Application-Specific Reconfigurable XOR-Indexing to Eliminate Cache Conflict Misses

Hans Vandierendonck
Ghent University
Dept. ELIS / HiPEAC
St.-Pietersnieuwstraat 41
B-9000 Gent, Belgium
hans.vandierendonck@elis.ugent.be

Philippe Manet and Jean-Didier Legat
Université catholique de Louvain
Microelectronics Laboratory
Place du Levant, 3
B-1348 Louvain-la-Neuve, Belgium
{manet,legat}@dice.ucl.ac.be

Abstract

Embedded systems allow application-specific optimizations to improve the power/performance trade-off. In this paper, we show how application-specific hashing of the address can eliminate a large number of conflict misses in caches. We consider XOR-functions: each set index bit is computed as the XOR of a subset of the address bits.

Previous work has considered simpler bit-selecting functions. Compared to such work, the contributions of this paper are two-fold. Firstly, we present a heuristic algorithm to construct application-specific XOR-functions. Secondly, in order to adapt the hashing to the application, we show that a reconfigurable XOR-function selector is inherently less complex than a reconfigurable selector for bit-selecting functions. This is possible by placing restrictions on the allowed XOR-functions.

Our evaluation shows a reduction of cache misses for standard benchmarks averaging between 30% and 60%, depending on the cache size.

1. Introduction

Embedded processors use cache memories to hide the memory latency. However, certain memory access patterns are plagued by a large number of conflict misses, resulting in low performance and high energy consumption. This is most common in direct mapped caches where the cache RAM is accessed using the least significant bits of the memory address. *Hash functions* eliminate conflict misses by hashing the address before accessing the cache. Hashing changes the access pattern and removes patterns with many conflict misses [10, 5, 12, 13].

A bit-selecting hash function selects the set index bits from the address bits. Conflicts may be eliminated by selecting higher-order address bits and leaving low-order bits unselected [1]. A XOR-function is computationally more complex as it computes every set index bit as the exclusive or (XOR) of a subset of the address bits. XOR-functions eliminate conflict misses for stride patterns [9] as well as for complex subsets of matrices [3, 14].

A hash function that minimizes conflict misses for one application does not necessarily perform well for another application, making it beneficial to tune the hash function to the executing application. This requires (i) algorithms to construct the hash function and (ii) dedicated hardware to dynamically reconfigurable the hash function.

In this paper, we present and analyze an algorithm for constructing XOR-functions. It makes use of a clever prediction-scheme that estimates the number of misses incurred by a hash function. This way, we sidestep the time-complexity of simulating a large number of hash functions.

We discuss the hardware implementation of reconfigurable XOR-functions and investigate the trade-off between conflict reduction and complexity of the hash functions. Finally, we determine a subset of XOR-functions with small hardware complexity while maintaining performance.

The remainder of this paper is organized as follows. Section 2 discusses background and related work. The algorithm for constructing XOR-functions is presented and analyzed in Section 3. Section 4 discusses permutation-based hash functions, a particular type of XOR-functions that allow for cheap reconfigurable indexing. The complexity of reconfigurable indexing is discussed in Section 5. Section 6 evaluates the algorithm using embedded benchmarks.

2. Background and Previous Work

XOR-based hash functions are conventionally represented by a $n \times m$ binary matrix [9], where n is the number of hashed address bits and m is the 2-logarithm of the

number of sets in the cache. An n-bit block addresses $\mathbf{a}$ is represented as a bit vector $a_{n-1}a_{n-2}\ldots a_0$, with a_{n-1} the most significant bit and a_0 the least significant bit. The hash function represented by the matrix H maps n address bits to m bits. The bit $h_{r,c}$ on row r and column c is 1 when address bit a_r is an input to the XOR computing the c-th set index bit. The computation of the set index $\mathbf{s}$ can be expressed as the vector-matrix multiplication over $GF(2)$ denoted by $\mathbf{s} = \mathbf{a}H$. Here, $GF(2)$ is the domain $\{0,1\}$ where addition is replaced by XOR and multiplication by logical *and*.

For the purpose of automating the construction of hash functions, it is desirable to transform the matrix to its *null space* [13, 14]. The null space $N(H)$ of a matrix H is the set of all vectors that are mapped to the zero vector:

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

The null space shows what addresses suffer conflict misses: Two addresses $\mathbf{x}$ and $\mathbf{y}$ conflict 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:

$$\mathbf{x}H = \mathbf{y}H \iff (\mathbf{x}\oplus\mathbf{y})H = \mathbf{0} \iff (\mathbf{x}\oplus\mathbf{y}) \in N(H) \quad (2)$$

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. By using the null space representation, we avoid evaluating the same hash function multiple times during the design space exploration. The number of n-to-m hash functions equals

$$N(n,m) = \prod_{i=1}^{m} \frac{2^{n-i+1}-1}{2^i-1}, \qquad 0 \le m \le n \quad (3)$$

There are $3.4e38$ distinct matrices, hashing 16 address bits to 8 set index bits but only $6.3e19$ distinct null spaces.

Several authors have shown the advantages of XOR-indexing caches [5, 12]. Their work draws on the insights gained by Rau [9] who investigated *polynomial hash functions* and their ability to map stride patterns conflict-free.

Seznec and Bodin [2] developed the skewed-associative cache using a different hash function in each cache bank. By using multiple hash functions the miss rate is not strongly dependent on the hash functions.

Algorithms for automatically constructing hash functions have been described in the case of bit-selecting hash functions. Abraham and Agusleo [1] construct hash functions from a list of frequently-occurring strides. Givargis [4] presents a profile-based approach that constructs hash functions for all access patterns. His algorithm is heuristic and sub-optimal. An optimal algorithm is presented by Patel *et al.* [8], who, in fact, developed a smart way to simultaneously simulate all bit-selecting hash functions. This is feasible because the number of bit-selecting functions equals the number of combinations of m out of n. A direct extension of their algorithm to XOR-functions is unfeasible due to the large number of such functions.

```
Let a_i = block address of reference i
Let n = length of vectors
Let misses(v) = accumulated misses for 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 reuse distance(a_i) > cache size 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 misses(v)
        od
        move a_i to top of stack
    fi
od
```

Figure 1. The profiling algorithm.

3. Algorithm for Constructing Optimized XOR-Functions

In order to limit analysis time and to combat the complexities resulting from the design space size, we developed a heuristic algorithm consisting of two phases. First, profiling information is gathered to identify when and how often conflict misses occur. In the second phase, we use the profiling information to efficiently search through the space of hash functions.

3.1. The Profiling Phase

A conflict miss involving cache blocks $\mathbf{x}$ and $\mathbf{y}$ can only occur if the vector $\mathbf{v} = \mathbf{x}\oplus\mathbf{y}$ is a member of the null space $N(H)$ (See Equation 2). In particular, an access to cache block $\mathbf{x}$ is a conflict miss if the program touches a block $\mathbf{y}$ since the previous access to $\mathbf{x}$ that is mapped to the same set as $\mathbf{x}$, i.e., $(\mathbf{x}\oplus\mathbf{y}) \in N(H)$. Thus, it is possible to detect conflict misses by keeping track of all blocks that are accessed between two accesses to the same block. This can be achieved by making a single-pass traversal of the memory access trace of a program.

For every memory access in the program, the accessed cache block is pushed on an LRU stack (Figure 1). An LRU stack has the property that the blocks are sorted with the most recently used block at the top and the least recently used block at the bottom. When an access to block $\mathbf{x}$ occurs in the memory access trace, then we need to know all blocks that have been accessed more recently since the pre-

vious access of $\mathbf{x}$. These blocks are found on the LRU stack above block $\mathbf{x}$. Then, we compute for every intermediately accessed block $\mathbf{y}$, the vector $\mathbf{x} \oplus \mathbf{y}$. If $(\mathbf{x} \oplus \mathbf{y}) \in N(H)$, then we know that a conflict miss occurs for block $\mathbf{x}$. By remembering how many times this vector occurs, we can make an estimate of the number of conflict misses for *any* XOR-function H. Let us assume that $misses(\mathbf{v})$ represents the number of times that vector $\mathbf{v}$ was encountered during analysis of the trace. Then the number of conflict misses that occur for hash function H can be estimated as:

$$misses(H) = \sum_{\mathbf{v} \in N(H)} misses(\mathbf{v}). \quad (4)$$

The quantities $misses(\mathbf{v})$ bear a small dependence on the hash function. It is a good approximation to assume that they are totally independent of the hash function. However, this makes the optimization algorithm as a whole heuristic and may produce sub-optimal results.

To improve the accuracy of the estimates, the profiling algorithm filters out two types of cache misses that are not solvable using XOR-indexing: (i) compulsory misses and (ii) capacity misses. We assume that capacity misses have a reuse distance larger than the cache capacity.

3.2. Searching the Design Space

The design space holding all XOR-functions is searched for the hash function that incurs the fewest cache misses. Search algorithms need to perform many evaluations before they converge on a hash function. The evaluation of a hash function is very fast when *estimating the number of conflict misses* using the technique of the previous subsection.

We use the hill climbing search algorithm (also known as steepest descend). This is an iterative algorithm starting in a fixed or randomly selected point in the design space. We start the algorithm in the null space of the conventional index function. Note that the design space consists of *null spaces* as this significantly reduces search times. The algorithm evaluates all *neighbors* of this index function. In our implementation, two null spaces are neighbors if they differ in exactly one dimension, i.e., the dimensionality of their intersection equals their own dimensionality minus 1. If the neighbor with the fewest conflict misses incurs fewer conflicts than the best null space found so far, then the algorithm moves to that null space and iteratively investigates all its neighbors. Otherwise, a local optimum has been found and the algorithm stops.

When limiting the number of inputs per XOR-gate, constructing permutation-based XOR-functions or bit-selecting functions, we run the search algorithm in exactly the same way as for general XOR-functions. At the end, the algorithm outputs the best function satisfying the criteria.

This algorithm constructs a hash function in 0.5 to 10 seconds on a 2 GHz Pentium 4, depending on the dimensions of the function (rows and columns) and on the profiling information.

3.3. Optimality

The presented algorithm is sub-optimal. The profiling phase is not exact, leading to incorrectly estimating the conflict misses. The search phase can only investigate a fraction of the design space. It is likely that both phases of the algorithm can be improved, at the expense of execution speed. However, we can proof [13] that it is impossible to devise a profiling phase that computes correct profiling information, i.e., Equation 4 allows one to correctly estimate the number of conflict misses for all XOR-functions. In order to improve the accuracy of the profiling phase, it is necessary to change the structure and/or the amount of profiling information.

4. Permutation-Based Hash Functions

Permutation-based hash functions are a special type of XOR-functions. They map every aligned run of 2^m consecutive cache blocks conflict-free. Thus, the mapping of every run of 2^m blocks to their set indices is a permutation of $0, \ldots, 2^m - 1$, hence the name. Permutation-based hash functions satisfy the following property:

$$N(H) \cap span(\mathbf{e}_0, \ldots, \mathbf{e}_{m-1}) = \{\mathbf{0}\} \quad (5)$$

where $\mathbf{e}_k$ is the vector of all zeroes except for a 1 at the k-th position and the *span* is the smallest vector space containing all of its arguments. An equivalent definition is to state that permutation-based hash functions have a matrix representation where the m low-order rows are all zeroes except for a diagonal of ones, i.e., row i equals $\mathbf{e}_i$ for $0 \leq i < m$.

When hashing the set index, care should be taken to properly compute the tag. The tag function and the set index function together should be bijective, i.e., two distinct addresses should either have a different tag or a different set index. It can be shown that, in the case of permutation-based functions, the tag can be computed by selecting the high-order $n - m$ address bits. This is the same tag computation as for the conventional modulo indexing. Indeed, the null space of the conventional tag function is $N(T) = span(\mathbf{e}_0, \ldots, \mathbf{e}_{m-1})$. For $\mathbf{x} \neq \mathbf{y}$ it follows that $\mathbf{x}H = \mathbf{y}H$ cannot occur at the same time as $\mathbf{x}T = \mathbf{y}T$ because $N(H) \cap N(T) = \{\mathbf{0}\}$ (Equation 5).

Non-permutation-based functions need a different tag function. We state without proof that the tag function is a bit-selecting function in all cases. Bit-selecting functions

Table 1. Number of switches required for reconfigurable indexing with $n = 16$ and 4-byte cache blocks.

cache size	1 KB	4 KB	16 KB
set index bits (m)	8	10	12
bit-select	256	256	256
optimized bit-select	144	136	112
general XOR	252	261	250
permutation-based	72	70	60

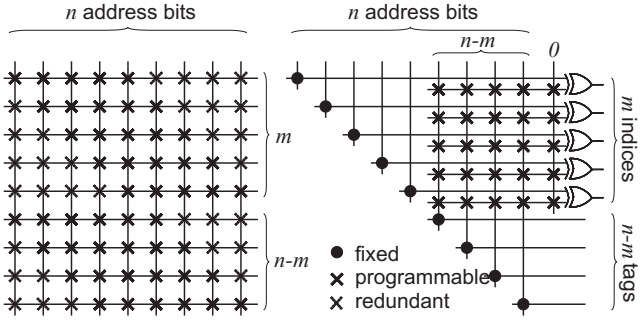

Figure 2. Reconfigurable selection networks.

are not permutation-based (except for the traditional modulo 2^m function). The tag function selects the address bits not included in the set index.

5. Reconfigurable XOR-Indexing

Reconfigurable XOR-indexing is necessary to dynamically adapt to changes between applications. Patel *et al.* [8] propose circuits for reconfigurable address bit selectors. In this paper, we analyze their complexity in terms of the type of function (bit-selecting vs. XOR-function), the inputs per XOR, the number of hashed address bits (n) and set index bits (m). There may be substantially fewer hashed address bits than the total address bits (N). In this case, the $N-n$ high-order address bits are only used to compute the tag.

For bit-selecting functions, every set index bit and every tag bit is selected among any of the n address bits. In total n 1-out-of-n selectors are needed. This solution is not optimal: with a given selection pattern of m bits, all permutations of those bits will lead to an equivalent configuration. As shown in Figure 2(a), shaded connections are redundant, reducing the complexity to m 1-out-of-$(n-m+1)$ selectors for the set index bits and $n-m$ 1-out-of-$(m+1)$ selectors for the tag bits. Some practical values are listed in Table 1.

For XOR-functions, 2-input XOR-gates are sufficient to achieve a low miss rate. Hereby, $n+m$ selectors are necessary. The optimization of the previous paragraph is applicable to the first XOR input and the tag bits. For the second input, a constant input must be allowed such that a bit can be selected instead of hashed. In total, $(n+1)m - m(m-1)/2$ switches are required.

It was noted in Section 2 that permutation-based hash functions have a matrix representation where the low-order m rows are zero except for the diagonal, which contains ones. Thus, the first input of the XOR can be fixed to one of the m low-order address bits. The second input to the XOR is selected from among the $n-m$ high-order address bits. The tag is also fixed and equals the high-order $N-m$ address bits. A 2-input permutation-based function can therefore be implemented using m 1-out-of-$(n-m+1)$ selectors followed by a 2-input XOR-gate (Figure 2(b)).

The selectors are implemented with a pass gate and a memory cell per switch. To reduce complexity, pass gates can be simplified [8]. Nevertheless, the selectors require many wires. These wires lead to very high capacitance nodes, making them slow and power consuming. Bit-selecting functions require n lines crossed by n. However, permutation-based XOR-functions require only $n-m$ lines crossed by m.

XOR-gates based on pass transistors are fast and compact. The first input of the XOR comes directly from the address register, allowing us to take both the address and its complement from the flip-flop. Consequently, 2 pass-gates but only one invertor are needed per XOR-gate. Putting the selector and the XOR-gates together shows that *a reconfigurable 2-input XOR-function requires less gates and interconnections than a reconfigurable bit-selecting function*.

6. Experimental Evaluation

We apply the optimized XOR-function algorithm to embedded benchmarks from the MediaBench [7] and MiBench [6] suites. The benchmarks are run with large data sets when available. The benchmarks are compiled for the SA-110 ARM processor by the ARM C compiler using optimization level 2. They are simulated using the PowerAnalyzer simulator (`http://www.eecs.umich.edu/~panalyzer/`).

We applied the algorithm to determine optimized XOR-functions for three cache sizes: 1, 4 and 16 KB. All caches are direct mapped and have 4-byte cache blocks. The number of hashed address bits $n = 16$. Different hash functions are generated per benchmark.

The first experiment determines the impact on the miss rate of using permutation-based XOR-functions versus general XOR-functions. We do not present full results due to space limitations. On average, general XOR-functions reduce the data cache miss rate by 34.6% (1 KB), 44.0% (4 KB) and 26.9% (16 KB), compared to conventional bit-selecting indexing. Permutation-based functions eliminate approximately the same fraction of misses: 32.3% (1 KB), 43.9% (4 KB) and 26.7% (16 KB). Thus, limiting the design

Table 2. Baseline misses/K-uop and percentage of cache misses removed with optimized XOR-functions in data caches and instruction caches.

	benchmark	1 KB cache				4 KB cache				16 KB cache			
		base	2-in	4-in	16-in	base	2-in	4-in	16-in	base	2-in	4-in	16-in
data caches	dijkstra	69.0	21.2	21.3	22.1	36.6	17.2	15.0	17.9	20.0	1.3	1.5	1.4
	fft	10.5	69.4	79.8	81.9	4.1	76.4	76.5	76.4	1.0	12.7	12.7	12.7
	jpeg_enc	61.9	42.7	44.4	45.6	33.9	36.6	36.8	36.8	15.9	15.2	15.5	15.5
	jpeg_dec	104.9	29.3	31.8	33.0	39.1	21.3	27.2	27.4	21.5	52.2	57.7	57.3
	lame	16.4	4.4	10.0	10.3	9.9	5.6	6.0	5.9	6.1	45.2	47.1	47.2
	rijndael	141.3	-2.6	-3.3	-2.7	48.1	4.6	5.8	5.8	20.7	100.0	100.0	100.0
	susan	19.7	16.9	17.0	17.1	11.6	22.4	23.1	23.2	6.9	7.1	7.7	7.7
	adpcm_dec	24.6	29.5	37.3	37.8	1.0	87.8	91.9	91.9	0.1	0.0	0.0	0.0
	adpcm_enc	32.4	51.3	51.7	51.7	0.5	90.8	90.8	90.8	0.0	0.0	0.0	0.0
	mpeg2_dec	10.3	38.9	49.3	49.6	2.9	61.0	63.4	63.9	0.7	25.1	27.7	27.7
	average	18.9	30.1	33.9	34.6	10.4	42.3	43.6	44.0	6.0	25.9	27.0	26.9
instruction caches	dijkstra	13.2	36.6	51.1	51.0	1.0	62.0	62.1	62.1	0.0	0.0	0.0	0.0
	fft	294.5	8.5	9.6	10.6	87.7	17.3	27.0	27.0	5.4	72.9	72.9	72.9
	jpeg_enc	37.5	50.5	55.5	55.5	1.0	44.8	61.2	62.4	0.4	80.0	85.8	85.8
	jpeg_dec	37.4	19.6	24.6	27.0	5.4	43.9	80.8	80.8	3.0	92.6	92.7	92.7
	lame	125.1	3.9	13.1	15.2	21.3	23.7	45.8	45.8	3.6	73.7	84.5	84.5
	rijndael	624.4	0.0	0.0	0.0	167.7	0.1	-0.1	-0.1	128.2	100.0	100.0	100.0
	susan	255.2	7.9	13.3	13.5	54.3	56.4	69.3	69.3	0.2	76.2	76.2	76.2
	adpcm_dec	31.3	1.8	13.6	13.6	0.8	76.8	83.1	83.1	0.1	0.0	0.0	0.0
	adpcm_enc	70.4	16.5	22.5	19.1	0.6	61.0	87.3	87.3	0.1	0.0	0.0	0.0
	mpeg2_dec	128.7	55.5	58.7	68.7	15.4	92.1	92.5	93.1	0.4	80.2	83.5	83.5
	average	143.6	20.1	26.2	27.4	27.7	47.8	60.9	61.1	5.6	57.5	59.6	59.6

space to permutation-based functions does not significantly limit performance.

In the second experiment, we determine the impact of the number of inputs per XOR for permutation-based functions. We run the optimization algorithm three times for every cache configuration, each time generating hash functions with a different degree of complexity. We consider data caches and instruction caches (Table 2). We show the misses per K-uop (base) and the percentage of misses removed with permutation-based XOR-functions with at most 2 inputs (2-in), 4 inputs (4-in) or without a limitation on the inputs (16-in).

The optimized 2-input XOR-functions eliminate 30%, 42% or 26% of cache misses, depending on cache size (Table 2). Allowing more inputs for the XOR-gates allows more powerful functions, but increases circuit-complexity. This additional strength is, on average, marginally useful and eliminates only a few additional percents of cache misses. This benefit diminishes with increasing cache size.

The same trends hold for the instruction caches (Table 2). The miss rate reduction is much larger in instruction caches than in data caches, showing averages around 50% (4 KB cache) and 60% (16 KB cache). Sometimes virtually all cache misses are eliminated.

The constructed hash functions introduce additional cache misses in a few situations. The percentage of additional misses is, in every case, very small and results from the heuristic nature of the algorithm. If needed, this adverse effect can be tested for and one can revert to the conventional index function if it occurs.

6.1. Discussion of Optimality

We test optimality by constructing bit-selecting functions and comparing the presented algorithm to the optimal algorithm [8]. Because the optimal algorithm is very slow, we have only been able to gather results for the short PowerStone benchmarks [11]. We report results only for the 4 KB data cache (Table 3); other cache configurations yield similar results. Comparing the optimal bit-selecting functions (column 'opt') to the heuristically generated bit-selecting functions (column '1-in') shows that optimal results are obtained for all but 3 benchmarks (bcnt, blit and compress).

The permutation-based XOR-functions out-perform the optimal bit-selecting functions. The 2-input XOR-functions eliminate 11.7% of cache misses, while the optimal bit-selecting functions eliminate 8.5%, implying that certain frequently-occurring memory access patterns can be mapped conflict-free by XOR-functions but not by bit-selecting functions.

Table 3. Percentage of misses removed by XOR- and optimal bit-selecting functions.

bench	opt	1-in	2-in	4-in	16-in	FA
adpcm	0.0	0.0	0.2	0.2	0.2	0.2
bcnt	5.2	0.0	0.0	0.0	0.0	0.0
blit	14.7	8.6	14.3	14.3	14.3	0.0
compress	3.2	3.0	2.4	2.8	2.9	2.7
crc	0.0	0.0	0.0	0.0	0.0	0.0
des	0.0	0.0	8.8	8.6	10.1	17.8
engine	36.2	36.2	36.2	36.2	36.2	36.2
fir	7.7	7.7	7.7	7.7	7.7	7.7
g3fax	0.0	0.0	37.1	41.1	41.1	57.0
jpeg	2.3	2.3	1.4	1.6	1.6	7.2
pocsag	3.0	3.0	3.0	3.0	3.0	3.0
qurt	0.0	0.0	0.0	0.0	0.0	0.0
ucbqsort	46.6	46.6	46.6	46.6	46.6	46.6
v42	0.0	0.0	5.6	6.2	6.0	18.0
average	8.5	7.7	11.7	12.0	12.1	14.0

We expect that there is room to develop algorithms for optimal XOR-functions, because the XOR-functions are sub-optimal (see bcnt, blit and compress) and hashing may out-perform full associativity (column 'FA') due to sub-optimality of the LRU replacement policy.

7. Conclusion

We presented an algorithm to construct XOR-functions that eliminate cache conflict misses. The algorithm is heuristic and profiles the cache accesses made by a program. Based on this profiling information, the best suitable XOR-function is computed.

The optimal XOR-function depends on the executed program, so it requires a reconfigurable XOR-function. To restrict complexity, we allow only *permutation-based* functions with 2-input XOR-gates: Such reconfigurable XOR-functions require less gates and interconnections than reconfigurable bit-selecting functions. Furthermore, they yield lower miss rates.

The algorithm is applied to generate XOR-functions for instruction and data caches, yielding average reductions of 30% to 60% of the miss rate for standard benchmarks. We show that the presented algorithm computes optimal bit-selecting functions for all but 3 benchmarks. Algorithms for optimal XOR-functions are not known, but our analysis suggests that there is potential room for improvement.

Acknowledgments

This research is sponsored in part by the Fund for Scientific Research-Flanders (FWO), the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) and Ghent University. Philippe Manet is funded by the Walloon region.

References

[1] S. G. Abraham and H. Agusleo. Reduction of cache interference misses through selective bit-permutation mapping. Technical Report CSE-TR-205-94, The University of Michigan, 1994.

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

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

[4] T. Givargis. Improved indexing for cache miss reduction in embedded systems. In *Design Automation Conference*, 2003.

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

[6] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *4th Annual Workshop on Workload Characterization*, Dec. 2001.

[7] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Conference on Microprogramming and Microarchitecture*, pages 330–335, Dec. 1997.

[8] K. Patel, L. Benini, P. Macii, and M. Poncino. Reducing cache misses by application-specific re-configurable indexing. In *International Conference on Computer-Aided Design*, pages 125–130, Nov. 2004.

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

[10] 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.

[11] J. Scott, L. Lee, J. Arends, and B. Moyer. Designing the low-power M Core architecture. In *Proceedings of the IEEE Power Driven Microarchitecture Workshop*, pages 145–150, June 1998.

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

[13] H. Vandierendonck. *Avoiding Mapping Conflicts in Microprocessors*. PhD thesis, Ghent University, 2004.

[14] H. Vandierendonck and K. De Bosschere. XOR-based hash functions. *IEEE Transactions on Computers*, 54(7):800–812, Sept. 2005.