ASCIB: Adaptive Selection of Cache Indexing Bits for Removing Conflict Misses

Alberto Ros*, Polychronis Xekalakis†§, Marcelo Cintra‡, Manuel E. Acacio*, and José M. García*

*Dept. Ingeniería y Tecnología de Computadores, Universidad de Murcia
{a.ros,meacacio,jmgarcia}@ditec.um.es

†Intel Labs Barcelona
polychronis.xekalakis@intel.com

‡School of Informatics, University of Edinburgh
mc@staffmail.ed.ac.uk

ABSTRACT

The design of cache memories is a crucial part of the design cycle of a modern processor. Unfortunately, caches with low degrees of associativity suffer a large amount of conflict misses, while high-associative caches consume more power per access. We propose ASCIB, a simple technique able to dynamically adjust the bits used for cache indexing so as to minimize conflict misses. By selecting at run time the bits that disperse the working set more evenly across the available sets, ASCIB removes 73% of the conflict misses on average. This results in an improvement in energy efficiency by 17% on average.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles — Cache memories

Keywords

Cache memories, adaptive indexing, conflict misses, power

1. INTRODUCTION

With the scaling of transistors following Moore's law, the significance of the memory hierarchy to the overall system performance continues growing. The design of the first level cache is an important parameter for achieving high performance systems since it is accessed in the critical path of the processor, which determines the clock frequency of the system. Therefore, first level caches should be as fast as possible, should consume as less power as possible, and should minimize the number of misses.

Cache memories commonly use the least significant bits (LSB) of the block address to form the cache index. For applications that exhibit high spatial locality, i.e., memory requests are mostly consecutive, this indexing function works fairly well because it uniformly distributes requested blocks across the available cache sets. However, for applications that do not exhibit spatial locality, low-associativity caches can suffer lots of conflict misses due to a non-uniform distribution of blocks into cache sets. Although by increasing cache associativity this type of misses can be completely removed, this comes at a high cost in terms of access latency, area required, and power consumption.

Prior work in the field has pointed out that at any given time, 10% of the sets of a first level cache account for 90% of the conflict misses using a typical LSB indexing function [7, 10]. This suggests that a LSB indexing function does not distribute the blocks evenly among the cache sets. As a consequence, several authors have proposed to change the cache indexing function in order to distribute blocks more evenly across cache sets, thus reducing the number of conflict misses [5, 7, 12].These indexing functions are independent of the application that is accessing the cache. However, a careful inspection of the sequence of requested addresses for a particular application can lead to remove a lot of conflicting accesses, for example, by picking the address bits that guarantee a better distribution of blocks among sets to form the index [4].

Finally, it is important to consider that the sets that have the conflicting accesses change not only per application but also within the same application for different program phases [13]. This implies that static indexing policies, such as [4, 5, 7, 12], will not be able to achieve an optimal distribution of blocks. Therefore, we claim that an adaptive cache indexing policy is necessary to minimize the number of conflicting accesses.

In this paper, we propose ASCIB (Adaptive Selection of Cache Indexing Bits), a cache indexing policy that tries to find the address bits that maximize the dispersion of the working set to the available cache sets. This way, most conflict misses can be avoided. Since the working set varies both per application and per phase within the same application, the set of bits that will result in a better dispersion also changes. Therefore, our indexing policy must be able to adapt to such changes at run time.

Experimental results for the SPEC CPU2006 benchmark suite [15] using cycle accurate, full system simulation suggest that our proposal is able to remove 73% of conflict misses and 47% of the total misses for a direct-mapped cache. This is reflected in IPC improvements of 8.5% on average when compared to a conventional LSB indexing policy. Having less accesses to the L2 cache, the proposed scheme also consumes 17% less energy on average. These significant benefits come with less than 2% area overhead.

2. RELATED WORK

There are two main approaches for reducing conflict misses in low-associativity caches. The first approach relies on rehashing conflicting blocks to alternative cache sets [1, 10, 11, 13]. The sec-

§This work was done before the author joined Intel, while being at the University of Edinburgh.

ISLPED'12, July 30–August 1, 2012, Redondo Beach, CA, USA.

ond approach aims at distributing referenced blocks more evenly across the sets by changing the indexing function [4, 5, 7, 12]. Since the mechanism presented in this paper follows the later approach, we start this section by analyzing the later works.

XOR-based mapping policies (e.g., bitwise [5] and polynomial [12]) are used to obtain a pseudo-randomly placement of blocks. These mapping functions require a previous computation of the block address to obtain the cache index, being in both cases fairly simple to implement. However, XOR-based mapping policies are not aware of the particular access patterns of each application and do not perform run-time adaptation.

Kharbutli et al. [7] propose two indexing functions based on operations with prime numbers (prime modulo and prime displacement). Requiring complex logic to calculate the cache index, makes these techniques prohibitive for first level caches, where timing is highly critical. Moreover, similar to the XOR-based mapping, they do not perform run-time adaptation.

The skewed-associative cache [14] is a 2-way cache that uses one hashing function per way. This function is derived by XORing two bit fields from the block address. The most significant limitation for skewed-caches is that they cannot easily use a pseudo LRU (or true LRU) replacement policy, since there is no notion of a cache set. Since we maintain the same indexing function per set, the proposed scheme does not have this issue.

In [4] it was also noticed that the use of some bits for the cache indexing could reduce the miss rate. For detecting which bits to use, they employ offline profiling. This approach assumes that the workloads are known prior to execution, which is generally not the case (for embedded devices this may be a valid assumption).

The B-Cache [17] tries to reduce conflict misses by balancing the accesses to the sets of direct-mapped caches. In order to do this they increase the decoder length and incorporate programmable decoders and a replacement policy to the design. Our proposal is similar in that we also extend the number of bits considered for the cache index. However, the proposed scheme can choose a larger number of bits for the indexing function which is very beneficial. Additionally, the proposed indexing logic is simpler since it only has to select the bits that comprise the cache index.

The column-associative cache [1] uses a direct-mapped cache and an extra bit for dynamically selecting alternate hashing functions. Although this form of semi-associativity is able to remove a large amount of misses it does so at the cost of a large number of second accesses. Column-associative caches can be extended to include multiple alternative locations [3, 18].

The V-Way cache [11] tries to remove many of the conflict misses by allowing more tags than physical sets and using pointers to associate the tags in use with the actual sets. This scheme tries to emulate a global replacement policy for set-associative caches, thus removing conflict misses that are due to poor replacement of blocks.

3. THE ASCIB INDEXING POLICY

3.1 Detecting the Best Address Bits

An appropriate choice of the address bits that comprise the indexing function is essential for achieving a good distribution of blocks among cache sets, and therefore, for avoiding conflict misses. Since the problem of obtaining these bits from all the available ones has been demonstrated to be NP-complete [4], we opt for a simpler heuristic that iteratively improves the indexing function by only swapping one bit upon every indexing function change. After several iterations we will reach a sub-optimal solution for which any bit change will lead to a worse distribution of blocks. Of course, this solution can change dynamically depending on the variations in the working set.

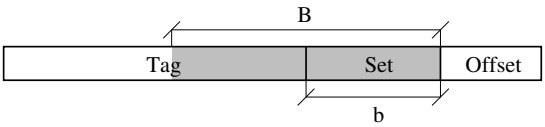


Figure 1: Memory address and bits used to index the cache.

In particular, each iteration of the algorithm is split into three phases: the *bit-victimization phase*, the *bit-selection phase*, and the *idle phase*. During the *bit-victimization phase*, our algorithm selects the bit that will be replaced from the current indexing function. Ideally, this bit should be the one that does not help in dispersing the accessed blocks evenly among the cache sets. When such a bit is found the *bit-selection phase* starts. During this phase, the bit that replaces the victimized one is selected. This bit should be the one that, along with the remaining bits (i.e., all bits forming the index apart from the victimized one), would result in spreading memory references to cache sets as much as possible. Since the bit-selection algorithm also considers the victimized bit, it may finally choose as the new bit the one we had before. This is a nice property of the algorithm, since if the current indexing function can not be improved, it is not changed. Finally, an *idle phase*, where no actions are performed by the algorithm, is introduced in order to save power consumption. Since no measurements are performed, during this phase no extra power is consumed. The underlying assumption is that once the bit-selection phase ends the indexing function will perform properly for some time.

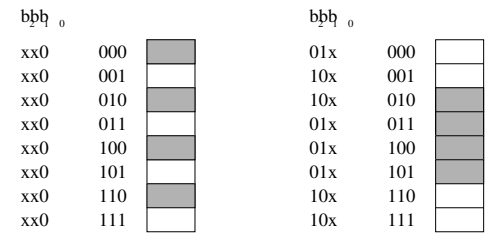
An important decision for designing the proposed scheme is the number of address bits that should be considered as candidates for being part of the indexing function. As expected the more bits are analyzed, the more conflict misses are avoided. However, at the same time if a very large number of bits is examined, our proposal may have a negative impact in both latency, area, and power consumption. Throughout this paper, we will call the number of bits needed to form the cache index as b and the number of address bits considered for selection as B (see Figure 1).

Next subsections describe these three phases in more detail. It is important to point out that the operations performed in each phase are not in the critical path of the cache access. As such the required hardware does not have any impact on the cache access latency.

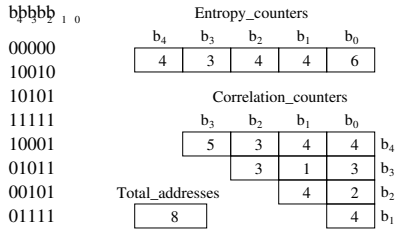
3.1.1 Bit-Victimization Phase

The goal of this phase is to find which bit should be removed from the current indexing function. We have found that bits that do not help in distributing accesses across the cache have one of the following two characteristics:

- *Low entropy*: Entropy is a metric that measures variability. A bit position that hardly changes its value for a certain set of memory references has low entropy. If a bit position with low entropy is used to form the cache index, most accessed blocks map to half of the sets in the cache, while the other half would remain almost unused. Figure 2a shows an example where bit b_0 presents no entropy at all, so referenced blocks can only map to the shaded sets.
- *High correlation*: Two bit positions that most times have the same value or most times have a different value show high correlation. If the cache index is form from two bits that have high correlation, again, half of the sets in the cache is highly utilized while the remaining half just maps a few blocks. Figure 2b shows an example where bits b_1 and b_2 are totally correlated. Again, referenced blocks can only map to the shaded sets.



(a) Low entropy example. (b) High correlation example.



(c) Counters required and example.

Figure 2: Examples and counters for the bit-victimization phase.

During this phase we are going to discard either the bit with low entropy or one of the two bits presenting higher correlation. Both metrics are calculated considering only the memory addresses of the blocks that suffer a cache miss. We need b counters to calculate the entropy of each bit and $\frac{b\ (b-1)}{2}$ counters for the correlation of any pair of bits. This results in a total of $\frac{b\ (b+1)}{2}$ counters. A deep analysis about area requirements can be found in Section 5.3.

The algorithm works as follows. At the beginning every counter is reset. Entropy counters are updated by adding each bit value to the corresponding counter, i.e., $entropy\_counter_i(EC_i)$ is increased when bit_i is 1. Correlation counters are updated by XORing every pair of bits and by adding the result to the corresponding counter, i.e., $correlation\_counter_{i,j}(CC_{i,j})$ is increased when $bit_i \oplus bit_j$ is 1. A final counter is used to store the total number of addresses processed. The victimization phase ends either when any of the counters saturates (each counter is comprised of 14 bits, i.e, it can count up to 16K misses) or after a certain number of cycles (through experimentation we have found that 100000 cycles works properly). At the end of the phase, the selected bit will be the one with lower entropy or higher correlation. In order to compare both metrics we define the usefulness metric (U) for a single bit and a pair of bits, which is calculated as follows:

$$U_i = MIN(EC_i, total\_addresses - EC_i) \quad (1)$$

$$U_{i,j} = MIN(CC_{i,j}, total\_addresses - CC_{i,j}) \quad (2)$$

The bit with lower U is chosen for being evicted from the indexing function. In case the lower U corresponds to a pair of bits (correlation), the one with lower entropy will be evicted. Figure 2c gives an example of the bit-victimization process for a 32-set cache ($b = 5$). The lower U corresponds to $CC_{1,3}$, and from these to bits, the one with lower entropy is bit b_3, so b_3 will be the discarded bit.

3.1.2 Bit-Selection Phase

This phase selects the bit that will be the best replacement for the victimized one. This choice is based on a metric that we name as *mean relative period* (MRP). In order to understand this metric, we will first explain the concept of *mean period* (MP) for an address bit position. By focusing on just one bit position of the address for a certain number of memory accesses, the period of the bit position can be defined as the number of consecutive 0's or 1's. When there is a bit change (from 0 to 1, and vice versa), a new period is computed. The mean period is then the average of these periods:

$$MP = \frac{\sum_{i=0}^{np} period_i}{np} = \frac{nb}{np} \quad (3)$$

where $\sum_{i=0}^{n} period_i$ corresponds to the number of bits in the sequence (nb), and np is the number of periods, i.e., the number of bit changes (from 0 to 1 or from 1 to 0) plus one. For example, for the sequence 00111011, $nb = 8$ and $np = 4$, so $MP = 2$.

However, although the bit with smaller mean period is the bit that more frequently changes, this metric does not guarantee that this is the best bit to be selected. This is because the new bit could highly correlated with one of the other bits that are going to form the index. Hence, we propose the *mean relative period* (MRP), which is the mean period calculated considering subsequent accesses whose address always has a set of bits kept unchanged. In our case, these set of bits are the $b-1$ bits that passed the bit-victimization phase.

$$MRP = \frac{\sum_{j=0}^{2^{b-1}} mean\ period(j)}{2^{b-1}} = \frac{\sum_{j=0}^{2^{b-1}} \frac{nb_j}{np_j}}{2^{b-1}} \quad (4)$$

Our goal is to find the bit from the remaining $B - (b-1)$ bits with lower MRP, because this will be the bit changing the most while keeping fixed the other ones, i.e., not correlated with any of the used bits. Conceptually, this will also be the bit that will help distribute better the requested blocks among sets. Figure 3 shows an example of the goodness of these two metrics.

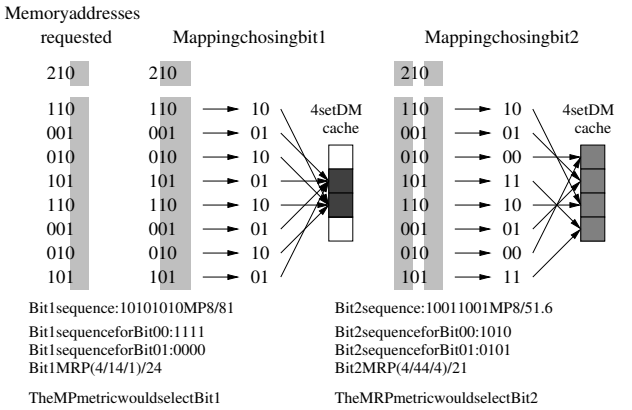


Figure 3: Example of MP and MRP metrics. The cache needs two bits for the indexing. b_o is the one that passed the bit-victimization phase. b_1 and b_2 are candidates to form the index.

The scheme illustrated in Figure 4 is used to compute this metric for each of the candidates. The required structure is a small *tag cache* with the same associativity as the data cache but half number of sets, and it is indexed using the same bits of the data cache after removing the bit that we wish to replace. Therefore, this cache uses a similar indexing logic as the data cache, which will be explained in Section 3.3. Only the address bits that we wish to analyze ($B - (b-1)$ bits) are kept in the tags field of the tag cache along with a valid bit, and as such it is fairly small. This cache is updated on every data cache access. However, due to its small size the power consumption of our proposal does not increase significantly. On a replacement from this structure, the tag of the evicted block and the tag of the new block are bitwise XORed. This provides information about changes of relative periods for each of the tested bits, which are kept in $B - (b-1)$ registers. After a certain number of cycles, the number of periods of each bit are compared. The bit with greater number of periods is selected, since it will be

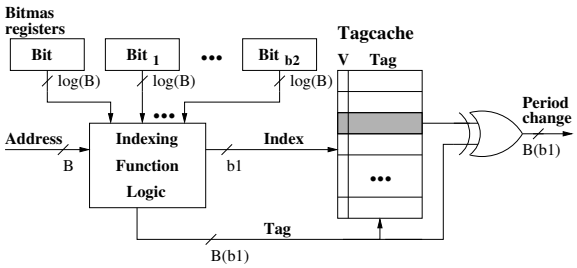


Figure 4: Scheme of the bit-selection process.

the one with smaller MRP. We have found through experimentation that a good length for this phase is 100000 cycles.

Having completed the second phase, the algorithm can proceed to update the indexing function if necessary, first evicting from the cache those blocks that will reside in a wrong set according to the new indexing function, as will be explained in Section 3.2.

3.1.3 Idle Phase

Once the bit-selection phase is completed and the indexing function has been changed (if necessary), we assume that the current function will perform appropriately for some time. Therefore, we decide to introduce an idle phase where no measurements are made, so we can save power. Through experimentation we have found that a length for this phase of 400000 cycles is a good trade-off because the proposed mechanism does not lose effectiveness. When this phase finishes, the algorithm begins a new iteration starting from the bit-victimization phase.

3.2 Updating the Indexing Function

From the point when the indexing function changes, all subsequent memory references will use the new function to access the cache. This can cause consistency issues since memory blocks that resided in the cache prior to the indexing change can be referenced and mapped to a different set.

A simple approach to maintain the consistency of the cache is to evict from cache those blocks that will be mapped to a different set after the index change. Since the proposed algorithm only changes one bit per iteration, some of the cached blocks will be mapped to the same set using the new indexing function.

In order to detect which cache blocks will map to a different set after the index change, it is only necessary to check if the selected bit and the victimized one have the same value for every block stored in cache. Therefore, a cache lookup is required on every index change. Note that the address of a cached block can be obtained by merging its tag and the set where is stored according to the indexing function employed to map the block to the cache.

If there is not a variation in the working set, our indexing function should not change. Therefore, we expect that index changes are triggered on the boundaries of different program phases where the working set also changes and as such many misses would occur in any case. As consequence, the overall impact of the misses caused by index changes should be small. Despite the simplicity of this scheme it works fine, as it will be shown in Section 5.

3.3 Forming the Adaptive Index

A critical component of the proposed mechanism is the one that forms the index out of the available bits. This circuit has to be fast since it will lie in the critical path of the cache access and as such it may impact the access time.

The proposed circuit is depicted in Figure 5. The indexing function is determined by the position of the bits selected for the indexing. This information is stored in b registers (called *bit mask registers*). These registers are updated when the indexing function changes. The size of each register is $log_2(B)$. These B bits are taken from each block address in order to calculate the corresponding cache set for the block. Our logic has b B-to-1 multiplexers, which are lied out in parallel, each of them used for selecting one address bit according to its corresponding bit mask register. The outputs of all the multiplexers form the desired cache index.

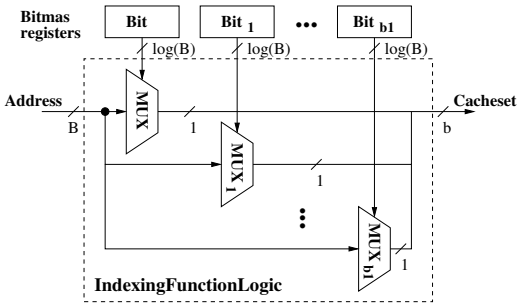


Figure 5: Forming the index using the selected bits

Table 1: Architectural parameters of the system simulated.

Parameter	Value
Core Frequency	2.33GHz
Fetch/Issue/Retire Width	6, 4, 4
I-Window/ROB	64, 100
Ld/St Queue	64, 64
Branch Predictor	YAGS 64Kb
BTB/RAS	1K entries, 2-way, 32 entries
L1 DCache and ICache	16KB, 1-way, 1 cycle
L2 Cache	2MB, 16-way, 7 cycles
Main Memory	150 cycles

The multiplexers shown in this figure can be implemented using transmission gates in order to minimize their delay. Since all multiplexers are processed in parallel, the delay of the indexing logic is represented by the delay of a B-to-1 multiplexer. This delay depends on the number of transistors in the critical path with can be calculated as a function of B. We found that a value for B greater than 16 does not improve the accuracy of the proposal, and therefore we can use 16-to-1 multiplexers, whose delay when implemented using transmission gates is very small (one inverter and four transmission gates in the critical path [2]). But even this small delay can effectively be completely hidden since the time required to perform the tag check is typically more than the one required to access the data component.

4. EVALUATION METHODOLOGY

To evaluate the proposed scheme we employ the Simics full-system simulator [8] extended with the Gems toolset [9] so as to simulate a cycle-accurate out-of-order processor (Opal) and memory subsystem (Ruby). Our area, power and timing estimations are based on Cacti 5.3 [16], assuming a 45nm process technology. We use the SPEC CPU2006 benchmark suite [15] to drive our simulation infrastructure. We fast forward all the benchmarks for the first 4 billion instructions. Throughout this period we only warm up the caches. We then simulate each of the benchmarks for a slice of 500 million instructions for which we keep statistics.

As the base configuration (Table 1) we evaluate a DM cache that uses an LSB indexing scheme (*LSB*). *ASCIB* uses a value $B = 16$. We also evaluate the bitwise Xor function [5] (*Xor*), the prime module function [7] (*PrimeMod*), the near-optimal static indexing [4] (*Static*) and the column-associative cache [1] (*ColumnAssoc*). We also include a section comparing ASCIB to set-associative caches that uses the least significant bits for indexing the cache (*LSB-2ways* and *LSB-4ways*). We pessimistically, for our scheme, assume the same access latency for all schemes.

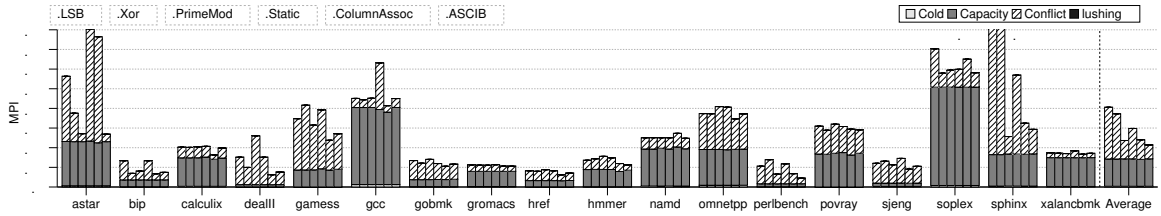


Figure 6: MPKI for the schemes evaluated in this paper.

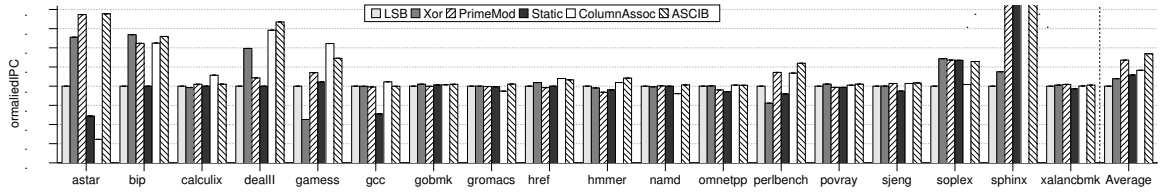


Figure 7: Normalized IPC of the schemes evaluated in this paper.

5. EXPERIMENTAL RESULTS

5.1 Performance Evaluation

Figure 6 presents the MPKI (misses per thousand instructions) for the different schemes evaluated. In this graph, cache misses are split into four categories: *Cold*, *Capacity* and *Conflict* misses corresponds to the typical categories [6], while *Flushing* misses represent the ones caused as consequence of evictions performed upon index changes. The graph shows that for the assumed 16KB DM cache, the amount of capacity misses is quite high.

We can observe that the main benefit of all the schemes over the base case comes in terms of a reduction in the conflict misses. *Xor* reduces the conflict misses by 13% on average. However, this approach increases the number of misses for several applications (e.g., *gamess*, *hmmer*, *perlbench*, and *sjeng*). *PrimeMod* avoids 65% of conflict misses on average, but do not behaves properly for some applications, particularly for *dealII* where the number of MPKI is almost doubled with respect to the base case. The static indexing is able to reduce the number of conflict misses by 41% on average. However, for some applications like *astar*, *gcc*, or *sjeng* it incurs in more misses than the base configuration. The column associative cache is able to reduce conflict misses by 62% on average, although it increases the number of MPKI for *astar* compared to *LSB*. Finally, we can see that our proposal is the one that is able to remove the largest fraction of conflict misses assuming a DM cache (73% on average), reducing the MPKI for all the applications (47% on average). Note that the amount of *Flushing* misses for our proposal is negligible. This supports our assumption that index changes happens on the boundaries of program phases.

In Figure 7, the IPC of the evaluated approaches is presented. As the graph confirms there is a strong correlation between the MPKI and the IPC. The *Xor* scheme is 2% better than the base case. Perhaps the most worrying fact for this scheme is its inconsistent behavior, since it is worse than the *LSB* indexing for some cases (e.g., *gamess* and *perlbench*). *PrimeMod* improves the IPC compared to *Xor*, obviating the fact that the logic for generating the index cannot be efficiently implemented for L1 caches. Particularly, if the hash function could be implemented in less than 1 cycle it would be 6.8% better than the base case. The static indexing is 3% better than the base case. However, it is not behaving better that the base configuration for all the benchmarks (e.g., *astar* and *gcc*). The column-associative improves the IPC by 4.1% on average over the base case. Although it obtains similar reduction in MPKI, sometimes hits require a second cache access –this happens for 22% of the hits on average– which negatively impacts performance. Finally, ASCIB is better than all the previously proposed schemes, improving IPC by 8.5% over the base case.

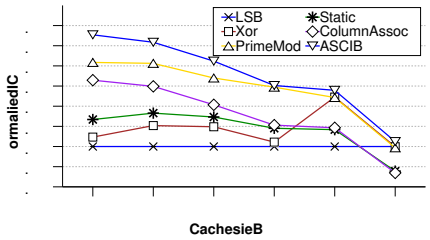


Figure 8: Normalized IPC considering different cache sizes.

To conclude the performance evaluation, Figure 8 shows the IPC of the evaluated proposals for different cache sizes (from 4KB to 128KB). We can see that the adaptive scheme outperforms the other proposals for all cache sizes. Obviously, when the cache size becomes very large, the cache miss rate decreases, and therefore, the improvements in IPC are lower. The schemes that perform closer to our proposal are *Xor* (for large cache sizes), and the unimplementable *PrimeMod*.

5.2 Energy Efficiency

Figure 9 presents an evaluation of the difference in the dynamic energy consumed by the caches for all the schemes. This figure splits the consumption of the three main structures involved in our memory hierarchy: the L1 cache, the L2 cache, and the tag cache and other counters employed by our mechanism. As it is shown, the more the L1 miss ratio is reduced, the more dynamic energy is saved at the L2 cache. The consumption of the L1 cache is similar for all the indexing policies except for the column-associative cache, where some times a cache hit requires two cache accesses. Finally, the energy consumption of the tag cache along with the counters employed by our proposal is negligible. Therefore, our proposal reduces the energy consumption compared to the base case by 17% on average, and it is the more energy efficient scheme from the proposals evaluated.

5.3 Area Overhead

Our algorithm uses the following extra structures to adapt the indexing function at run time: $\frac{b\ (b+1)}{2}$ *counters* used to detect low-entropy and high-correlated bits during the bit-victimization phase, a *tag cache* that computes the MRP for the bits considered in the bit-selection phase, and the other counters for storing the MRP for each bit being analyzed to be selected.

Table 2 shows the extra area requirements of our proposal for several cache sizes. Each of the counters employed in the bit-

Figure 9: Normalized dynamic energy consumption for the schemes evaluated in this paper.

Table 2: Area overhead of the proposed mechanism.

Cache size (KB)	Cache sets	Index bits (b)	Candidate bits	Bit-victimization bytes (overhead)	Bit-selection bytes (overhead)	ASCIB overhead
4	64	6	11	39.25 (0.90%)	69.00 (1.57%)	2.47%
8	128	7	10	51.50 (0.59%)	107.25 (1.22%)	1.81%
16	256	8	9	65.50 (0.37%)	177.50 (1.02%)	1.39%
32	512	9	8	81.25 (0.23%)	303.75 (0.87%)	1.10%
64	1024	10	7	98.75 (0.14%)	526.00 (0.75%)	0.89%
128	2048	11	6	118.00 (0.08%)	908.25 (0.66%)	0.74%

victimization phase has 14 bits. Since we only consider up to $B = 16$ bits from the address and there are $b - 1$ bits remaining from the bit-victimization phase, only $16 - (b - 1)$ bits plus the valid bit need to be stored in the tag cache, which has half number of sets than the data cache. We can also observe that our proposal scales very well with the size of the cache, i.e., the area overhead decreases as the cache size increases. Overall, the area overhead of our proposal is only 1.39% with respect to a 16KB DM cache.

5.4 Comparison to associative caches

In this section we compare our proposal to 2-way and 4-way associative caches, idealistically considering that the access time of all caches is the same, i.e., favoring associative caches. Figure 10 shows this comparison in terms of CPI, energy, and area. For the energy consumption we consider both the L1 and the L2 caches, since an associative L1 cache incur in more energy consumed per access, but also in less consumption at the L2 cache. Results are normalized with respect to a traditional DM cache (*LSB-DM*). The bars labeled as *Energy_L1* and *Energy_L2* represent the faction of the *Energy_L1+L2* bar consumed by each cache level.

The 2-way associative cache reduces CPI by 11.7%, while the 4-way associative cache reduces it by 13.4%. Although this is a higher reduction than the obtained by our proposal, this numbers consider 1-cycle hit time for the associative cache –for a 2-cycle 2-way cache, the CPI is slightly worse than our proposal–. However, this performance comes at both area and energy costs. The consumption of the L1 cache increases when we use associative caches. Although the number of L2 accesses is reduced, the impact in L1 energy consumption is more important so the overall power increases, up to 52% for a 4-way cache compared to a DM or a 2-way cache. Finally, the area overhead of ASCIB with respect to a 16KB DM cache is less than 2%. A 2-way set-associative cache increases area requirements by 8,7%.

6. CONCLUSIONS

We have proposed an adaptive cache indexing policy (ASCIB) that is able to reduce the conflict misses, by more uniformly spreading the memory references to the available sets. The basic premise of this work is that the non-uniformity in the set usage is caused by a poor selection of the index bits. Instead, by selecting as index bits the bits that disperse the working set more evenly in the available sets, a large fraction of the conflict misses can be removed.

We have shown that ASCIB is able to reduce the percentage of conflict misses by 73%. This is reflected in IPC improvements by 8.5% on average when compared with a conventional cache that used the least significant bits. Finally, when ASCIB is used for DM

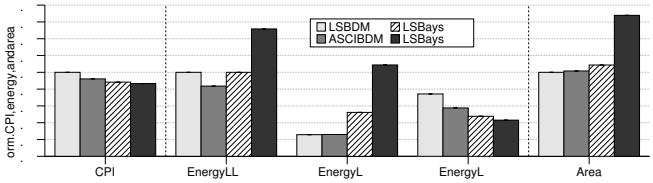


Figure 10: Comparison to associative caches in terms of CPI, dynamic energy consumption, and area. Lower is better.

caches it can obtain a CPI close to associative caches, consuming 17% less energy, and requiring 6% less area than a 2-way associative cache.

7. ACKNOWLEDGMENTS

This research was supported by the Spanish MEC and MICINN under Grant TIN2009-14475-C04.

8. REFERENCES

[1] A. Agarwal and S. D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In *20st Int'l Symp. on Computer Architecture (ISCA)*, pages 179–190, May 1993.
[2] K. Chaudhary, P. D. Costello, and V. M. Kondapalli. Programmable circuit optionally configurable as a lookup table or a wide multiplexer. U.S. Patent 7075333, Nov. 2006.
[3] B.-K. Chung and J.-K. Peir. LRU-based column-associative caches. *Computer Architecture News*, 26(2):9–17, May 1998.
[4] T. Givargis. Improved indexing for cache miss reduction in embedded systems. In *40th Design Automation Conference (DAC)*, pages 875–880, June 2003.
[5] A. González, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating cache conflict misses through XOR-based placement functions. In *11th Int'l Conference on Supercomputing (ICS)*, pages 76–83, June 1997.
[6] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers (TC)*, 38(12):1612–1630, Dec. 1989.
[7] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 288–299, Feb. 2004.
[8] P. S. Magnusson, M. Christensson, and J. Eskilson, et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.
[9] M. M. Martin, D. J. Sorin, and B. M. Beckmann, et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33(4):92–99, Sept. 2005.
[10] J.-K. Peir, Y. Lee, and W. W. Hsu. Capturing dynamic memory reference behavior with adaptive cache topology. In *8th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 240–250, Oct. 1998.
[11] M. K. Qureshi, D. Thompson, and Y. N. Patt. The v-way cache: Demand based associativity via global replacement. In *32nd Int'l Symp. on Computer Architecture (ISCA)*, pages 544–555, June 2005.
[12] B. R. Rau. Pseudo-randomly interleaved memory. In *18th Int'l Symp. on Computer Architecture (ISCA)*, pages 74–83, June 1991.
[13] D. Rolán, B. B. Fraguela, and R. Doallo. Adaptive line placement with the set balancing cache. In *42nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 529–540, Dec. 2009.
[14] A. Seznec. A case for two-way skewed-associative caches. In *20st Int'l Symp. on Computer Architecture (ISCA)*, pages 169–178, May 1993.
[15] Standard Performance Evaluation Corporation. SPEC CPU2006. `http://www.spec.org/cpu2006`.
[16] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20, HP Labs, Apr. 2008.
[17] C. Zhang. Balanced cache: Reducing conflict misses of direct-mapped caches. In *33rd Int'l Symp. on Computer Architecture (ISCA)*, pages 155–166, June 2006.
[18] C. Zhang, X. Zhang, and Y. Yan. Two fast and high-associativity cache schemes. 17(5):40–49, Sept. 1998.