{n-2,m-2} & \cdots & h_{n-2,0} \\ \vdots & \vdots & & \vdots \\ h_{1,m-1} & h_{1,m-2} & \cdots & h_{1,0} \\ h_{0,m-1} & h_{0,m-2} & \cdots & h_{0,0} \end{bmatrix}.$$

[1] Vectors are written in boldface type.

The bit $h_{r,c}$ on row r and column c is 1 when address bit a_r is an input to the XOR-gate computing the c-th set index bit. Consequently, the computation of the set index $\mathbf{s}$ can be expressed as the vector-matrix multiplication over $GF(2)$, the domain $\{0,1\}$ where addition is computed as XOR and multiplication is computed as logical *and*, denoted by $\mathbf{s} = \mathbf{a}\,H$.

3.2 Polynomial Division

Polynomial set index functions are expressed as division of polynomials over $GF(2)$. In a polynomial $A(x) = a_{n-1}x^{n-1} + \cdots + a_1x^1 + a_0$ over $GF(2)$, the coefficients $a_i, 0 \leq i < n$ take the values 0 or 1. Addition, subtraction, multiplication and division are similar to those operations for conventional polynomials, except that all operations on coefficients are performed over $GF(2)$.

An address $\mathbf{a} = a_{n-1} \cdots a_1 a_0$ corresponds to the polynomial $A(x) = a_{n-1}x^{n-1} + \cdots + a_1x_1 + a_0$ of order n (i.e.: all coefficients a_i with $i \geq n$ are 0). Let $P(x)$ be a polynomial of order m, then $A(x)$ can be uniquely decomposed into the polynomials $V(x)$ and $R(x)$ as

$$A(x) = V(x) * P(x) + R(x)$$

where the order of $R(x)$ is less than m. The bit-vector corresponding to $R(x)$ is used as the set index for address $\mathbf{a}$ when the polynomial set index function defined by $P(x)$ is used.

The division of polynomials over $GF(2)$ can be computed very efficiently. In fact, every polynomial set index function has a corresponding matrix H over $GF(2)$. The ith row $H_{i,*}$ of this matrix corresponds to the polynomial $R_i(x) = (x^i \bmod P(x))$ [10]. These rows can be generated using a feedback shift register as follows. First, $R_0(x) = 1$. Then, each $R_{i+1}(x)$ is computed as $R_i(x) * x + P(x)$ if the coefficient of x^{m-1} in $R_i(x)$ is 1 and as $R_i(x) * x$ if that coefficient is 0. Note that this method guarantees that each $R_i(x)$ is at most of order m.

A special sub-class of polynomial set index functions are the functions defined by *irreducible polynomials*. An irreducible polynomial (I-poly) is a polynomial over $GF(2)$ that is not divisible by any other polynomial over $GF(2)$. In terms of the matrix representation, this implies that rows 0 to $2^m - 1$ inclusive are all different (i.e.: the feedback shift register generates a sequence of m-bit rows with a maximum period of $2^m - 1$). This property is beneficial for interleaved memories [10]. E.g.: in a 16-way interleaved memory, $m = 4$ and an I-poly randomisation scheme would randomise $2^4 - 1 = 15$ address bits[2]. Using a reducible polynomial would result in less randomisation. Another benefit of I-poly schemes is in the case where the addresses $A_0, A_1, \ldots, A_K$ are referenced and $P(x) \bmod A_i(x) = G(x)$ for each $i = 0, \ldots, K$. In such a case, it was shown that only $2^{(m-q)}$ memory banks are used if $m = order(P(x))$ and $q = order(G(x))$ [10]. If $P(x)$ is irreducible, then it is always the case that $q = 0$. This situation occurs when an address sequence with stride S is accessed, and S contains long sequences of 0's between the 1's [10].

As important as irreducible polynomials are for interleaved memories, irreducible polynomials offer no significant benefit for set index functions. First, the number of distinct rows in the matrix representation is purely theoretical in the case of set index functions, since one would need a system with at least $2^m - 1$ (virtual) address bits in order to exploit the number of distinct rows. E.g.: an 8 kB direct mapped cache with 32 byte blocks has $m = 8$ and $2^m - 1 = 255$. A system with 64 bit (virtual) addresses would use at most 59 of those rows. Hence, reducible polynomials with a period that is longer than the number of (virtual) address bits used by the system are equally appropriate from a theoretical viewpoint. The difference in performance obtained by either of the set index functions is determined by the actual values of the $R_i(x)$, more than the fact that all $R_i(x)$ are pair-wise different.

We show that the second benefit is negligible when we investigate the contents of a data cache, because the most important stride for *storing* data is 1 (even though the most important stride for *accessing* that data may be different). Therefore, the situation described above is not relevant for set index functions.

3.3 The Null Space

Every function in the design space of XOR-based set index functions can be represented by the *null space* of its matrix. The null space $N(H)$ of a matrix H is the set of all vectors that are mapped to the zero vector:

$$N(H) = \{\mathbf{x} \in \{0,1\}^{1\times n} \mid \mathbf{x}\,H = \mathbf{0}\}.$$

Two addresses cause conflict misses when they are mapped to the same set of the cache: $\mathbf{x}\,H = \mathbf{y}\,H$. This implies that the bitwise XOR of these addresses is a member of the null space of the set index function:

$$\begin{aligned} & \mathbf{x}\,H = \mathbf{y}\,H \\ \iff\quad & (\mathbf{x} \oplus \mathbf{y})\,H = \mathbf{0} \\ \iff\quad & (\mathbf{x} \oplus \mathbf{y}) \in N(H). \end{aligned}$$

We call the members of $N(H)$ *conflict vectors* because they determine whether two addresses can cause conflict misses.

For an $n \times m$ matrix H in which all columns are linearly independent, the null space has dimension $n - m$ and contains 2^{n-m} n-bit vectors. Hence, for increasing n and fixed

[2]The next 15 address bits are randomised in the same way as the first 15, because the shift register goes through the same 15 states again.

m, the number of possible null spaces, and randomisation functions, increases exponentially. This suggests that by increasing the number of randomised address bits, it should be possible to obtain better randomisations.

Different matrices can have the same null space. In such cases, the matrices would result in exactly the same cache misses for the same blocks, although these misses would occur in a different set.

Matrices can be manipulated without changing their null space. The null space remains unchanged under two operations on the columns of the matrix H, namely (i) swapping two columns and (ii) adding one column to another.

3.4 Reducing the Design Space

The design space of XOR-based set index functions can be reduced without removing interesting functions in two steps: (i) by removing clearly uninteresting functions and (ii) by allowing only one function per null space.

It is reasonable to require that randomising set index functions must place 2^m consecutive blocks into distinct sets of the cache[3], because consecutive cache blocks are often simultaneously present in a cache due to spatial locality. A necessary and sufficient condition is that the least significant m address bits are involved in the randomisation (e.g.: when the least significant bit is not used, then blocks $a_{n-1} \ldots a_1 0$ and $a_{n-1} \ldots a_1 1$ are always mapped to the same set). In other words, the matrix should have at least one 1-bit on each of the rows $H_{*,0}$ through $H_{*,m-1}$.

Based on the observations above on swapping and adding columns, every matrix that uses the lowest m address bits can be transformed into a matrix with a diagonal of 1's in its lowest m rows. Hence, if we only consider matrices with such a diagonal, then we will consider only one matrix per null space in our evaluation. This allows us to estimate the number of possibly usefull matrices with n rows and m columns to $2^{m(n-m)}$. This is a fraction of 2^{-m^2} of the whole space of matrices. All of these matrices are different, i.e.: for each pair of matrices, there exists a memory reference sequence for which different miss rates are obtained.

3.5 Reducing Implementation Cost

Since many matrices have the same null space, and thus the same effect on the number of cache misses, it is possible to choose that matrix that results in the best circuit-level properties, such as latency, area and power consumption[4]. The latency of a set index function can be roughly estimated by the maximum fan-in per XOR-gate, or the maximum number of 1-bits per column. This delay can reduced by applying suitable operations on the columns of a matrix. This optimisation is illustrated in a few steps on the matrix corresponding to the irreducible polynomial $x^7 + x^6 + x^5 + x^4 + x^3 + x^0$, corresponding to integer 505. This polynomial was investigated for direct mapped caches in [7]. It can be optimised by (Figure 2): adding column 0 to column 3, adding column 7 to column 4 and adding column 7 to column 5. We also show that the maximum load on each driver feeding the address bits to the XOR-gates (max. fan-out) is simultaneously reduced.

4 Evaluation Procedure for XOR-Based Set Index Functions

We have opened up the design space for XOR-based set index functions to include $2^{m(n-m)}$ matrices. Since this number is huge, we need a way to quickly determine which matrices are more interesting than others. We use a semi-analytical technique for this purpose. This technique uses profiling information about a program, together with the null space of a matrix to assign a score to the matrix. Matrices that incur fewer misses will (generally) have lower scores and hence the best-performing matrices can be detected.

The score for a randomisation function is decomposed into a score for each vector in its null space:

$$score(H) = \sum_{\mathbf{v} \in N(H)} cost(\mathbf{v}).$$

We compute $cost(\mathbf{v})$ as an estimate of the number of conflict misses caused by $\mathbf{v}$, if that vector were a member of the null space of a matrix. The estimates $cost(\mathbf{v})$ for each vector $\mathbf{v}$ are computed during a profiling run of the program and stored in a profile. Once the profile is known, the required steps consist of (i) computing the null space of the matrix, (ii) obtaining the cost of each vector in the null space from the profile and (iii) adding all costs into a score (Figure 3).

The profile consisting of the values $cost(\mathbf{v})$ is obtained during a profiling run. The profiling algorithm first eliminates compulsory and capacity misses, since they can not be removed by the set index functions. Each reference that misses in an ideal cache (i.e.: a fully associative cache using LRU replacement) of the same size as the investigated cache, is considered to be either a compulsory or a capacity miss and is ignored by the profiling algorithm.

For the other references, the profiling algorithm computes the circumstances under which they could be a con-

[3] Note that randomising set index functions based on XOR's can only achieve this for 2^m blocks that are aligned on a 2^m boundary. Functions of this type are permutation-based functions, since each chunk of 2^m blocks is independently permuted.

[4] These properties can, of course, also be optimised against each other by scaling the sizes of the transistors, etc.

	Matrix 1	Matrix 2	Matrix 3	Matrix 4
column	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
	1 . 1 1	1 . 1 1	1 . 1	1 . . . .
	. 1 . 1 1 . . .	. 1 . 1 1 . . .	. 1 . 1 1 . . .	. 1 . 1 1 . . .
	. . 1 . 1 1 . .	. . 1 . 1 1 . .	. . 1 . 1 1 . .	. . 1 . 1 1 . .
	. . . 1 . 1 1 .	. . . 1 . 1 1 .	. . . 1 . 1 1 .	. . . 1 . 1 1 .
	. . . . 1 . 1 1	. . . . . 1 1	. . . . . 1 1	. . . . . 1 1
	1 1 1 1 1 . . 1	1 1 1 1 . . . 1	1 1 1 1	1 1 1
	1	1	1 . . 1	1 . 1 1 . . . .
	. 1	. 1	. 1	. 1 . . . . . .
	. . 1	. . 1	. . 1	. . 1 . . . . .
	. . . 1	. . . 1	. . . 1	. . . 1 . . . .
	. . . . 1 . . .	. . . . 1 . . .	. . . . 1 . . .	. . . . 1 . . .
	. . . . . 1 . .	. . . . . 1 . .	. . . . . 1 . .	. . . . . 1 . .
	. . . . . . 1 .	. . . . . . 1 .	. . . . . . 1 .	. . . . . . 1 .
	 1	. . . . 1 . . 1	. . . . 1 . . 1	. . . . 1 . . 1
fan-in	3 3 4 5 5 3 3 3	3 3 4 5 4 3 3 3	3 3 4 4 4 3 3 3	3 3 3 4 4 3 3 3
max. fan-out	6	5	4	3

Figure 2. Four matrices that always generate the same conflict misses.

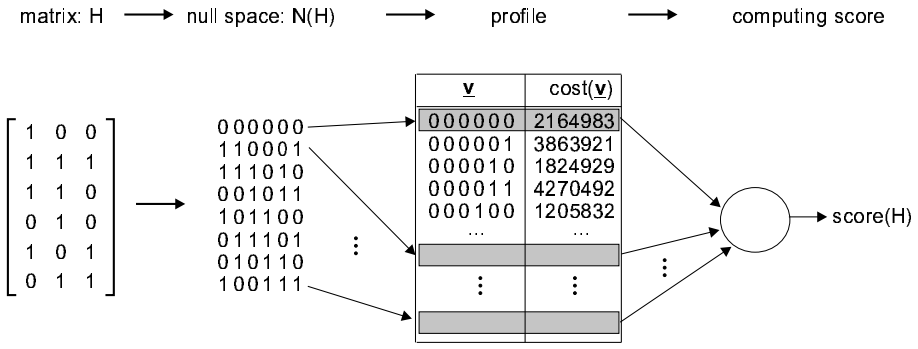


Figure 3. Evaluating randomising set index functions.

flict miss. To see how this works, we first discuss an example memory reference stream: $\mathbf{d}\ \mathbf{a}\ \mathbf{b}\ \mathbf{c}\ \mathbf{b}\ \mathbf{a}$. The cache block with address $\mathbf{a}$ is referenced twice. The first reference to $\mathbf{a}$ is a compulsory miss. We assume the cache is large enough such that the second reference to $\mathbf{a}$ is either a hit or a conflict miss, depending on the set index function. This reference will be a conflict miss when one or more of the cache blocks $\mathbf{b}$ and $\mathbf{c}$ is mapped to the same frame as $\mathbf{a}$. These blocks will displace $\mathbf{a}$ from the cache before it is referenced for the second time. Note that $\mathbf{d}$ exerts no influence on this. Because the blocks $\mathbf{b}$ and $\mathbf{c}$ can displace $\mathbf{a}$ from the cache, the cost of having the conflict vectors $\mathbf{a} \oplus \mathbf{b}$ and $\mathbf{a} \oplus \mathbf{c}$ in the null space is incremented by 1.

The profiling algorithm as described above can be implemented using an LRU stack. When profiling a reference to a block $\mathbf{a}$, then the blocks that have been referenced since the previous reference to $\mathbf{a}$ are found on the LRU stack above block $\mathbf{a}$. For each of these blocks $\mathbf{a}_j$, we compute the conflict vector $\mathbf{v} = \mathbf{a} \oplus \mathbf{a}_j$ and increment $cost(\mathbf{v})$ by one (Figure 4). This algorithm is a more accurate variant of the one described in [16]. It will generally overestimate the number of conflict misses.

5 Evaluation

The evaluation uses the SPEC95 benchmarks compiled for the MIPS-like instruction set used by SimpleScalar [1]. Each benchmark runs the training input set during 500 million instructions. We evaluate randomising set index functions for an 8 kB direct mapped data cache with 32 byte blocks. The cache allocates blocks on load and store misses and we assume the absence of hardware prefetching.

```
Let a_i = block address of reference i
Let n = length of vectors
Let cost(v) = accumulated cost for vector v,
      initially 0

for each reference i in program trace do
      if a_i is a compulsory miss then
            push a_i on stack
      elsif a_i is a capacity miss then
            move a_i to top of stack
      else      /* Count conflict vectors */
            for each a_j on stack above a_i do
                  v = (a_i ⊕ a_j) truncated to n bits
                  increment cost(v)
            od
            move a_i to top of stack
      fi
od
```

Figure 4. The profiling algorithm.

5.1 The Impact of Randomisation

Randomising set index functions based on polynomial division have a strong impact on data cache miss rates. We computed the miss rate for all polynomial set index functions for the 8 kB cache for associativities of 1, 2 and 4. We show the miss rate of the best performing and the worst performing functions, together with the miss rates for conventionally indexed set-associative caches with LRU replacement (Figure 5). For many benchmarks, the best randomising set index functions results in miss rates comparable to that of a two-way or a four-way set-associative cache. However, the worst randomisation functions almost always increases the miss rate of the direct mapped cache. Thus, not all randomisation functions perform equally well.

The results also show that randomisation improves performance when combined with associativity, as is the case for ijpeg. For other programs, e.g. go, randomisation helps, but adding associativity helps much more. In the case of fpppp, both randomisation and associativity improve performance to the same extent. On the other hand, conflict misses in swim are removed by randomisation, but associativity does not help at all. Randomisation and associativity are different aspects: different programs benefit more from different techniques. Most of the benefits of randomisation is obtained in direct mapped caches. Hence, we limit this study to direct mapped caches.

Besides higher associativity, large caches may also make randomisation uninteresting. We computed miss rates for 100 randomly selected polynomial functions for direct mapped caches of 8 and 16 kB with 32 byte lines and of 32 and 64 kB with 64 byte lines (Figure 6). The figure shows the relative reduction in miss rate when the cache size is increased. The most spectacular benefits from randomisation are seen in the smallest caches: 55% of cache misses is removed. Around one third of the misses is removed in larger caches. The graph also shows that the addition of randomisation to a cache can give the same benefits in terms of miss rate as a doubling in cache size.

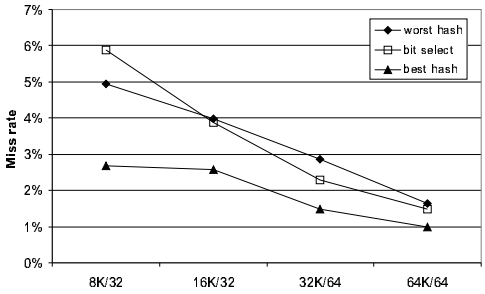


Figure 6. Randomisation versus cache size.

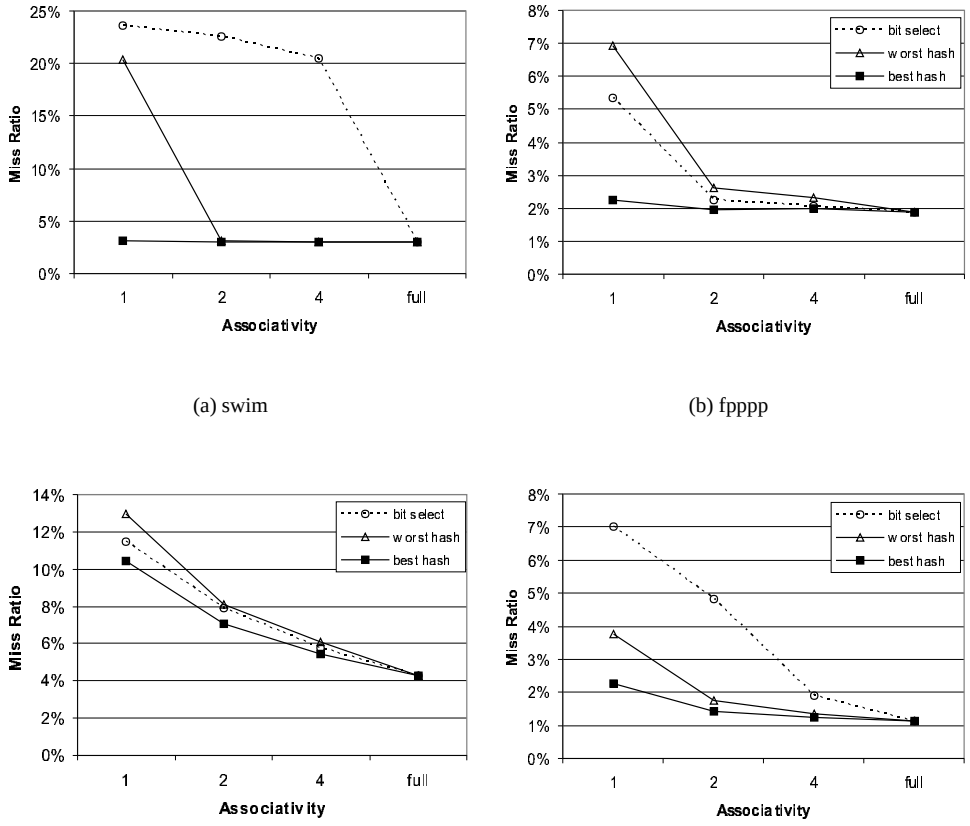
The motivation for using polynomial set index functions is that an architect can be confident that the performance of programs will be predictable [15]. Our findings do not validate this assumption. We computed the miss rates for several polynomial set index functions when the number of randomised address bits is varied (i.e.: the number of rows in the matrices is varied) (Figure 7). One expects that by hashing more address bits miss rates should decrease and not increase, because the address space is randomised to a greater extent.

However, we found that the miss rates vary strongly in an unpredictable way. E.g., for polynomial 505, the miss rate drops significantly for the floating-point benchmarks when 14 bits are randomised, but for 15 bits, the miss rate is as high as it is for 13 bits. This kind of behaviour suggests that the stride-based patterns, for which the performance of polynomial randomisation functions is predictable, are not the most important patterns when studying set index functions. Specifically, the performance of polynomial randomisation functions is only predictable when a single big array with a possibly non-unit stride is stored in the cache [10]. Under this assumption, randomising more input bits always implies lower miss rates.

5.2 Investigating the Contents of a Data Cache

In this section, we check whether the contents of a data cache consist of one big array with a possibly non-unit stride, or maybe a few arrays, each with a possibly different stride.

The contents of an 8 kB ideal cache with 32 byte blocks



(a) swim

(b) fpppp

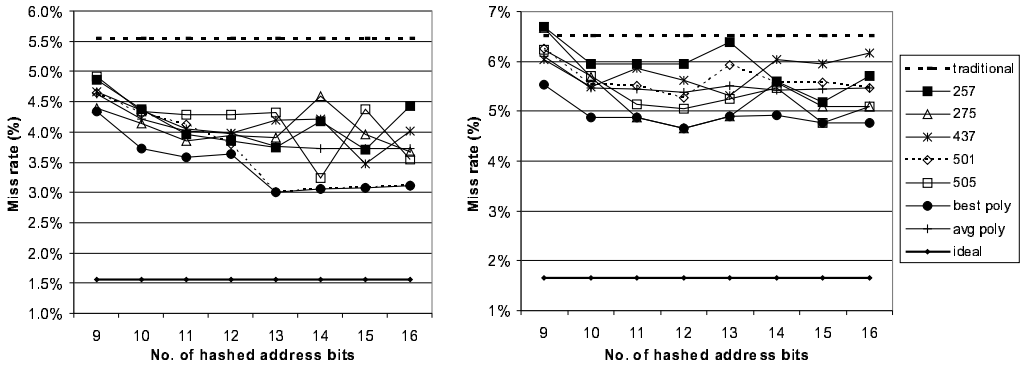
(c) go

(d) ijpeg

Figure 5. Combining randomisation and associativity.

are characterised as a number of arrays. After each memory reference, we divide the contents of the cache into a number of arrays. Each block can be present in only one array and we assume that arrays do not overlap. The statistics for the length, the stride and the number of arrays is averaged over all memory references (Table 1). Not all data is part of an array (column array), e.g.: because they are part of a non-array data structure or because only some elements of the array are in the working set of the program. On average, the method recognises 94% of the cache contents for SPECfp and 64% for SPECint as part of an array. This matches the intuition that the SPECfp programs are more regular than the SPECint programs. The part of the cache contents that is a member of an array, is spread over 15 (SPECfp) or 20 (SPECint) different arrays. The arrays are on average 28 (SPECfp) or 8 (SPECint) cache blocks long and have an average stride of 2.5 (SPECfp) or 3 (SPECint) cache blocks. Furthermore, the longest arrays usually have a small stride and stride 1 is the most common: Around 81% (SPECfp) or 62% (SPECint) of the array data is stored in arrays with unit stride (column stride1). Exceptions are fpppp, go and vortex. We conclude that the data cache contents consists of chunks of consecutive data, separated by uncached data, although here and there an isolated cache block is interspersed between the arrays. Hence, the polynomial model is maybe not the best model for randomising set index functions, since the underlying assumption that one big array with some (non-unit) stride is cached at any one time is clearly not valid.

If applications access data frequently with non-unit strides, why then does the data cache contains mostly unit stride arrays? The contents of the data cache is not entirely dictated by the access pattern of a program. For instance, when a program accesses an array with a stride of 4, but it does so once for the array starting at address A, and once for $A + 1$, $A + 2$ and $A + 3$ then, assuming the cache is large enough, the whole array will be cached and the dominant stride present in the cached data is 1. This situation could



(a) Floating-point benchmarks

(b) Integer benchmarks

Figure 7. Miss rates for different polynomial set index functions for increasing number of randomised address bits.

occur for a loop that is 4 times unrolled. Another reason is that many strides that can be observed in access patterns are smaller than the cache block size (32 bytes).

5.3 Accuracy of Estimated Miss Rates

We have shown that the mathematical model of the cache contents on which the theory for the polynomial functions is built is not valid. Still, we need to answer the question how good the polynomial index functions perform, relative to other randomisation functions. We use the semi-analytical technique to search the space of all possible matrices.

We validated the technique by generating both the miss rates and the scores for 64 polynomial set index functions. The accuracy of the scores can be described in two ways: (i) the ranking accuracy determines whether lower scores correspond to lower miss rates and (ii) the relative accuracy determines whether the miss rates (or number of conflict misses) can be correctly estimated.

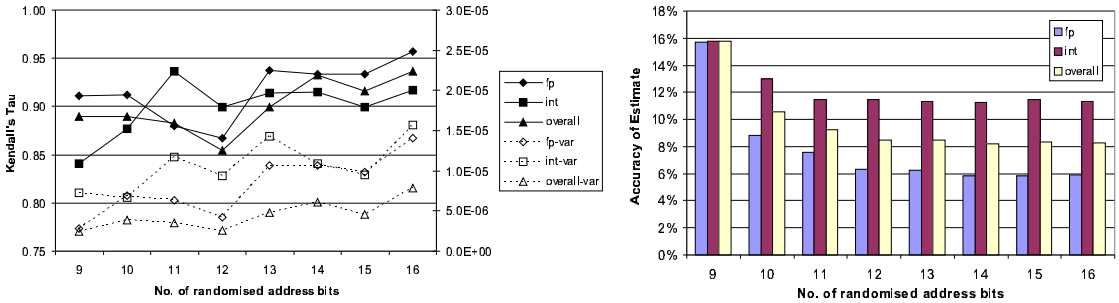
The ranking accuracy is computed as Kendall's τ coefficient, measuring the number of inversions of rank (i.e.: a set index function with a lower miss rate has a higher score). The τ coefficient is a correlation coefficient between the miss rates and the scores and is limited to the range $(-1, 1)$. The results show good correlation coefficients in the range of 0.85–0.95 (Figure 8(a)). We see that the coefficients vary strongly with the number of randomised address bits. This is caused mainly by the variation present in the miss rates (dotted lines), because it is harder to correctly rank the set index functions if their miss rates are packed close together.

The miss rate can be estimated from the score by adding the number of compulsory and capacity misses and dividing by the number of memory references. The relative accuracy is simply the relative error on the estimated miss rate (Figure 8(b)). The relative error is significantly larger for the integer programs than it is for the floating-point programs. The scores are overestimated more when a smaller number of address bits is randomised. By combining the results in Figure 8, we conclude that there is a large constant error present in the scores, but the part of the error that differs between the set index functions is relatively small.

5.4 Are Polynomial Set Index Functions Better?

We randomly generated 10 million matrices with $n = 9, \ldots, 16$ rows and $m = 8$ columns. Each matrix has a diagonal of 1's in its lower 8 rows, as described in section 3.4. For each matrix, we computed the score and kept track of the minimum, maximum and average score (Figure 9). We also computed the same statistics for all 128 (even) polynomial set index functions $257, 259, \ldots, 511$. The results show that on average, the polynomial set index functions are much better than the other set index functions. However, in many cases, there exists randomising set index functions that (are predicted to) perform better than the best polynomial set index function. In the worst case, the best polynomial set index function with 16 rows for the floating-point benchmarks has a 21.4% higher score (estimated conflict miss rate) and a 7.2% higher estimated miss rate (the sum of the scores and the compulsory and capacity misses) than the best function found.

To validate this sub-optimality of the polynomial set index functions, we computed the miss rates for the set index functions with the 10 lowest scores. Of those, we consider



(a) Ranking Accuracy

(b) Relative Accuracy

Figure 8. Validation of Semi-Analytical Estimation Technique

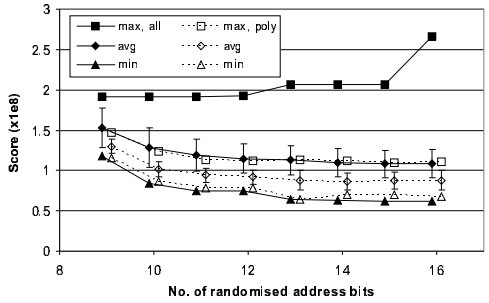


Figure 9. Scores for the polynomial and all randomisation functions.

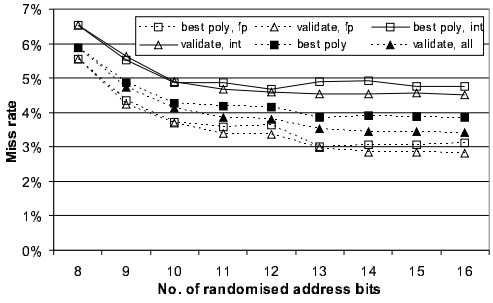


Figure 10. Miss rates for best polynomial set index function and best function found.

the function with the lowest miss rate as the best function that our techniques can find. The miss rates obtained with that function are shown in (Figure 10). The best polynomial set index function almost always has a significantly higher miss rate then the best one overall. Table 2 lists the relative amount that the estimated miss rate for the best polynomial set index function is higher than the best set index function found. The simulation results show that the difference can be as high as 13.8% (not 7.2% as the estimates showed). Note that the difference is higher for all benchmarks together than it is for the floating-point and integer benchmarks separately. The former number is based on set index functions optimised for the whole set of benchmarks and is not an average of the results for the integer and floating-point benchmarks.

We conclude that the polynomial set index functions are on average better than the general class of functions, although there exist functions that are significantly better than the polynomial set index functions.

5.5 Irreducible Polynomials

We investigate whether polynomial set index functions using irreducible polynomials really works better than using other polynomials. We computed the scores for all 128 polynomials that are applicable to the 8 kB direct mapped cache, of which 30 are irreducible. The minimum, maximum and average score for the irreducible and reducible polynomials is shown separately in Figure 11.

The average and the deviation of the scores for the irreducible polynomials does not differ significantly from those of the reducible polynomials. The irreducible polynomials show clearly better worst-case behaviour and they have similar best-case behaviour than the reducible ones. Specifically, when considering all benchmarks and randomising 13 address bits, the best irreducible polynomial is 3.2% worse than the best reducible one and when randomising 14 address bit, the best irreducible polynomial is 2% better. This leads us to the conclusion that irreducible polynomials

Table 1. Characterisation of the contents of an 8 kB ideal cache using strided arrays.

SPECfp95 Benchmarks					
Program	array	no.	length	stride	stride1
applu	99%	13.71	18.43	1.47	85.21%
apsi	96%	27.72	5.30	1.43	84.91%
fpppp	56%	21.47	4.80	4.43	3.60%
hydro2d	100%	3.28	47.45	1.73	98.02%
mgrid	99%	9.81	23.12	1.31	86.22%
su2cor	100%	1.85	110.64	1.05	98.20%
swim	100%	5.22	27.00	1.23	99.51%
tomcatv	100%	5.82	26.28	1.00	99.92%
turb3d	94%	47.17	4.31	9.83	74.25%
wave5	97%	11.81	13.86	1.16	81.96%
fp-avg	94%	14.79	28.13	2.46	81.18%
SPECint95 Benchmarks					
Program	array	no.	length	stride	stride1
compress	35%	25.05	1.93	2.43	84.27%
gcc	71%	19.49	9.29	1.51	82.33%
go	36%	23.35	2.66	2.74	0.04%
ijpeg	96%	15.00	14.35	1.15	92.41%
li	69%	25.58	6.88	1.29	82.61%
m88ksim	88%	29.16	2.03	1.53	65.47%
perl	66%	3.83	27.82	10.00	83.46%
vortex	49%	18.11	5.23	3.65	8.81%
int-avg	64%	19.95	8.77	3.04	62.43%
avg	80%	17.08	19.53	2.72	72.84%

Table 2. Percentage increase in miss rate of best polynomial set index function over all set index functions.

Rows	FP	INT	overall
9	2.45%	-1.57%	2.86%
10	1.00%	-0.62%	3.62%
11	5.05%	4.32%	8.91%
12	8.43%	1.49%	9.75%
13	0.70%	7.65%	9.42%
14	7.50%	8.45%	13.80%
15	8.39%	4.53%	13.11%
16	10.74%	5.70%	12.76%

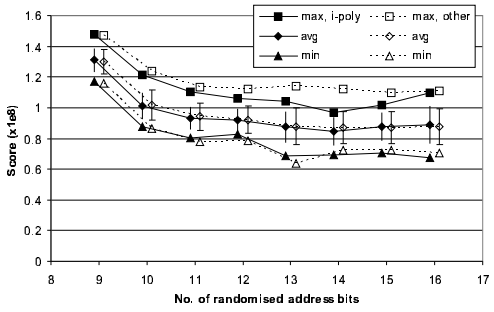


Figure 11. The distribution of the scores for irreducible and reducible polynomials over all benchmarks.

do not perform significantly better or worse than reducible polynomials on the tested benchmarks.

5.6 Implementation Complexity

Different matrices can have the same null space and hence result in the same performance. However, circuits implementing the functions may have different properties. As an example, we use the maximum number of 1-bits per column of the matrix as a crude approximation of the complexity (latency) of the resulting circuit.

The maximum number of 1-bits per column varies from 1 to 9 (16 address bits are randomised). For each of these complexity numbers, we randomly generated 1 million matrices (Figure 12). The polynomial set index functions were also optimised with respect to our complexity measure. More complex functions generally achieve lower scores than less complex ones, although this does not hold for the polynomial functions at 7 inputs per XOR-gate. The polynomial set index functions need to be more complex than other set index functions in order to achieve the same score: e.g.: polynomial functions with a complexity factor of 4 or more have a higher score than the best set index function with a complexity factor of 3.

The irreducible polynomials have scores similar to the reducible ones, but they are inherently more complex: for 16 randomised address bits, irreducible polynomials always need up to 5 or 6 inputs per XOR-gate. This again pleads against the use of irreducible polynomials.

6 Conclusions

Randomisation functions can significantly reduce conflict misses in data caches. It was claimed that functions based on division of polynomials over $GF(2)$ result in the

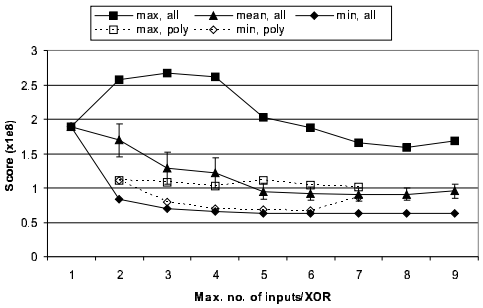


Figure 12. Distribution of scores classified by implementation cost for $n = 16$.

least conflict misses however, this claim has not been validated.

We compared the polynomial set index functions to other XOR-based functions. Because the design space of randomising set index functions is very large (even when considering only XOR-based functions), we developed a technique that quickly estimates the miss rate incurred by a randomisation function. This allows us to search through a large part of the design space of randomisation functions.

Application of this technique shows that polynomial set index functions are on average better than other XOR-based set index functions. However, there exist set index functions that clearly perform better than the best polynomial set index functions, with differences in the miss rate up to 13.8%. Irreducible polynomials are assumed to incur fewer conflicts than reducible polynomials. However, our analysis shows no clear difference.

Using a crude approximation of the complexity of the circuit, we showed that polynomial randomisation functions are more complex than other functions with the same miss rates. Irreducible polynomials result in still more complex circuits.

Acknowledgements

We would like to thank the anonymous referees for their useful comments. Hans Vandierendonck is supported by the Flemish Institute for the Promotion of Scientific-Technological Research in the Industry (IWT).

References

[1] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Technical report, Computer Sciences Department, University of Wisconsin-Madison, July 1996.

[2] B. Calder, C. Krintz, S. John, and T. A. Austin. Cache-conscious data placement. In *Eight International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, 1998.

[3] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure definition. In *ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pages 13–24, 1999.

[4] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[5] J. M. Frailong, W. Jalby, and J. Lenfant. XOR-schemes: A flexible data organization in parallel memories. In *Proceedings 1985 International Conference on Parallel Processing*, pages 276–283, Aug. 1985.

[6] A. González, M. Valero, N. Topham, and J. Parcerisa. On the effectiveness of XOR-mapping schemes for cache memories. Technical Report UPC-CEPBA-1996-14, Universitat Politècnica de Catalunya, 1996.

[7] A. González, M. Valero, N. Topham, and J. Parcerisa. Eliminating cache conflict misses through XOR-based placement functions. In *ICS'97. Proceedings of the 1997 International Conference on Supercomputing*, pages 76–83, July 1997.

[8] D. T. Harper III and D. Linebarger. A dynamic storage scheme for conflict-free vector access. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 72–77, May 1989.

[9] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Apr. 1991.

[10] B. R. Rau. Pseudo-randomly interleaved memory. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 74–83, May 1991.

[11] M. Schlansker, R. Shaw, and S. Sivaramakrishnan. Randomization and associativity in the design of placement-insensitive caches. Technical Report HPL-93-41, HP Laboratories, June 1993.

[12] A. J. Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Transactions on Software Engineering*, SE-4(2):121–130, Mar. 1978.

[13] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, Sept. 1982.

[14] N. Topham and A. González. Randomized cache placement for eliminating conflicts. *IEEE Transactions on Computers*, 48(2):185–192, Feb. 1999.

[15] N. Topham, A. González, and J. González. The design and performance of a conflict-avoiding cache. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 71–80, Dec. 1997.

[16] H. Vandierendonck and K. De Bosschere. Efficient profile-based evaluation of randomising set index functions for cache memories. In *International Symposium on Performance Analysis of Systems and Software*, pages 120–127, Nov. 2001.