Adaptive Selection of Cache Indexing Bits for Removing Conflict Misses

Alberto Ros, Polychronis Xekalakis, Marcelo Cintra, Manuel E. Acacio, and José M. García

Abstract—The design of cache memories is a crucial part of the design cycle of a modern processor, since they are able to bridge the performance gap between the processor and the memory. Unfortunately, caches with low degrees of associativity suffer a large amount of conflict misses. Although by increasing their associativity a significant fraction of these misses can be removed, this comes at a high cost in both power, area, and access time.

In this work, we address the problem of high number of conflict misses in low-associative caches, by proposing an indexing policy that adaptively selects the bits from the block address used to index the cache. The basic premise of this work is that the non-uniformity in the set usage is caused by a poor selection of the indexing bits. Instead, by selecting at run time those bits that disperse the working set more evenly across the available sets, a large fraction of the conflict misses (85%, on average) can be removed. This leads to IPC improvements of 10.9% for the SPEC CPU2006 benchmark suite. By having less accesses in the L2 cache, our proposal also reduces the energy consumption of the cache hierarchy by 13.2%. These benefits come with a negligible area overhead.

Index Terms—Cache memories, conflict misses, adaptive indexing, working set variations.

1 Introduction and Motivation

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 accesses to second level caches.

Direct-mapped caches are faster, consume less energy per access and are smaller in terms of area than set-associative caches [1]. Since direct-mapped caches do not need to hold multiple blocks per set as their set-associative counterparts do, they only have to store one tag per set in the tag array. Hence, they only require one tag comparison per access and do not require a multiplexer to select which of the ways holds the correct tag. The lack of these structures significantly reduces the amount of required wires. These properties lead to a shorter critical path, and thus, typically to a faster access time. Additionally, the overall reduction in area due to the non-existence of multiple ways and the circuitry required to support them, is significant. This leads to both less dynamic power consumed per access and less static power consumption. These characteristics make direct-mapped caches an attractive approach for being employed as first level caches, especially for embedded systems [2].

On the other hand, set-associative caches achieve higher hit rates than direct-mapped caches due to a reduction in the number of conflict misses. Conflict misses typically occur because the requested blocks tend to have non-uniform distribution across the available cache sets. By increasing the associativity of the cache, these types of misses can be completely removed. However this comes at a high cost in terms of access latency, required area, and power consumption. A cache structure achieving a good trade-off between direct-mapped and set-associative caches, would result in a smaller, more power efficient, and better performing cache.

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 a lot of conflict misses due to a non-uniform distribution of blocks into cache sets. By increasing cache associativity, these types of misses can be completely removed. However, high-associativity caches have the previously stated drawbacks, which make them undesirable.

This work is motivated by three key observations. First, 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 [3], [4]. 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], [3], [6].

Second, a careful inspection of the sequence of requested addresses for a particular application can lead to the removal of 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 [7]. Unfortunately, most of the

- *A. Ros, M.E. Acacio, and J.M. García are with the Department of Computer Engineering, University of Murcia, Spain. E-mail: {aros,meacacio,jmgarcia}@ditec.um.es*
- *P. Xekalakis and M. Cintra are with Intel. E-mail: {polychronis.xekalakis,marcelo.cintra}@intel.com*

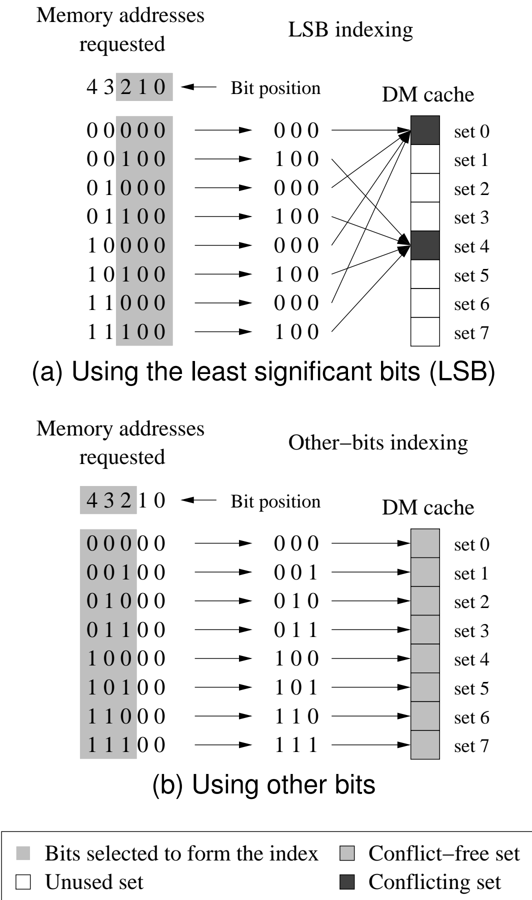


Fig. 1: Mapping memory requests to a direct-mapped cache.

previously proposed indexing functions are independent of the application that is accessing the cache.

Figure 1 illustrates the second observation for a stride memory access pattern. This is a common access pattern that appears, for example, in applications that perform matrix multiplication, where different memory blocks which are not adjacent but have fixed distance are accessed [8]. In the particular scenario shown in Figure 1, the stride is 256 bytes, and the cache is a 8-set direct-mapped (DM) cache with 64-byte cache blocks using a LSB indexing function. Figure 1a depicts how the LSB indexing policy would map the stride accesses to the mentioned cache. In this case, the choice of the least significant bits (0, 1, and 2) results in conflicting accesses in *set 0* and *set 4*. However, these conflicting accesses may be removed if a more informed choice of bits is made. This is shown in Figure 1b, where the choice of other address bits (2, 3, and 4) lead to a better dispersion of the addresses and as such, no conflicting accesses. Notice that, in this example, even a 2-way set associative cache would not be able to avoid conflicting sets.

Third, the sets that have the conflicting accesses change not only per application but also within the same application for different program phases [9]. This implies that static indexing policies, such as [7], [5], [3], [6], 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 present 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.

As demonstrated by Givargis [7], the problem of finding the optimal indexing bits is a problem unsolvable in polynomial time (i.e., NP-complete). However, we find that a smart heuristic can get a solution very close to the optimal one, while also being able to be calculated at run time. Our algorithm, similar to the simplex method, changes a single bit of the indexing function on each iteration. The decision of whether this step will lead us to a better solution is driven by a metric which we name as *mean relative period*.

Finally, changing the indexing of the cache implies that some of the cache blocks will have to be either moved, or evicted from the cache in order to keep it consistent. Fortunately, these indexing changes typically happen on program phase changes, where the working set would change in any case. Therefore, the additional misses incurred by these evictions are in most cases not very significant. Although the proposed adaptive indexing scheme is transparent to the associativity of the underlying cache, we believe that it is more appropriate for use alongside low-associativity caches, since it directly targets conflict misses.

Experimental results for the SPEC CPU2006 benchmark suite [10] using cycle accurate, full system simulation suggest that our proposal is able to remove 85% of conflict misses and 55% of the total misses for a direct-mapped cache. This is reflected in IPC improvements of 10.9% on average when compared to a conventional LSB indexing policy. Having less accesses to the L2 cache, the cache hierarchy of the proposed scheme also consumes 13.2% less energy on average. These significant benefits come with a negligible area and latency overhead.

A preliminary version of the proposed adaptive indexing policy was presented in [11]. Here, we improve the motivation and description of that work, we implement two variations of the proposal that employ two index functions and that obtain significantly better results, and we extend the evaluation by analyzing the area requirements with the CACTI tool, studying the impact in performance of the number of address bits considered for the indexing function, and showing the variation in the address bits at runtime.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 presents the adaptive cache indexing policy. The experimental methodology is then described in Section 4 and Section 5 presents the results obtained. Finally, Section 6 concludes the paper.

2 Related Work

The seminal work by Hill [1] was the first to suggest that in some cases the use of direct-mapped caches may be more profitable than the use of set-associative caches. Following

this work, many researchers investigated on design points that lie between direct-mapped and set-associative caches. Ideally, caches should have the access latency, area requirements, and power consumption of a direct-mapped cache coupled with the high hit rates of a set-associative one. One way to bridge the differences between direct-mapped caches and set-associative ones, is to start with a set-associative cache and try to access only the way that holds the requested block [12], [13], [14]. In this way, access latency can be lowered since only one way needs to be accessed for each cache access. An alternative approach is to start from a direct-mapped cache organization and try either to rehash conflicting blocks to other sets [15], [4], [16], [9] or to distribute them more evenly across the sets [7], [5], [3], [6]. Since the mechanism presented in this paper follows the latter approach, this section mainly focuses on the these works.

XOR-based mapping policies (e.g., bitwise [5] and polynomial [6]) are used to obtain a pseudo-randomly placement of blocks. In this way, a better distribution of blocks among cache sets can be obtained, thus reducing the amount of conflict misses. Like our proposal, 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. Then, they are able to reduce the conflict misses only up to some extent, and in some cases they even increase them, as we demonstrate in Section 5.2.

Kharbutli et al. [3] propose two indexing functions based on operations with prime numbers (prime modulo and prime displacement). The first one achieves better dispersion at the cost of having a more complex calculation. Unfortunately, both proposals have the drawback of wasting a few cache sets because the cache index is calculated by performing a module operation with the largest prime number that is smaller than the number of cache sets. Although these functions are able to reduce conflict misses, they are most suitable for large caches that are not accessed in the critical path of the processor (e.g., second level caches) due to two reasons. First, the fraction in unused cache sets decreases for larger caches. Second, the logic required for calculating the cache index is too complex (even for the prime displacement function). Again, the lack of adaptivity of this proposal prevent it from avoiding as much conflict misses as ours (see Section 5.2).

The skewed-associative cache [17] 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 wouldn't have the same limitation if applied to a set-associative cache. Moreover, different from our proposal, the skewed associative cache cannot be applied to direct mapped caches.

The work presented at [7] also noticed that the use of some bits for the cache indexing could reduce the miss rate. For detecting which bits to use, a new offline algorithm is presented which, when fed with some information acquired by a first run of the given workloads, can provide the overall best set of bits to use for indexing. Although this approach makes sense for the embedded domain, this is not the case for more general purpose computing. Additionally, run-time adaptation can yield significantly better results. In fact, as it will later be shown in Section 5.3, the optimal indexing functions found by our mapping policy vary both per application and per phase whitin the same application.

The B-Cache [18] 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 the sense 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, as it will be shown, it is very beneficial. Additionally, the proposed indexing logic is simpler since it only has to select the bits that comprise the cache index.

Jouppi [19] proposes the use of a small fully associative buffer, the victim cache, whose purpose is to hold conflicting blocks evicted from cache. Although the victim cache has been demonstrated to be very efficient at reducing conflict misses, different from the previous proposals and ours it adds extra cache capacity. Additionally, the victim cache requires a fully associative search for each cache miss. Therefore, our approach is more power efficient, while it also does not require an additional cycle for each conflict miss that it is able to save. Furthermore, Etsion and Feitelson [20] propose to improve the victim cache by storing frequently used blocks in direct-mapped cache and transient blocks in the victim cache. Our scheme is orthogonal to both proposals, and therefore, it could be used along with a victim cache to further reduce conflict misses.

The column-associative cache [15] 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. Moreover, our adaptive scheme is able to reduce a greater fraction of conflict misses by varying the index function at run time. Column-associative caches can be extended to include multiple alternative locations, which are described in [21], [22]. Again, our proposal can also be used for indexing column-associative caches, as we discuss later in Section 3.4.

The adaptive group-associative cache (AGAC) [4] attempts to find under-utilized cache sets in order to allocate evicted blocks to them. Similar to the column-associative cache, this approach has the drawback of needing sometimes a second access to the cache.

The V-Way cache [16] 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. Finally, the Set Balancing Cache [9], shows that further improvements can be achieved over the v-way cache by performing better set-balancing.

Differently from us, the proposed techniques aim explicitly non-first level caches.

3 ASCIB: Adaptive Selection of Cache Index Bits

The key observation this work is based on is that by changing the indexing bits depending on run-time behavior, a more uniform distribution of blocks to cache sets can be achieved. To this end, we first present an algorithm for detecting the address bits that more uniformly spread the memory references across the cache sets. Second, we outline the hardware required to form the cache index from the selected address bits. And third, we deal with the consistency problems of changing the cache indexing function at run time. We propose two solutions for the latter problem: flushing the blocks that are not correctly mapped after the index change and keeping the previous mask active.

3.1 Algorithm for 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 most conflict misses. Since the problem of obtaining these bits from the available ones has been demonstrated to be NP-complete [7], we opt for a simpler heuristic, which is similar to the simplex method. Particularly, our algorithm iteratively improves the indexing function by only swapping one bit upon every indexing function change. As it happens in the simplex method, 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.

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, extra dynamic power consumption is not introduced during this phase. The assumption made to introduce this phase 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 it will

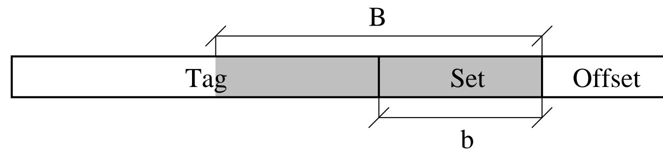


Fig. 2: Memory address and bits used to index the cache.

be demonstrated in Section 5.1, 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 2).

The following 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 because it is not helping in distributing evenly memory blocks among sets. We have found that the 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 presents 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 3a shows an example where bit b_0 presents no entropy at all. Therefore, referenced blocks can only map to the shaded sets, which represent half of the cache size.
- *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 formed 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 3b shows an example where bits b_1 and b_2 are totally correlated. As in the previous example, referenced blocks can only map to the shaded sets, which represent half of the cache size.

Therefore, during this phase we are going to discard either the bit with lower entropy or one of the two bits presenting higher correlation. In order to calculate both metrics we only consider memory addresses for blocks that suffer a cache miss. In this way, we reduce the number of times that we account for the same address while we also focus mainly on conflicting accesses and save a considerable amount of energy.

Since for each cache miss we calculate both the entropy of single bits and the correlation of pairs of bits, we need b counters for 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. Note that b is $log_2\ s$, where s is the number of cache sets. A deep analysis about the area requirements of this proposal is carried out in Section 5.5.

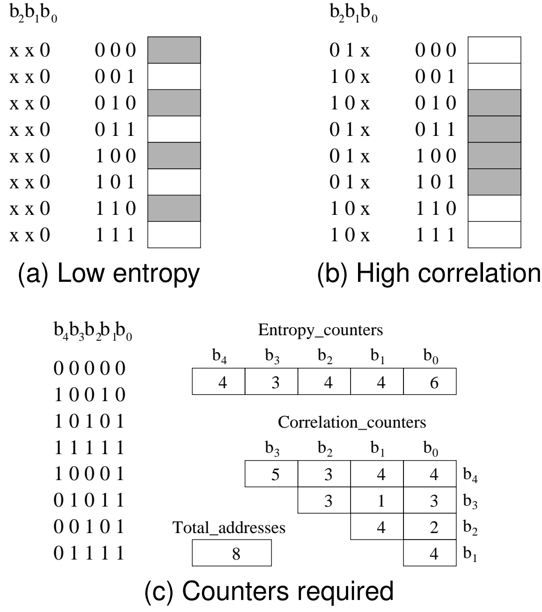


Fig. 3: Examples and counters for the bit-victimization phase.

The algorithm for the victimization phase works as follows. At the beginning of the phase, every counter is reset. Entropy counters are just 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 entropy or correlation counters saturates (each counter is comprised of 15 bits) 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 usefulness is chosen for being evicted from the indexing function. In case the lower usefulness corresponds to a pair of bits (correlation), the one with lower individual usefulness (entropy) will be evicted. Figure 3c gives an example of the bit-victimization process for a 32-set cache ($b = 5$). The lower usefulness corresponds to $correlation_{1,3}$, and from these two bits, the one with lower entropy is bit b_3, so b_3 will be the discarded bit.

3.1.2 Bit-Selection Phase

Having found a bit to victimize, this next phase is responsible for finding the bit that will be the best replacement for the victimized one. This choice is based on a metric named 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 and it is calculated as follows:

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

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 (changing every time leads to MP=1), this metric does not guarantee that this is the best bit to be selected. This is because the new bit could vary with a similar pattern with one of the other bits that are going to form the index (i.e., is correlated with an already used bit), which will ultimately lead to unused sets, as shown in the previous section. Therefore, 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.

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

In Eq. (4), nb_j represents the number of accessed addresses whose $b - 1$ bits that passed the previous phase correspond to j, and np_j is the number of 0-to-1 or 1-to-0 changes for the bit that we are evaluating considering only addresses where the $b - 1$ bits that passed the previous phase are equal to j.

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 4 shows an example of the goodness of these two metrics, where bits b_1 and b_2 are candidates to form the index along with bit b_o. Although bit b_1 shows lower MP than bit b_2, it is highly correlated to bit b_0, which will lead to a bad indexing function. Since the MRP metric also considers the other bits in the indexing function, it notices this correlation, and consequently, chooses bit b_2 as the new bit, which shows the lowest MRP possible.

The scheme illustrated in Figure 5 is used to compute the selected bit. The required structure is a small direct-mapped *tag cache* with 2^{b-1} sets. This tag cache is indexed using the same bits of the data cache after removing the bit that we

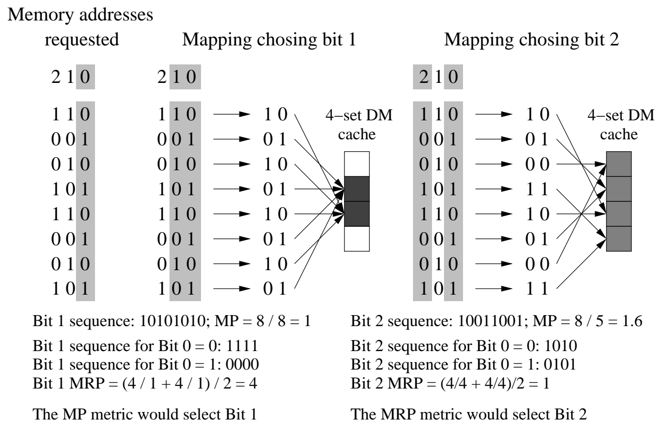


Fig. 4: Example of MP and MRP metrics.

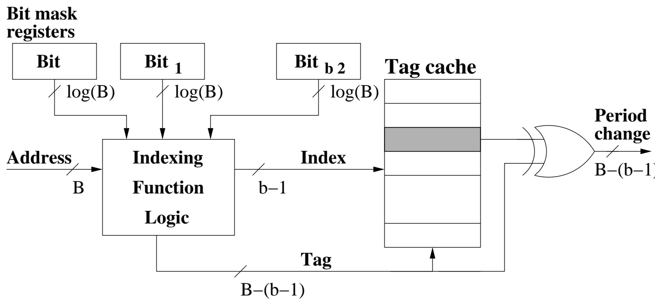


Fig. 5: Scheme of the bit-selection process.

wish to replace. Therefore, this cache uses a similar indexing logic as the data cache, which will be explained in Section 3.2. Only the address bits that we wish to analyze ($B - (b - 1)$ bits) are kept in the tags field of the tag cache. The tag cache just holds those bits (9 bits as we will see in Section 5.1), 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 bits. The aggregate number of relative periods ($\sum_{j=0}^{2^{b-1}} np_j$) for each tested bit is 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 the bit with smaller MRP. Note that this hardware does not compute the number of accesses to each set in the tag cache (nb_j). In this way, we simplify Eq. (4), thus simplifying the hardware (e.g., we do not require to perform divisions) and we save area. This simplification is based on the expectation that every nb_j will have similar values since they are the bits that better distribute the accesses. Experimentally we found that accounting for nb_j in the equation does not offer any performance improvements. Additionally, we found 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. Since an indexing function change can affect the mapping of the blocks in cache, some inconsistency issues could arise. In Section 3.3, we address this problem.

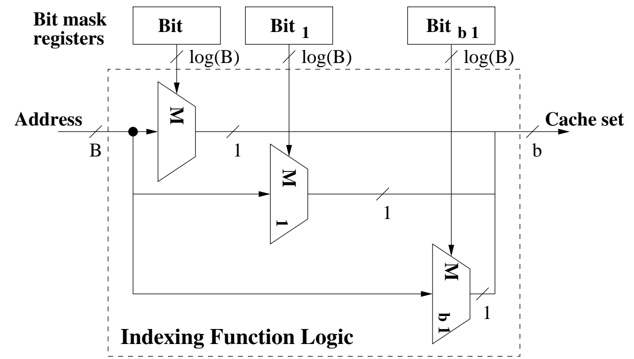


Fig. 6: Forming the index using the selected bits.

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 actions are taken by the proposed algorithm. In this way, the extra power consumption entailed by our proposal can be reduced, since no extra structures are accessed during this phase. 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 much of its effectiveness. When the idle phase finishes, the algorithm begins a new iteration starting with the bit-victimization phase.

3.2 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 its access latency.

The proposed circuit is depicted in Figure 6. 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*). One of these registers is 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 is used for selecting one address bit according to its corresponding bit mask register. The output of all the multiplexers forms the desired cache index, that is used as input for the cache decoder.

The multiplexers shown in the figure can be implemented using transmission gates in order to have small delay. Since all multiplexers are processed in parallel, the delay of the indexing logic corresponds to the delay of a B-to-1 multiplexer. This delay depends on the number of transistors in the critical path which can be calculated as a function of B. We have found that a value for B greater than 16 does not improve the accuracy of our indexing policy (see Section 5.1), and therefore we employ 16-to-1 multiplexers. The delay of a 16-to-1 multiplexer when it is implemented using transmission gates is very small since it only requires one inverter and four transmission gates in the critical path, as described in [23]. We have performed an analysis with LTSpice to check the delay

incurred by this logic. According to this tool the delay is at most 19.8 picoseconds, which is very small, and therefore we consider in the evaluation of this work that our cache can be accessed in the same number of cycles as a traditional direct-mapped cache.

3.3 Indexing Function Changes

From the point when the previously described algorithm changes the indexing function (at the end of the bit-selection phase), all subsequent memory references 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, thus having two different cache locations for the same memory block.

A simple approach to maintain the consistency of the cache is to flush those blocks that will be mapped to a different set when using the new index function. Since the proposed algorithm only changes one bit per iteration, it is quite possible that some cached blocks will be mapped to the same set using the new indexing function. Otherwise, blocks that will find their indexing changed must be evicted from cache.

Therefore, a cache lookup is required on every indexing change. For every block stored in the cache, we check whether the selected bit and the victimized bit have the same value (i.e., the block maps to the same set). If so, we only need to update the tag according to the new function. Otherwise, the block is evicted from cache. The cache is locked while it is performing the lookup.

Evicted cache blocks need to calculate their memory address if they are written back to the next cache level. Since in our proposal the tag is formed by using different address bits than in a LSB proposal, the mechanism to compose the block address is slightly different. Particularly, the address of a block is obtained by merging its bit in the tag and the bits of the set where the block is mapped according to the indexing function that was used to map it to that cache set.

If there is not a variation in the working set, our indexing function should not change (in this case the selected bit is the victimized one). Fortunately, we expect that indexing 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 a consequence, the overall impact of the misses caused by indexing changes is expected to be small. Despite its simplicity, this scheme works well, as shown later, in Section 5.

3.4 Allowing Two Indexing Functions to Coexist

An alternative approach in keeping the cache consistent is to keep active the previous index on an index change, i.e., two indexing functions at a time. In this way, it is not necessary to flush the blocks located according to the previous index. Although indexing changes are not frequent, flushing multiple blocks at the same time can cause a bottleneck.

In this approach, whenever a requested block is not found in the set designated by the current index function (*current* set), a second lookup is performed in the set obtained by the previous index function (*previous* set), in a similar way to the column-associative cache [15]. Supporting two indexing functions requires another set of bit mask registers and an extra bit for every cache entry to indicate according to which function the block is mapped.

Although keeping two functions results in alleviating most of the cache flushing, the number of cache lookups can increase due to the second cache lookups. Fortunately, by comparing how useful are the blocks that reside in the two sets, we can place the most useful ones in the current set. Blocks can thus be *promoted* from the previous set to the current set. Additionally, when a new block is brought into the cache (in the current set) causing the eviction of another block, the evicted block can be *victimized* to the previous set.

In order to perform the promotion and victimization of blocks we need a way of judging whether a line will be useful or not in the near future. We leverage upon a technique that was previously proposed to reduce the static power consumption of cache memories, namely cache decay [24]. Similarly to cache decay, we require a two-bit saturated counter per set. We increment these counters every 2K cycles and reset them on every cache access. Blocks that have their counters saturated, i.e., decayed blocks, are then thought of being no further used. Note that in contrast with the cache decay technique, we do not perform gated-vdd to the decayed blocks, we merely used the mechanism as a proxy to the usefulness of the blocks.

A block found in the cache in its previous set is promoted to the current set when the current set has either an available or a decayed entry. In the case of a decayed block, it will be evicted from cache. Likewise, a block evicted from a particular set is victimized to its alternative set (the current set if it is victimized from the previous set, and vice versa) if the alternative set has an entry which is available, decayed, or with a higher decay value than the victimized block. In the case of a decayed or live block, it will be evicted from cache. Note that some blocks cannot promote or be victimized because both indexing functions map the block to the same set.

Since we opt for the decay counter solution in order to decide on whether we should promote or victimize blocks, the additional hardware required for this scheme is only minimal and as such we feel it is a reasonable optimization.

Finally, when a decision is taken to change the current indexing function again, the previous index will be removed, the current index will become the previous one, and the newly generated index will become the current one. In this case, all blocks stored in the cache according to the old-previous index must be evicted. Since index changes happen at a very coarse granularity and thanks to the promotion mechanism, typically these blocks are very few.

4 Evaluation Methodology

4.1 Simulation Environment and Benchmarks

The proposed indexing policy is evaluated with the Simics full-system simulator [25] running the Solaris 10 operating system and extended with the Gems toolset [26] so as to simulate a cycle-accurate out-of-order processor (Opal) and memory subsystem with two cache levels (Ruby). Our power

TABLE 1: Base CACTI configuration.

Parameter	Value
Block size	64 bytes
Number of ports	1 read-write
Bank count	1 bank
Technology (nm)	32
Cell & peripheral type	itrs-hp
Bus width	64 bits
Temperature ($^{\circ}K$)	350

TABLE 2: Architectural parameters.

Parameter	Value
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
Minimum misprediction	16 cycles
L1 D-cache and I-cache	16KB, 1-way, 1 cycle
L2 cache	2MB, 16-way, 7 cycles
Main memory	150 cycles
Victimization phase timeout	100K cycles
Selection phase timeout	100K cycles
Idle phase timeout	400K cycles

and area estimations are based on Cacti 6.5 [27], assuming the parameters shown in Table 1. The main differences with respect to the CACTI parameters in [11] are the technology and the bus width ($45nm$ and 256 bits in [11]).

Although the proposed indexing policy can also be implemented for associative caches, this work evaluates its impact on direct-mapped L1 data caches (and not to the instructions cache), since it is for those caches when it is more effective. In this way, we use as the base for the simulations a 16KB direct-mapped L1 cache. According to Cacti 6.5, we can assume 1 cycle for accessing the 16KB L1 cache considering a processor frequency of 3.8GHz or lower. Additionally, we also provide a sensitivity analysis for L1 cache sizes ranging from 4KB to 128KB (Section 5.2). Finally, we also compare our proposal applied to direct-mapped caches to both 2-way and 4-way set-associative caches (Section 5.6). We employ the same access latency for every indexing function and L1 cache evaluated (1 cycle). The parameters chosen for the base system are shown in Table 2. Through experimentation we have found that a value between 50K and 200K for the victimization and selection phases show similar results (and the best in terms of IPC). For the idle phase, the IPC is not increased until 800K cycles, where it increases less than 1%, on average.

Table 3 shows in the first group of rows the specific parameters for each cache structure in the system (i.e., the L1 cache, the L2 cache, and the *tag cache* employed by our indexing policy). In the second group of rows the table shows the results obtained with the CACTI tool for each cache. These results focus in both the power and area consumed by the caches. We can see the increase associativity for the L1 cache results in increase in dynamic energy, leakage, and area. The larger size and associativity of the L2 cache also leads to more consumption and area than the L1 cache. Finally, the tag cache incurs minimal power consumption and area. More details about power consumption and area requirements can be found in Section 5.

We use the SPEC CPU2006 benchmark suite [10] 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 collect statistics.

4.2 Overview of evaluated approaches

We evaluate three approaches that implement our adaptive indexing function and we compare them to other direct-mapped caches that use previously proposed hashing functions and rehashing techniques.

As the base configuration, we evaluate a direct-mapped cache that uses the least significant bits for indexing the cache (*LSB*). The results obtained for the other proposals evaluated in this work are normalized with respect to this base configuration.

The three approaches that implement our adaptive indexing are: *ASCIB*, which is the basic approach that implements a single indexing function; *ASCIB-2*, which implements two indexing functions but never moves blocks from one set to another; and *ASCIB-2-decay*, which implements two functions and employs the decay technique to promote and victimize blocks.

Regarding the state of the art, we also evaluate the bitwise Xor function proposed by González et al. [5] (*Xor*) and the prime module function presented by Kharbutli et al. [3] (*PrimeMod*). Although the latter approach requires a large latency to form the cache index, we optimistically assume that it can be calculated in a fast way and the L1 cache can be accessed in just one cycle. The near-optimal static indexing proposed by Givargis [7] (*Static*) have also been evaluated. In order to get the near-optimal indexing function we have first run all the applications and for everyone we have obtained the bits forming the indexing function according to the heuristic algorithm presented in [28]. Additionally, we have implemented and evaluated the column-associative cache [15] (*ColumnAssoc*), assuming that second accesses require one extra cycle.

Finally, we also perform a comparison with respect to 2-way and 4-way set-associative caches using the least significant bits for indexing the cache (*LSB-2ways* and *LSB-4ways*, respectively). We also assume for this comparison that these caches can be accessed in just one cycle. Obviously, adding extra cycles in the access of these caches will lead to a worse performance.

5 Experimental Results

In this section, we first analyze the number of bits that should be considered by the metric employed in the bit-selection phase, since this number can affect the cache access latency. We then evaluate the performance of the proposed indexing function with respect to the other proposals described in the previous section. We also study the importance of the adaptation to different phases and applications in a indexing policy. Then, we show the memory overhead and the power consumption of our proposal. Finally, we perform a comparison to set-associative caches.

TABLE 3: Cache memory hierarchy details and CACTI's outputs.

Parameter // Output	L1 cache (1 way)	L1 cache (2 ways)	L1 cache (4 ways)	L2 cache (16 ways)	Tag cache (1 way)
Size (data)	16384 bytes	16384 bytes	16384 bytes	2097152 bytes	0 bytes
Access mode	fast	fast	fast	normal	normal
Associativity	1 way	2 ways	4 ways	16 ways	1 way
Tag size	default	default	default	default	9 bits
Dynamic energy per access (nJ)	0.00797120	0.0107714	0.0172302	0.101193	0.000370928
Leakage (mW)	7.70152	8.68475	11.0876	827.136	0.0819848
Area (mm^2)	0.0436331	0.0495356	0.0644173	3.91218	0.000485455

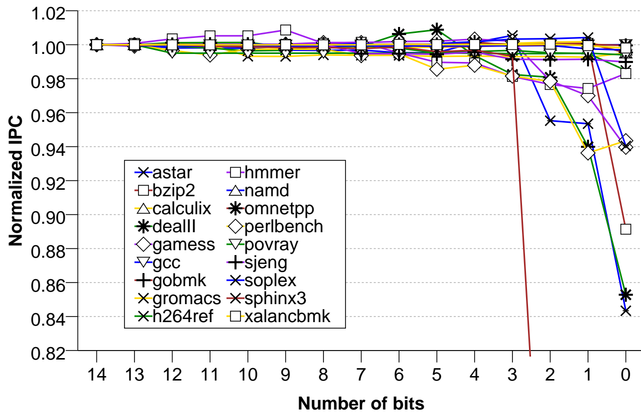


Fig. 7: IPC varying the number of extra address bits considered for the adaptive indexing function.

5.1 Analysis of the Number of Address Bits Considered for the Indexing Function

The number of bits considered for generating the index is an important design decision, since it trades avoidable cache misses for complexity/latency for generating the index. The more cache misses we avoid the more IPC we obtain.

Figure 7 presents the IPC for our ASCIB approach (with a single function) when we vary the number of address bits to be considered. The x axis represents the value of $B-b$, which is the number of extra bits (apart from the ones already used for the indexing function) to be analyzed. When $B-b$ takes a value of zero (i.e., zero extra bits considered), our proposal behaves as a LSB policy. IPC is normalized with respect to a configuration that considers 14 extra bits.

As expected, IPC improves as more address bits are considered. These improvements in IPC can be directly contributed to reductions in the number of conflict misses. Note also that a value for $B-b$ greater than 8 (i.e., $B=16$, since $b=8$ for a 256-set direct-mapped cache) does not improve the IPC considerably. Because the indexing logic for values of B larger than 16, would introduce an additional transmission gate in the critical path of the cache access [23], we conclude that a value of B equal to 16 is the best design point. Thus, we will assume this value for the simulation results shown in the following sections.

5.2 Performance Evaluation

Figure 8 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 3C categories [29], while *Flushing* misses represent the misses caused as consequence of the cache flushing performed upon an indexing change. The graph shows that for the assumed 16KB direct-mapped 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 amount of conflict misses. Compared to the base case, *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* is able to avoid 65% of conflict misses, on average, but it does not behave 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 reduces conflict misses by 62%, on average. However, it increases the number of MPKI for *astar* when compared to *LSB*. Finally, *ASCIB* is able to remove 73% of conflict misses, reducing the MPKI for all the applications by 47%, on average; *ASCIB-2* obtains results similar to *ASCIB*, since the impact of flushing is not very high; and *ASCIB-2-decay* removes the largest fraction of conflict misses thanks to the victimization policy that considers the decay degree of blocks (85%, on average), thus reducing the MPKI by 55%, on average.

We can also notice that the amount of *Flushing* misses added by *ASCIB* is negligible. These results support our assumption that the eviction of cache blocks upon indexing changes does not severely impact on applications' performance, since indexing changes are not too frequent and, moreover, they happen on the boundaries of program phases.

Figure 9 presents the IPC for the evaluated approaches. As the graph confirms there is a strong correlation between MPKI and IPC. The *Xor* scheme improves the base case by 2%. Perhaps the most worrying fact for this scheme is its inconsistent behavior, since it is worse than *LSB* for some applications (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 we assume that a L1 cache that uses the *PrimeMod* hash function can be accessed in 1 cycle, the obtained improvements would be 6.8% with respect to the base case. The static indexing improved IPC by 3% compared to the base case. However, it is not behaving better than the

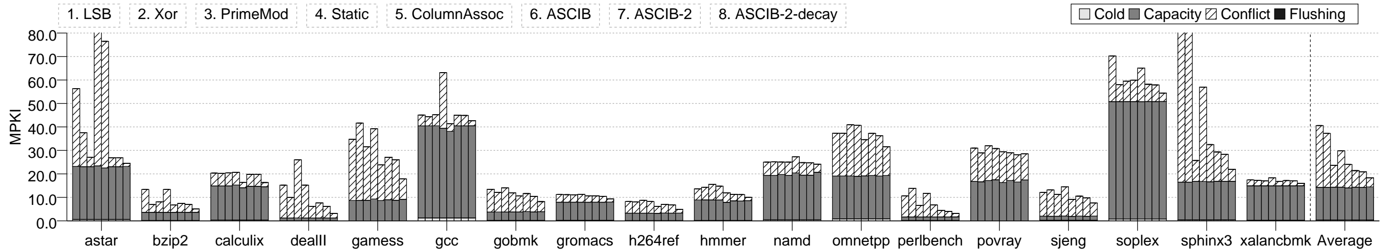


Fig. 8: MPKI for the schemes evaluated in this paper.

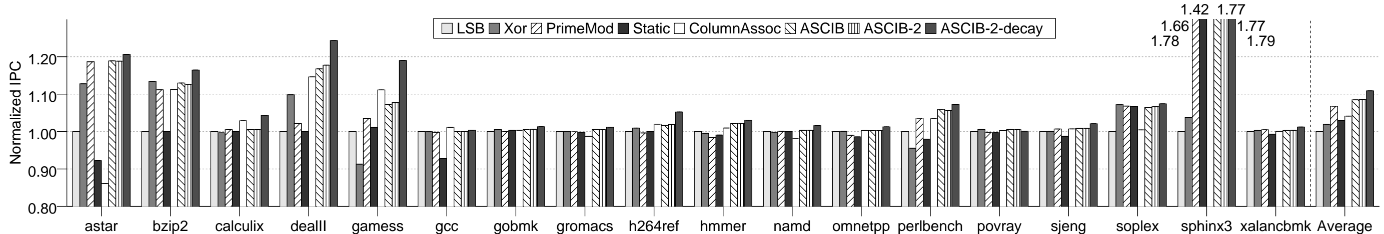


Fig. 9: Normalized IPC of the schemes evaluated in this paper.

base configuration for all the benchmarks (e.g., *astar* and *gcc*). The column-associative cache improves the IPC by 4.1%, on average, over the base case. Although it obtains similar reduction in MPKI as the *PrimeMod* function, sometimes hits are due to a second access, which negatively impacts performance (our results show that 22% of the hits of column-associative caches need a second access). *ASCIB* is better than all the previously proposed schemes, improving IPC by 8.5% over the base case. Similar results are obtained for *ASCIB-2*, but when the decay technique is applied, the improvement of using two indexing functions are notable. Particularly, *ASCIB-2-decay* improves execution time by 10.9%.

To conclude the performance evaluation, we do a sensitivity analysis varying the cache size. Figure 10 shows the IPC of the evaluated proposals for cache sizes ranging from 4KB to 128KB. We can observe that the adaptive schemes outperform 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. We can also appreciate that the advantages of using two index functions remain valid for all evaluated cache sizes. The schemes that perform closer to *ASCIB* are *Xor*, for large cache sizes, and *PrimeMod* (although the latter cannot be efficiently implemented for L1 caches).

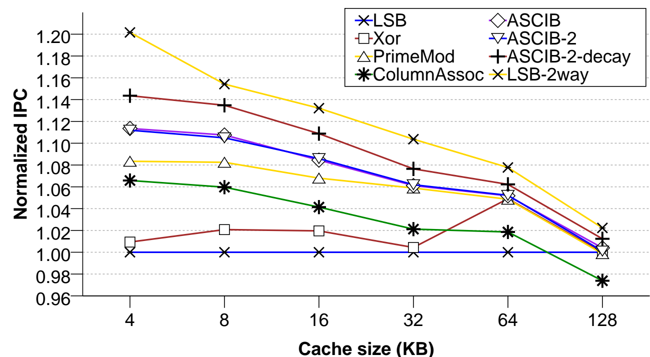


Fig. 10: Normalized IPC considering different cache sizes.

5.3 Analyzing Run-Time Adaptation

This section tries to shed some light on how the adaptation scheme works. First, we show how many indexing changes happened in the simulated time interval and how many lines are evicted in each change. Then, we plot the run-time variation of the bits used to form the index for every application.

As Table 4 reveals, the numbers of indexing changes during the execution of 500 million instructions are very low. We can observe that *Sphinx3* is the application with more changes (around 1 change every 200000 instructions). On the other hand, *povray* is the application with less index changes (just twelve). On every indexing change some lines have to be evicted. Since our cache size is 16KB, the number of 64-byte blocks in the cache is 256. We can observe that, when we employ a single indexing function, between one half or two thirds of the cache blocks are evicted upon indexing changes, on average. When we employ a second indexing function and it has to be discarded due to an index change, there are almost no blocks in the cache mapped according to this previous function because the new blocks coming to the cache are stored in the current set. Finally, when the decay technique is employed, the victimization of blocks moves more blocks to the previous set and, consequently, there are more flushed blocks than in *ASCIB-2*.

Figure 11 plots the run-time variation of the bits used to form the index for every application. In particular, the figure depicts the eight bits forming the cache index from the 16 address bit analyzed by our mechanism (from position 6th to 21st if we skip the block offset). We can observe that the bits forming the indexing policy varies both per application and per phases within the same application. Therefore, a static hashing function, that is not able to adapt to such variations, is fundamentally limited.

Interestingly, some applications have lots of index changes

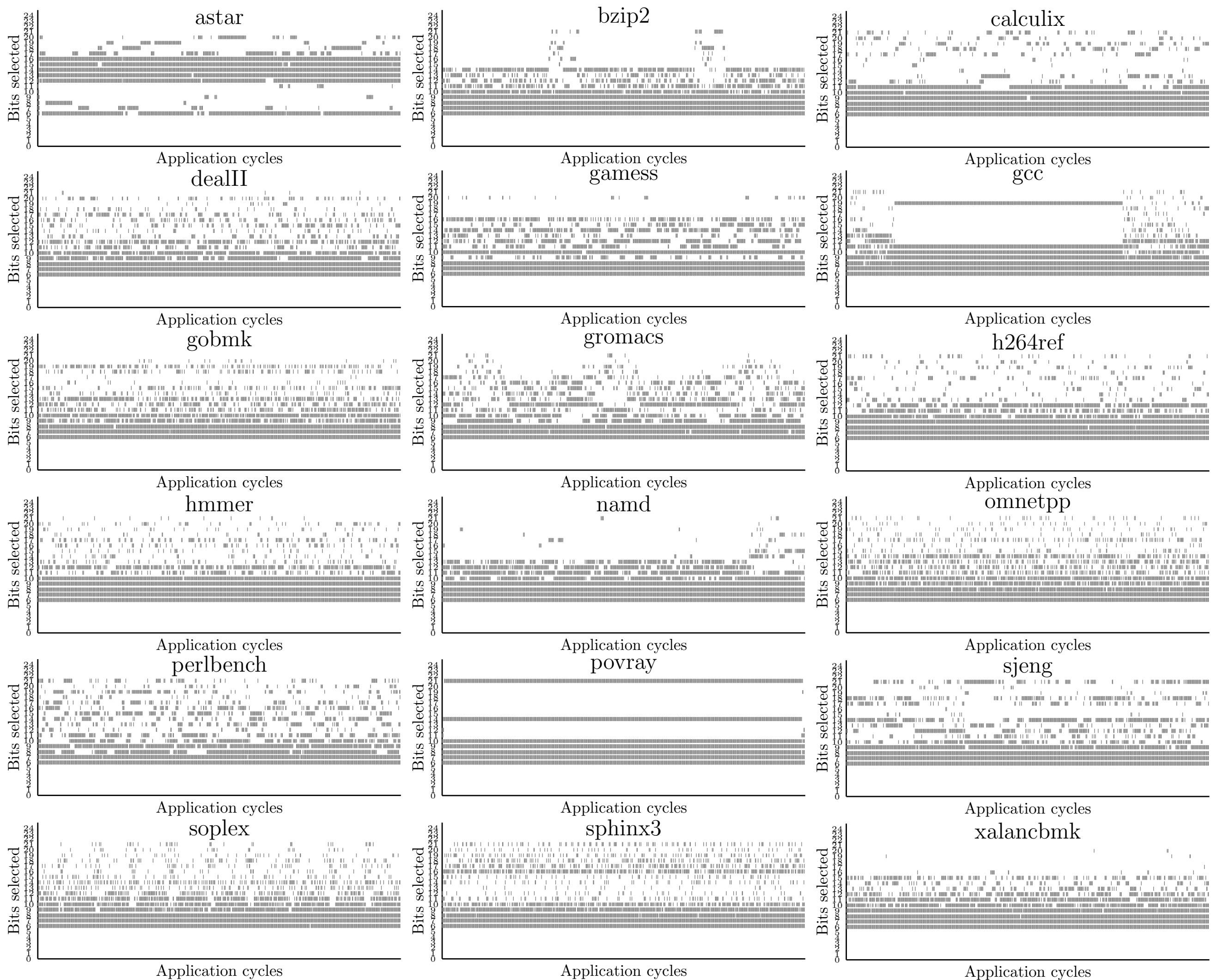


Fig. 11: Run-time variation of bits used to index the L1 cache for each application.

TABLE 4: Number of indexing changes and average number of lines evicted per change.

Benchmark	Indexing changes	Flushes per change		
		ASCIB	ASCIB-2	ASCIB-2-decay
astar	154	175.2	12.3	58.9
bzip2	444	136.1	11.4	38.1
calculix	120	166.2	0.4	36.9
dealII	547	148.3	33.1	51.5
gamess	294	179.8	14.5	45.9
gcc	765	155.1	7.5	46.2
gobmk	1281	158.0	23.9	42.9
gromacs	741	166.9	4.0	45.8
h264ref	485	149.4	14.0	45.8
hmmer	734	164.8	0.5	35.3
namd	184	171.2	0.8	44.9
omnetpp	2304	168.3	1.3	44.3
perlbench	679	123.0	35.5	39.7
povray	12	147.9	28.2	37.6
sjeng	506	146.9	18.9	36.5
soplex	2461	176.7	1.8	48.9
sphinx3	2650	162.8	12.9	61.5
xalancbmk	763	160.4	0.1	43.0

while others use the same index bits for a long time. Particularly, for some applications (e.g., *bzip2*, *calculix*, *gromacs*, and *soplex*) a similar behavior of our indexing policy is repeated several times during the execution of the application. This behavior corresponds to several iterations, where the working set changes inside each iteration.

Finally, as expected, the least significant bits are more used than the most significant bits on average. For example, *bzip2*, *calculix*, *namd*, and *xalancbmk* use to form the index by using such bits. However, some applications require the use of some most significant bits to achieve a more even distribution. Surprising cases are *astar*, *gcc*, and *povray*. *Astar*, avoids using the bits from position 7th to position 11th, while *gcc* and *povray* use most of the time one bit of the most significant ones (21st and 19th, respectively). These are examples where the B-Cache would not work as well as the proposed scheme.

5.4 Energy Efficiency

Figure 12 presents an evaluation of the difference in the dynamic energy consumed by the caches employed for all the schemes. The figure splits the consumption according to 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

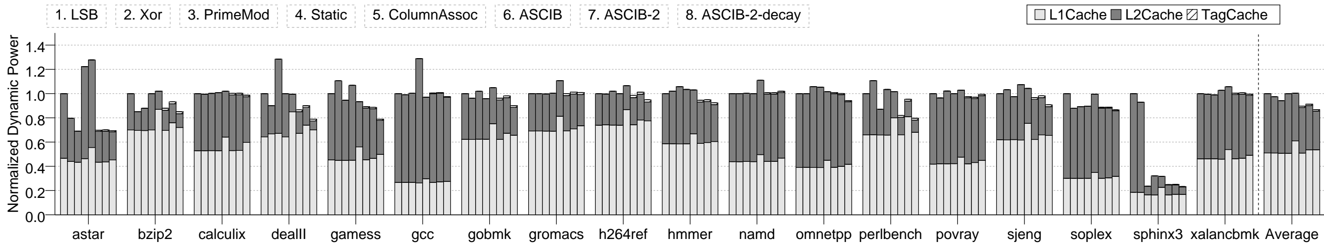


Fig. 12: Normalized dynamic energy consumption for the schemes evaluated in this paper.

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 and for the two-index ASCIB proposals, where sometimes a cache hit requires two cache accesses. We found that the number of hits in the second set in the column associative cache is significantly higher than in both ASCIB proposals. Finally, it is important to note that the energy consumption of the tag cache along with the counters employed by our proposal is insignificant. Therefore, *ASCIB* reduces the energy consumption of the cache hierarchy by 10.3% compared to the base case, on average, *ASCIB-2*, has slightly more consumption due to the second accesses to the L1, and *ASCIB-2-decay* is the most energy efficient scheme (13.2% reductions with respect to *LSB*, on average) thanks to the large reduction achieved in the number of misses.

5.5 Memory and Area Overhead

This section analyses the overhead of *ASCIB* compared to a cache that uses a LSB indexing function. We mainly focus on the size of the structures used for the algorithm that adapts the indexing function at run time: the $\frac{b\ (b+1)}{2}$ *counters* used to detect low-entropy and high-correlated bits during the bit-victimization phase, a *tag cache* used to compute the mean relative period for all the bits considered in the bit-selection phase, and other counters for storing the mean relative period for each bit being analyzed to be selected.

Table 5 shows the extra memory requirements of our proposal for several cache sizes. Each of the counters employed in the bit-victimization phase has 14 bits. The total counter has 20 bits. On the other hand, since we only consider up to 16 bits from the address (this is the value that we have choose for B) and there are $b-1$ bits remaining from the bit-victimization phase, only $16-(b-1)$ bits need to be stored in the tag cache. Moreover, the number of sets of the tag cache is 2^{b-1}, i.e., half the number of sets of the data cache. From table 5, we can also observe that our proposal scales very well with the size of the cache, i.e., the area overhead of our proposal decreases as the cache size increases. The memory overhead of our proposal with respect to a 16KB direct-mapped cache is only 1.28%.

We also have used CACTI to calculate the area requirements of our tag cache and its area overhead with respect to the L1 cache, as Table 5 also shows. Again, we can see that the area requirements of our tag cache are insignificant, having a 1.00% overhead over the LSB indexing policy for 16KB direct-mapped caches. Additionally, the larger the cache size is the lower is the overhead of our tag cache.

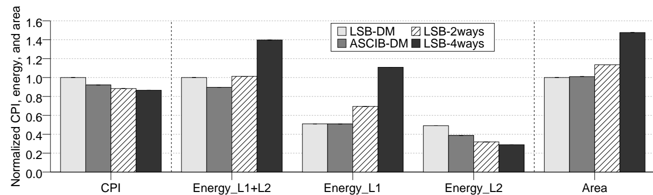


Fig. 13: Comparison to associative caches in terms of CPI, dynamic energy consumption, and area. Lower is better.

Finally, note that the requirements for the tag cache in area overhead are the same for the three *ASCIB* proposals. With respect to the L1 cache, *ASCIB-2* only requires one extra bit per entry to indicate the function that was used to map the block, while *ASCIB-2-decay* only requires two additional bits to store the decay value of each block in the cache. Therefore, the memory overhead these two proposals is very small (less than 0.2% for *ASCIB-2* and 0.6% for *ASCIB-2-decay* with respect to *ASCIB*).

5.6 Comparison to associative caches

In this section we compare *ASCIB* to 2-way and 4-way associative caches, considering that the access time of all caches is the same. Figure 13 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 labelled as *Energy_L1* and *Energy_L2* represent the fraction 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 that obtained by our proposal, these numbers consider 1-cycle hit time for the associative cache –for a 2-cycle 2-way cache, the CPI is slightly worse than that reached by our proposal. However, this performance comes at both area and energy costs. The energy consumption of the L1 cache increases when we use associative caches, as previously shown in Table 3. Although the number of L2 accesses is reduced, the impact in the energy consumption of the L1 cache is more important so the overall power increases, up to 1.3% and 39.8% for a 2-way and 4-way cache, respectively, compared to a direct-mapped cache. Thus, our proposal consumes 11.5% less energy than a 2-way associative cache and 35.9% less energy than a 4-way associative cache. Finally, the area overhead of our proposal with respect to a

TABLE 5: Memory and area overhead of the proposed mechanism.

Cache size (KB)	Cache sets	Index bits (b)	Candidate bits	Memory overhead: Bit-victimization bytes (overhead)	Memory overhead: Bit-selection bytes (overhead)	Memory overhead: ASCIB overhead	Area overhead: L1 cache area (mm^2)	Area overhead: Tag cache area (mm^2)	Area overhead: ASCIB overhead
4	64	6	11	39.25 (0.90%)	63.25 (1.44%)	2.34%	0.00944308	0.000140266	1.49%
8	128	7	10	51.50 (0.59%)	97.50 (1.11%)	1.70%	0.0162362	0.000231336	1.42%
16	256	8	9	65.50 (0.37%)	159.75 (0.91%)	1.28%	0.0436331	0.000436909	1.00%
32	512	9	8	81.25 (0.23%)	270.00 (0.77%)	1.00%	0.0934962	0.000883619	0.95%
64	1024	10	7	98.75 (0.14%)	460.25 (0.66%)	0.80%	0.195248	0.00213503	1.09%
128	2048	11	6	118.00 (0.08%)	778.50 (0.56%)	0.64%	0.375293	0.00290737	0.77%

16KB DM cache is about 1%, while a 2-way set-associative cache increases area requirements by 13.5%. Therefore, our proposal requires 11.0% less area than a 2-way associative cache and 31.6% less area than a 4-way associative cache.

6 Conclusions

In this paper we present an adaptive cache indexing policy 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, which finally leads to improvements in applications' execution time.

Because the proposed mechanism is able to better disperse memory blocks across cache sets, it achieves significantly lower miss rates than conventional caches. Overall, we show that the proposed scheme reduces the percentage of conflict misses by 73% and 85% with one and two index functions, respectively. This is reflected in IPC improvements by 8.5% and 10.9%, on average, when compared to a conventional cache that uses a least significant bits indexing. Additionally, our proposal reduces the energy consumption of the cache hierarchy by 10.3% and 13.2%, respectively, with a negligible area overhead compared to a direct-mapped cache. Finally, when our indexing policy implementing a single function is used along with direct-mapped caches it can obtain a CPI close to that of associative caches, consuming 11.5% less energy and requiring 11.0% less area than a 2-way associative cache.

Acknowledgements

This work was supported by the Spanish MINECO, as well as by European Commission FEDER funds, under grant TIN2012-38341-C04-03.

References

[1] M. D. Hill, "A case for direct-mapped caches," *IEEE Computer*, vol. 21, no. 12, pp. 25–40, Dec. 1988.

[2] G. Bournoutian and A. Orailoglu, "Miss reduction in embedded processors through dynamic, power-friendly cache design," in *45th Design Automation Conference (DAC)*, Jun. 2008, pp. 304–309.

[3] 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)*, Feb. 2004, pp. 288–299.

[4] 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)*, Oct. 1998, pp. 240–250.

[5] A. González, M. Valero, N. Topham, and J. M. Parcerisa, "Eliminating cache conflict misses through XOR-based placement functions," in *11th Int'l Conf. on Supercomputing (ICS)*, Jun. 1997, pp. 76–83.

[6] B. R. Rau, "Pseudo-randomly interleaved memory," in *18th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1991, pp. 74–83.

[7] T. Givargis, "Improved indexing for cache miss reduction in embedded systems," in *40th Design Automation Conference (DAC)*, Jun. 2003, pp. 875–880.

[8] J.-L. Baer and T.-F. Chen, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers (TC)*, vol. 44, no. 5, pp. 609–623, May 1995.

[9] 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)*, Dec. 2009, pp. 529–540.

[10] Standard Performance Evaluation Corporation, "SPEC CPU2006," http://www.spec.org/cpu2006. [Online]. Available: http://www.spec.org/cpu2006

[11] A. Ros, P. Xekalakis, M. Cintra, M. E. Acacio, and J. M. García, "Ascib: Adaptive selection of cache indexing bits for reducing conflict misses," in *Int'l Symp. on Low Power Electronics and Design (ISLPED)*, Jul. 2012, pp. 51–56.

[12] B. Calder and D. Grunwald, "Predictive sequential associative cache," in *2nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 1996, pp. 244–253.

[13] T. Juan, T. Lang, and J. J. Navarro, "The difference-bit cache," in *23rd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1996, pp. 114–120.

[14] L. Liu, "Cache designs with partial address matching," in *27th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 1994, pp. 128–136.

[15] 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)*, May 1993, pp. 179–190.

[16] 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)*, Jun. 2005, pp. 544–555.

[17] A. Seznec, "A case for two-way skewed-associative caches," in *20st Int'l Symp. on Computer Architecture (ISCA)*, May 1993, pp. 169–178.

[18] C. Zhang, "Balanced cache: Reducing conflict misses of direct-mapped caches," in *33rd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2006, pp. 155–166.

[19] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *17th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1990, pp. 364–373.

[20] Y. Etsion and D. G. Feitelson, "L1 cache filtering through random selection of memory references," in *16th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2007, pp. 235–244.

[21] B.-K. Chung and J.-K. Peir, "LRU-based column-associative caches," *Computer Architecture News*, vol. 26, no. 2, pp. 9–17, May 1998.

[22] C. Zhang, X. Zhang, and Y. Yan, "Two fast and high-associativity cache schemes," vol. 17, no. 5, pp. 40–49, Sep. 1997.

[23] 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.

[24] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *28th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2001, pp. 240–251.

[25] P. S. Magnusson, M. Christensson, and J. Eskilson, et al, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[26] M. M. Martin, D. J. Sorin, and B. M. Beckmann, et al, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[27] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0," HP Labs, Tech. Rep. HPL-2009-85, Apr. 2009.

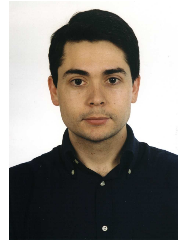
[28] T. Givargis, "Zero cost indexing for improved processor cache performance," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 11, no. 1, pp. 3–25, Jan. 2006.

[29] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Transactions on Computers (TC)*, vol. 38, no. 12, pp. 1612–1630, Dec. 1989.

Alberto Ros received the MS and PhD degree in computer science from the University of Murcia, Spain, in 2004 and 2009, respectively. In 2005, he joined the Computer Engineering Department at the same university as a PhD student with a fellowship from the Spanish government. He has been working as a postdoctoral researcher at the Technical University of Valencia and at Uppsala University. Currently, he is Associate Professor at the University of Murcia. He is working on designing and evaluating efficient and scalable manycore architectures. His research interests include cache coherence protocols and memory hierarchy designs.

Polychronis Xekalakis is currently a computer architect at Intel. He received his Ph.D. degree from the University of Edinburgh in 2009 and his Dipl. Eng. from the University of Patras in 2005. His research interests include co-designed virtual machines, speculative multithreading and architectural techniques for low power.

Marcelo Cintra received the BS and MS degrees from the University of Sao Paulo in 1992 and 1996, respectively, and the PhD degree from the University of Illinois at Urbana-Champaign in 2001. He was a Professor at the University of Edinburgh until 2011, when he joined Intel Labs as a Senior Research Scientist. His research interests include parallel architectures, optimizing compilers, and parallel programming. He has published extensively in these areas. He is a senior member of the ACM, the IEEE, and the IEEE Computer Society.



Manuel E. Acacio is an Associate Professor of computer architecture at the University of Murcia, Spain. He joined the Computer Engineering Department (DiTEC) in 1998, after he received the MS degree in computer science. Dr. Acacio started as a Teaching Assistant, at the time he began his work on his PhD thesis, which he successfully defended in March 2003. Before, in the summer of 2002, Dr. Acacio worked as a summer intern at IBM TJ Watson, Yorktown Heights (NY). After that, he became an Assistant Professor in 2004, and an Associate Professor in 2008. Currently, Dr. Acacio leads the Computer Architecture & Parallel Systems (CAPS) research group at the University of Murcia, which is part of the ACCA group. He is currently an Associate Editor of IEEE TPDS. His research interests are focused on the architecture of multiprocessor systems. More specifically, on prediction and speculation in multiprocessor memory systems, synchronization in CMPs, power-aware cache-coherence protocols for CMPs, fault tolerance, and hardware transactional memory systems. He is a member of the IEEE.

José M. García received an MS degree in Electrical Engineering and a PhD degree in Computer Engineering both from the Technical University of Valencia. He is a professor of Computer Architecture at the Department of Computer Engineering at the University of Murcia (Spain), and also the Head of the Research Group on Parallel Computer Architecture. He has served as the Dean of the School of Computer Science for seven years. His current research interests lie in the design of power-efficient heterogeneous systems, and the development of data-intensive applications for those systems (especially bioinspired evolutionary algorithms, and bioinformatics apps for new drug discovery). He has published more than 140 refereed papers in different journals and conferences in these fields. He is a member of HiPEAC, the European Network of Excellence on High Performance and Embedded Architecture and Compilation. He is also a member of IEEE and ACM.