Constructing Optimal XOR-Functions to Minimize Cache Conflict Misses

Hans Vandierendonck and Koen De Bosschere

Dept. of Electronics and Information Systems (ELIS)/HiPEAC, Ghent University,
St.-Pietersnieuwstraat 41, B-9000 Gent, Belgium,
{hans.vandierendonck,koen.de.bosschere}@elis.ugent.be

Abstract. Stringent power and performance constraints, coupled with detailed knowledge of the target applications of a processor, allows for application-specific processor optimizations. It has been shown that application-specific reconfigurable hash functions eliminate a large number of cache conflict misses. These hash functions minimize conflicts by modifying the mapping of cache blocks to cache sets.
This paper describes an algorithm to compute optimal XOR-functions, a particular type of hash functions based on XORs. Using this algorithm, we set an upper bound on the conflict reduction achievable with XOR-functions. We show that XOR-functions perform better than other reconfigurable hash functions studied in the literature such as bit-selecting functions.
The XOR-functions are optimal for one particular execution of a program. However, we show that optimal XOR-functions are less sensitive to the characteristics of the execution than optimal bit-selecting hash functions. This again underlines that XOR-functions are the best known hash functions to implement reconfigurable hash functions.

1 Introduction

The design of embedded systems is strongly dominated by power and performance constraints. To maximize performance and minize power to the fullest extent, it is necessary to apply application-specific optimizations to the processor. The optimizations that are applied depend on the executing program: the optimizations act differently for different programs. In the context of caches, attention has been drawn to application-specific reconfigurable hash functions to minimize conflict misses.

The literature focusses on two types of reconfigurable hash functions: bit-selecting functions and XOR-functions. Bit-selecting functions determine each set index bit by selecting one of the address bits. XOR-functions involve some computation: each set index bit is the XOR of a subset of the address bits. The conventional modulo-power-of-2 indexing belongs to both categories.

A particular sub-class of XOR-functions, namely permutation-based functions where the XOR is performed on at most 2 address bits, are particularly interesting to implement reconfigurable hash functions in hardware, yielding circuitry that is less complex than that of reconfigurable bit-selecting functions [1].

Although XOR-functions require less hardware support than bit-selecting functions, the problem remains of constructing hash functions that minimize the number of conflict misses. A heuristic algorithm for constructing XOR-functions is described in [1, 2]. It is, however, very hard to state how good a heuristic algorithm actually is if it is not known what the optimum is. To close this gap, we present an optimal algorithm for computing XOR-functions. With this algorithm, we further reduce the number of conflict misses and we can evaluate how near to optimal the heuristic algorithm is.

The optimal XOR-function is optimal with respect to one particular execution of a program by nature of the algorithm. The same property is true for the heuristic algorithm: the estimates are valid only for a single run of a program. In practice, however, the hash functions should work properly across multiple program executions. We show that optimality is lost due to changing the program's input data set, but the net conflict reduction of applying a hash function remains impressive. Furthermore, the optimal XOR-function removes more conflicts than a heuristically constructed XOR-function, regardless of the input data set.

The remainder of this paper is organized as follows. Section 2 describes related work. The optimal algorithm is explained and illustrated in Section 3. It is evaluated in Section 4 and the paper concludes with Section 5.

2 Background and Related Work

XOR-functions are generally represented by a matrix [3, 4]. When n address bits are mapped onto m set index bits ($m < n$), then the XOR-function is a $n \times m$ binary matrix. The bit in column c and row r is 1 if the r-th address bit is included in the XOR computing the c-th set index bit.

Permutation-based functions are a subset of the XOR-functions that obey spatial locality [4]. Their matrix representation is characterized by a diagonal of ones, and otherwise zeroes, in the lower m rows. In a 12×6 permutation-based function, only the cells with a dot can be chosen freely:

$$H = \begin{bmatrix} 0&0&0&0&0&1 \\ 0&0&0&0&1&0 \\ 0&0&0&1&0&0 \\ 0&0&1&0&0&0 \\ 0&1&0&0&0&0 \\ 1&0&0&0&0&0 \\ \cdot&\cdot&\cdot&\cdot&\cdot&\cdot \\ \cdot&\cdot&\cdot&\cdot&\cdot&\cdot \\ \cdot&\cdot&\cdot&\cdot&\cdot&\cdot \\ \cdot&\cdot&\cdot&\cdot&\cdot&\cdot \\ \cdot&\cdot&\cdot&\cdot&\cdot&\cdot \\ \cdot&\cdot&\cdot&\cdot&\cdot&\cdot \end{bmatrix}. \tag{1}$$

Note that the rows are numbered from the top down (the top row involves address bit 0) and the columns are numbered from right to left.

Permutation-based functions are non-overlapping with bit-selecting functions, except for the conventional modulo-power-of-2 indexing that belongs to both classes of functions.

Reconfigurable hash functions are implemented in hardware by adding a small amount of configuration memory and a multiplexer to every address line of the cache memory [5, 1]. The multiplexer selects one out of a set of address bits. The select input of the multiplexer is stored in the configuration memory. As the configuration remains fixed during long periods of time, the multiplexer introduces marginable latency [5].

Reconfigurable permutation-based functions can be implemented with less hardware than reconfigurable bit-selecting functions, provided that the number of inputs per XOR is restricted to 2 [1]. This implies that, in the example above (Equation 1), only one of the dots in a single column can be 1. Note that allowing more inputs per XOR-gate allows small additional reductions of the conflict misses, but the complexity of the hardware implementation of the reconfigurable XOR-functions increases strongly [1].

Several algorithms have been described to optimize hash functions to a particular program execution. Most of these are profile-based approaches that analyze reuse edges in the memory access stream [6, 5, 1]. Another algorithm analyzes only the strides [7]. The algorithm of Patel *et al* [5] computes an optimal bit-selecting function. It is a guide to our algorithm for optimal XOR-functions.

3 Optimal Algorithm for XOR-Functions

We present an algorithm for computing optimal XOR-functions. The algorithm achieves simultaneous simulation of all hash functions in a given class by cleverly encoding the conflict miss count for each hash function in a symbolic expression. The algorithm combines the reuse-edge analysis of [1] with the symbolic modeling of conflict miss counts of [5].

3.1 Definitions

The execution of a program is represented by a trace of memory accesses. The trace length is denoted N. Each memory access refers to an address. Addresses are denoted in bold, e.g. $\mathbf{a}$. The i-th address in the trace is $\mathbf{a}_i$ for $0 \leq i < N$.

3.2 Data Reuse and the Structure of the Algorithm

Caches exploit temporal locality [8]: each cache block is typically used multiple times during a given time span. This *reuse* makes temporarily storing the cache block in a local memory efficient.

Reuse behavior in a memory access trace can be analyzed using *reuse edges* [9, 10]. The reuse edge points from one access to a cache block to the previous access in the trace relating to the same cache block. The position of this access in the trace is denoted $r(i)$, so there is an edge from i to $r(i)$. Note that $r(i)$ is undefined

if the i-th address in the trace is the first access to a cache block. The actual reuse of the block depends on the cache blocks accessed in the portion of the trace that is covered by the reuse edge. If any of these cache blocks maps to the same set of the cache as the current cache block, then it displaces the current cache block from the cache.

These observations lead to the following structure of an algorithm for constructing hash functions [1, 2]. The algorithm analyzes the memory access stream one memory access at a time. If the currently analyzed cache block has not been accessed before (it is not possible to construct a reuse edge), then a compulsory miss occurs [8].

If a reuse edge exists, it marks the set of cache blocks that can intervene in the reuse of the current block: If any of these cache blocks map to the same cache set as the current block, then a conflict miss occurs for the current block. This rule can be used to decide if a conflict miss occurs for a particular hash function. The optimal algorithm uses this rule to compute the set of hash functions where a conflict miss occurs. In principle, one can iterate over all hash functions and over all blocks spanned by the reuse edge to compute this set of hash functions. A more appropriate solution is obtained by symbolically modeling the set of hash functions incuring a conflict miss. This symbolic model is built on the *miss conditions*.

3.3 Recapitulation of Miss Conditions for Bit-Selecting Functions

The set of hash functions incurring a conflict miss for particular reuse edge is denoted by a symbolic expression. The symbolic expression contains unknowns describing the hash function. In the case of bit selecting functions, the unknowns describe the bits that are selected by the hash function. For an n-bit address, there are n boolean variables $y_0, \ldots, y_{n-1}$. Variable y_i is true when address bit i is selected by the hash function.

The symbolic expression for the set of hash functions incuring a conflict miss is built in two steps. First, we define the *direct conflict pattern DCP* [5]. $DCP_{i,j}$ expresses whether the addresses $\mathbf{a}_i$ and $\mathbf{a}_j$ are mapped to the same set index:

$$DCP_{i,j} = \bigwedge_{k=0}^{n-1} (\gamma_k \; y_k)'$$

where $\gamma_k = 1$ when the k-th address bits of $\mathbf{a}_i$ and $\mathbf{a}_j$ are equal. The rationale is that, for a conflict miss to occur, the set indices of $\mathbf{a}_i$ and $\mathbf{a}_j$ must be equal. This happens when the hash function selects only address bits that are equal in $\mathbf{a}_i$ and $\mathbf{a}_j$.

The total conflict pattern CP_i accumulates the occurence of a conflict miss across all accessed blocks spanned by a reuse edge. A conflict miss occurs for the reuse edge if one or more of the accessed blocks cause a direct conflict:

$$CP_i = \bigvee_{j=r(i)+1}^{i} DCP_{i,j}$$

where $r(i)$ is the position in the trace of the previous use of the block accessed at position i in the trace.

A total conflict miss count (CMC) is computed for the whole trace. The CMC evaluates to the total number of conflict misses for any hash function:

$$CMC = \sum_{i=0}^{N} CP_i$$

The boolean values 0 and 1 in CP_i are reinterpreted as arithmetic 0 and 1 in the equation above.

3.4 Miss Conditions for Permutation-Based Functions

Hash functions have different degrees of freedom, so they are described using a different set of variables. The degrees of freedom for an n-to-m permutation-based hash function are the unknown bits in the matrix representation (Equation 1), so there are $m(n-m)$ boolean variables $y_{i,j}$ with $0 \leq i < m$ and $m \leq j < n$. Variable $y_{i,j}$ is 1 if there is a 1-bit on row i and column j in the matrix model of the XOR-functions (Equation 1). It is 0 otherwise. Note however that we need an additional restriction on the variables $y_{i,j}$ variables: at most one of the variables $y_{i,j}$ for $m \leq j < n$ and i fixed can be one, as we allow only 2-input XORs.

The DCP is a little more complex in the case of permutation-based hash functions. Let us consider only the k-th set index bit for the moment. The k-th set index bit is the XOR of address bit k with one of the address bits $m, \ldots, n-1$, or none of these bits. The k-th set index bits of $\mathbf{a}_i$ and $\mathbf{a}_j$ differ when one of the address bits k and the selected bit differ, but not both (property of the XOR). The extra bit differs between $\mathbf{a}_i$ and $\mathbf{a}_j$ when $\bigvee_{l=m}^{n-1} \gamma_l \; y_{k,l} = 1$, so we can write the DCP for permutation-based functions:

$$DCP_{i,j} = \bigwedge_{k=0}^{m-1} \left(\gamma_k \oplus \left(\bigvee_{l=m}^{n-1} \gamma_l \; y_{k,l} \right) \right)' \tag{2}$$

3.5 BDD and ADD Data Structures

As in [5] we represent the direct conflict pattern (DCP) and total conflict patterns (CP) using binary decision diagrams (BDD) [11]. The conflict miss count (CMC) evaluates to an integer and is represented by an arithmetic decision diagram (ADD) [12].

The BDDs and ADDs turn out as relatively simple symbolic expressions in the case of bit-selecting functions. Therefore, a straightforward encoding into BDDs is used: the BDD is a binary tree, where an internal node at any level of the BDD tests the value of one of the boolean unknowns and branches two-ways. Each terminal node corresponds to a hash function. The path from the root of

```
Let n = length of vectors
Let a_i = block address of reference i
Let CMC = conflict miss count, initially all
zeroes

for each reference i in program trace do
    if a_i was accessed before then
        CP_i = 0
        for each a_j on stack above a_i do
            compute DCP_{i,j}
            CP_i = CP_i ∨ DCP_{i,j}
        od
        move a_i to top of stack
        CMC = CMC + CP_i
    else    /* compulsory miss */
        push a_i on stack
    fi
od
```

Fig. 1. The profiling algorithm.

the BDD or ADD to the terminal node tells what bits are selected by the hash function by the y_i variables that are 1.

The BDDs and ADDs for XOR-functions are more complex and BDD/ADD computation time dominates the algorithm. We optimize the size of the BDD using the constraint that at most one of the variables $y_{i,j}$ is 1, where $m \leq j < n$ and i fixed.

The BDD/ADD is an m-level tree where each level of the tree corresponds to one of the columns of the hash function. The BDD branches $(n-m+1)$-ways at each level. $(n-m)$ branches correspond to XORing the fixed address bits with one of the address bits $m, \ldots, n-1$, in which case exactly one of the $y_{i,j}$ variables is 1, with i equal to the level in the tree. The $(n-m+1)$-th branch corresponds to the case where the fixed bit is not XORed with any other address bits: all $y_{i,j}$ variables are 0.

The structure of the BDD and ADD is illustrated in Figure 2. Here, $n = 4$ address bits are hashed into $m = 2$ bits. The top level of the tree selects between the possible cases for the hashed bit 0 (column 0 in the matrix representation). The possible cases are: (i) XOR address bit 0 with address bit 2 ($y_{0,2} = 1$), (ii) XOR address bit 0 with address bit 3 ($y_{0,3} = 1$) or (iii) do not XOR address bit 0 with any other address bit ($y_{0,2} = 0$ and $y_{0,3} = 0$).

3.6 The Algorithm

The algorithm is presented in Figure 1. Some optimizations are useful to speedup the algorithm.

The total conflict pattern is computed over the list of blocks that are spanned by a reuse edge. It is not uncommon that this list contains many duplicates.

Table 1. Computation of DCPs and CPs on an example trace.

Conflict pair	DCP	CP
1100-0100	$y'_{0,3}y'_{1,3}$	$y'_{0,3}y'_{1,3}$
0100-1101	$y_{0,3}y'_{1,3}$	
0100-1100	$y'_{0,3}y'_{1,3}$	$y'_{1,3}$
0100-1001	$(y_{0,2}+y_{0,3})y'_{1,2}y'_{1,3}$	$(y_{0,2}+y_{0,3})y'_{1,2}y'_{1,3}$
1100-0100	$y'_{0,3}y'_{1,3}$	
1100-1001	$y_{0,2}y'_{1,2}$	
1100-1101	0	$y_{0,2}y'_{1,2}+y'_{0,3}y'_{1,3}$

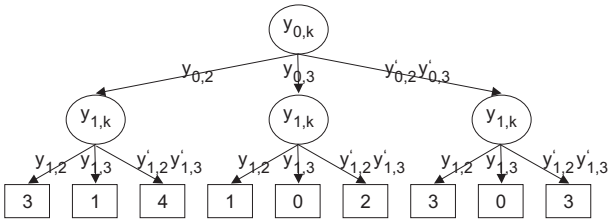

Fig. 2. The ADD computed for the example.

Constructing the DCP for each of these duplicates is a waste of time, as the total conflict pattern ORs all these DCPs.

Filtering out duplicates from the list of blocks is straightforward when using a stack. The stack is ordered such that more recently used blocks are placed above less recently used blocks.

The DCP degrades to a special case if it is computed for a pair of addresses where only the m low-order address bits differ, In this case, XORing the low-order address bits with high-order bits cannot solve the conflict, because the operation performed on the low-order address bits (toggle or not toggle) is the same for both addresses. This can also be seen by setting all $\gamma_l = 0$ with $m \leq l < n$ in Equation 2.

3.7 Example

We present a simple example to illustrate the analysis of conflict misses and the construction of the ADD. We assume that 4 address bits are mapped to 2 set index bits ($n = 4$, $m = 2$). The ADD has 2 levels ($m = 2$). Each node branches 3-ways ($n-m+1 = 3$). Furthermore, we assume that a program accesses following block addresses: 12, 4, 12, 13, 4, 9, 4, 12. The computation of the DCPs and CPs is illustrated in Table 1.

The CMC for this example is shown in Figure 2. There are two optimal hash functions in the example, achieving zero conflict misses. These hash functions

can be constructed by tracing the path from the root node of the ADD to the terminal node. They are:

$$H_0 = \begin{bmatrix} 0\,1 \\ 1\,0 \\ 0\,0 \\ 1\,1 \end{bmatrix} \qquad H_1 = \begin{bmatrix} 0\,1 \\ 1\,0 \\ 0\,0 \\ 1\,0 \end{bmatrix}$$

4 Experimental Evaluation

We characterize the algorithm for optimal XOR-functions using benchmarks from the PowerStone [13], MediaBench [14] and MiBench [15] suites. Many of the PowerStone benchmarks are trivial. We use them to make a comparison to prior work [5, 6]. The other 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 compute hash functions for demand-fetching data caches with sizes of 1 KB, 4 KB and 16 KB. The cache block size is 32 bytes. All caches are direct mapped.

Optimal bit-selecting functions are computed using the algorithm of Patel *et al* [5]. Heuristic XOR-functions are computed from a profile that is measured as in [1]. Then, a XOR-function is proposed by randomly generating a large number of XOR-functions and estimating their performance using the profile information. The number of randomly generated XOR-functions is determined emperically and it was found that 10000 functions is sufficient to obtain high-quality XOR-functions [2].

4.1 Minimizing Conflict Misses

XOR-functions obtained using heuristics remove more conflict misses than bit-selecting functions [1]. We evaluate first if optimally determined XOR-functions provided an added benefit over heuristic XOR-functions. Table 2 shows the misses per kilo-uop incurred in the different cache configurations, averaged over the PowerStone benchmarks. Optimal bit-selecting functions incur fewer misses

Table 2. Misses per kilo-uop averaged over the PowerStone benchmarks.

Hash	1 KB	4 KB	16 KB
modulo	21.32	8.63	2.03
bit-select	18.17	7.64	1.88
heur. XOR	16.35	6.87	1.78
opt. XOR	16.21	6.79	1.76

Table 3. Misses per kilo-uop for the 4 KB cache and the PowerStone benchmarks.

Bench-mark	modulo index	bit-select	heur. XOR	opt. XOR
adpcm	4.07	4.07	1.40	1.40
bcnt	79.80	17.30	17.30	17.30
blit	202.74	27.91	27.91	25.28
compress	14.77	14.77	11.62	11.60
crc	1.70	1.62	1.62	1.62
des	10.60	10.60	8.97	8.73
engine	3.87	0.03	0.03	0.03
fir	0.20	0.16	0.16	0.16
g3fax	2.24	2.24	0.68	0.68
jpeg	4.56	4.56	4.02	4.01
pocsag	1.44	0.96	0.96	0.96
qurt	1.88	1.88	1.88	1.88
ucbqsort	6.63	0.26	0.26	0.26
v42	17.94	17.74	16.73	16.50
average	8.63	7.64	6.87	6.79

than conventional modulo indexing, heuristic XOR-functions incur still fewer misses and optimal XOR-functions incur the fewest misses of all. This behavior is consistent across all cache configurations, although the difference between the hash functions diminishes quickly with increasing cache size.

The relative performance between the hash functions is valid too for every single benchmark. Table 3 shows the per-benchmark misses per kilo-uop for the 4 KB cache. The difference between two hashing functions can be large, i.e., it succeeds in removing conflict misses or it does not succeed. E.g. the optimal bit-selecting function for g3fax is modulo indexing incurring 2.24 misses per kilo-uop. On the other hand, a heuristically found XOR-function incurs only 0.68 misses per kilo-uop, which is also the optimum. Another example involves bcnt, where any type of hash function performs well.

4.2 Impact of Cross-Profiling

The hash functions are determined for one particular run of the benchmark, using one particular input data set. In practice, applications execute many different data sets, which may lead to a shift in the optimal hash function. We analyze the impact of the input data set using cross-profiling, i.e. we determine the hash functions using a different (smaller) data set than the data set used to present results.

Table 4 shows the misses per kilo-uop for the 4 KB cache and media benchmarks. When considering only the self-profile, we can draw the same conclusions as for the PowerStone benchmarks. The input data set, however, has a large impact on the conflict-avoidance properties of the hash function. The relative

Table 4. Misses per kilo-uop for the 4 KB cache and the media benchmarks.

Bench-mark	modulo index	Self-profile bit-select	Self-profile heur. XOR	Self-profile opt. XOR	Cross-profile bit-select	Cross-profile heur. XOR	Cross-profile opt. XOR
susan	6.78	4.60	4.53	4.46	8.09	4.84	4.78
jpeg enc	20.12	11.94	7.03	6.95	20.12	7.67	7.42
jpeg dec	16.53	16.27	12.38	12.17	23.88	16.87	17.23
adpcm dec	1.49	1.49	0.05	0.05	1.49	0.05	0.05
adpcm enc	0.86	0.86	0.03	0.03	0.86	0.03	0.03
epic dec	4.47	4.29	3.31	3.29	4.64	3.85	3.41
mb/jpeg dec	20.63	16.89	12.76	12.25	22.16	21.03	16.95
average	13.24	9.31	6.53	6.40	14.39	8.25	7.65

performance of the hash functions remains unaltered, underlining again the performance of XOR-functions.

The misses per kilo-uop become significantly higher when applying cross-profiling. The misses raise by 1.25 and 1.72 per kilo-uop on average for optimal and heuristic XOR-functions, respectively. The misses raise by 5.08 misses per kilo-uop for optimal bit-selecting functions, which is worse than the baseline modulo index.

XOR-functions prove to be much more resilient to changing the input data set of a program than optimal bit-selecting functions. This is not surprising since XOR-functions can be applied without knowledge of the running program too [3, 16]. They eliminate conflict misses by randomizing the accesses to the cache, which works for all programs with a high conflict miss rate.

4.3 Performance Improvement

We compute the performance improvement of the hash functions on a processor model that ressembles the XScale processor. Our processor model is an inorder-issue processor that fetches 1 instruction per cycle and can issue up to 2 instructions per cycle. The branch predictor is a 128-entry bimodal branch predictor. The memory hierarchy consists of 4 KB level-1 caches backed by memory with a 32-cycle memory access time. We assuem the same cache access latency in all configurations.

Figure 3 shows the IPC improvement of this processor when different hash functions are applied relative to the IPC of the processor with a conventional modulo index function. These results confirm the previous findings: XOR-functions improve performance by minimizing the number of conflict misses. We observe again that the choice of profiling input has an important impact on the quality of the XOR-function. Differences in conflict reduction between the heuristic and optimal XOR-functions remain present but their effect is smaller in the IPC metric than it is in the cache miss rate metric.

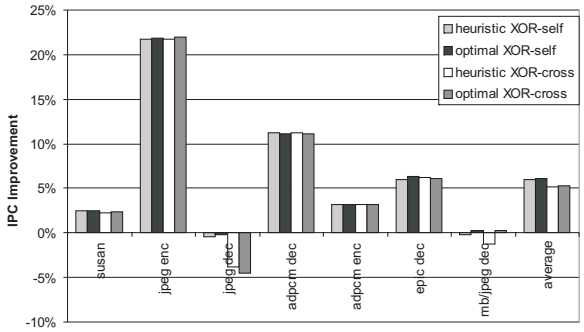

Fig. 3. IPC improvement for different XOR-functions.

5 Conclusion

Reconfigurable hash functions are an application-specific processor optimization that provides large cache conflict miss reductions. We consider hash functions that are XOR-functions, but the set of allowable functions are restricted such that the XOR is performed on at most two address bits and one of the address bits is fixed. Previous work has shown that this is an interesting class of functions.

This paper presents an algorithm to compute an optimal XOR-function yielding maximal conflict miss avoidance. The algorithm models the occurence of conflict misses using symbolic expressions. Evaluating the expression for a particular hash function yields the number of conflict misses incurred by that hash function.

Our algorithm produces hash functions that reduce more conflict misses than any other algorithm described in the literature. XOR-functions halve the number of cache misses for a set of embedded benchmarks accessing a 4 KB cache.

The optimal algorithm yields XOR-functions that outperform those of heuristic algorithms by a small margin. Yet, this small margin is sufficiently large (0.60 misses/Kuop) to motivate the use of the slower optimal algorithm.

Hash function construction algorithms generally are profile-based algorithms, which make their results dependent on the input data set used during profiling. The optimal XOR-functions are less sensitive to variations in the input data set than hash functions constructed using other algorithms described in the literature.

Acknowledgments

Hans Vandierendonck is Post-Doctoral Fellow with the Fund for Scientific Research-Flanders (FWO). This research is sponsored in part by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), Ghent University and HiPEAC.

References

1. Vandierendonck, H., Manet, P., Legat, J.D.: Application-specific reconfigurable XOR-indexing to eliminate cache conflict misses. In: Design, Automation and Test Europe. (mar 2006) 357–362
2. Vandierendonck, H.: Avoiding Mapping Conflicts in Microprocessors. PhD thesis, Ghent University (2004)
3. Rau, B.R.: Pseudo-randomly interleaved memory. In: Proceedings of the 18th Annual International Symposium on Computer Architecture. (May 1991) 74–83
4. Vandierendonck, H., De Bosschere, K.: XOR-based hash functions. IEEE Transactions on Computers **54**(7) (September 2005) 800–812
5. Patel, K., Benini, L., Macii, P., Poncino, M.: Reducing cache misses by application-specific re-configurable indexing. In: ICCAD-04: ACM/IEEE International Conference on Computer-Aided Design. (November 2004) 125–130
6. Givargis, T.: Improved indexing for cache miss reduction in embedded systems. In: Design Automation Conference. (2003)
7. Abraham, S.G., Agusleo, H.: Reduction of cache interference misses through selective bit-permutation mapping. Technical Report CSE-TR-205-94, The University of Michigan (1994)
8. Smith, A.J.: Cache memories. ACM Computing Surveys **14**(3) (September 1982) 473–530
9. Temam, O.: Investigating optimal local memory performance. In: Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems. (November 1998) 218–227
10. Vandierendonck, H., De Bosschere, K.: An optimal replacement policy for balancing multi-module caches. In: Proceedings of the 12th Symposium on Computer Architecture and High Performance Computing. (October 2000) 65–72
11. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3) (1992) 293–318
12. Bahar, R., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design. (1993) 188–191
13. Scott, J., Lee, L., Arends, J., Moyer, B.: Designing the low-power M Core architecture. In: Proceedings of the IEEE Power Driven Microarchitecture Workshop. (June 1998) 145–150
14. Lee, C., Potkonjak, M., Mangione-Smith, W.H.: MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In: Proceedings of the 30th Conference on Microprogramming and Microarchitecture. (December 1997) 330–335
15. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. In: IEEE 4th Annual Workshop on Workload Characterization. (December 2001)
16. Topham, N., González, A., González, J.: The design and performance of a conflict-avoiding cache. In: Proceedings of the 30th Conference on Microprogramming and Microarchitecture. (December 1997) 71–80