Adaptive Line Placement with the *Set Balancing Cache*

Dyer Rolán Basilio B. Fraguela Ramón Doallo

Depto. de Electrónica e Sistemas
Universidade da Coruña
A Coruña, Spain
{drolan, basilio, doallo}@udc.es

ABSTRACT

Efficient memory hierarchy design is critical due to the increasing gap between the speed of the processors and the memory. One of the sources of inefficiency in current caches is the non-uniform distribution of the memory accesses on the cache sets. Its consequence is that while some cache sets may have working sets that are far from fitting in them, other sets may be underutilized because their working set has fewer lines than the set. In this paper we present a technique that aims to balance the pressure on the cache sets by detecting when it may be beneficial to associate sets, displacing lines from stressed sets to underutilized ones. This new technique, called Set Balancing Cache or SBC, achieved an average reduction of 13% in the miss rate of ten benchmarks from the SPEC CPU2006 suite, resulting in an average IPC improvement of 5%.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*cache memories*

General Terms: Design, Performance

Keywords: cache, performance, adaptivity, balancing

1. INTRODUCTION

Memory references are often not uniformly distributed across the sets of a set-associative cache, the most common design nowadays [14]. As a result, at a given point during the execution of a program there are usually sets whose working set is larger than their number of lines (the associativity of the cache), while the situation in other sets is exactly the opposite. The outcome of this is that some sets exhibit large local miss ratios because they do not have the number of lines they need [9], while other sets achieve good local miss ratios at the expense of a poor usage of their lines, because some or many of them are actually not needed to keep the working set. An intuitive answer to this problem is to increase the associativity of the cache. Multiplying by n the associativity is equivalent to merging n sets in a single one, joining not only all their lines, but also their corresponding working sets. This allows to balance smaller working sets with larger ones, making available previous underutilized lines for the latter, which results in smaller miss rates. Unfortunately, increments in associativity impact negatively access latency and power consumption (e.g. more tags have to be read and compared in each access) as well as cache area, besides increasing the cost and complexity of the replacement algorithm. Worse, progressive increments in the associativity provide diminishing returns in miss rate reduction, as in general, the larger (and fewer) the sets are, the more similar or balanced their working sets tend to be. This way, only restricted levels of associativity are found in current caches.

In this paper we propose an approach to associate cache sets whose working set does not seem to fit in them with sets whose working set fits, enabling the former to make use of the underutilized lines of the latter. Namely, this cache design, which we call Set Balancing Cache or SBC, shifts lines from sets with high local miss rates to sets with underutilized lines where they can be found later. Notice that while an increase in associativity equates to merging sets in an indiscriminate way, our approach only exploits jointly the resources of several sets when it seems to be beneficial. Also, increases in associativity cannot choose which sets to merge, while the SBC can be implemented using either a static policy, which also preestablishes which sets can be associated, or a dynamic one that allows to associate a set with any other one. Thus, as we will see in the evaluation, the SBC achieves better performance than equivalent increases in associativity while not bringing their inconveniences.

The rest of this paper is organized as follows. Next section will describe the algorithm and structure of a static SBC, in which sets can only be associated with other sets depending on a preset condition on their index. Section 3 will introduce a dynamic SBC that allows to shift lines from a set that presents a bad behavior to the best set available (i.e. not yet associated) in the cache. Both SBC proposals will be evaluated using the environment described in Section 4, the results being discussed in Section 5. The cost of both approaches will be examined in Section 6. A deeper analysis of the cost and performance of the SBC is presented in Section 7. Related work will be discussed and compared in Section 8. The last section is devoted to the conclusions and future work.

2. STATIC SET BALANCING CACHE

We seek to reduce the pressure on the cache sets that are unable to hold all the lines in their working set, by displacing

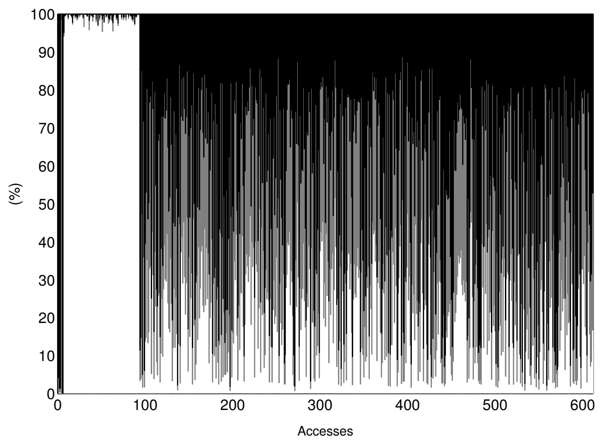

Figure 1: Distribution of the sets with a high saturation level(black), medium saturation level(gray) and low saturation level(white) in *473.astar*. Samples each 10^7K accesses.

some of those lines to sets that seem to have underutilized lines. These latter sets are those whose working set fits well in them, giving place to small local miss rates. This idea requires in the first place a mechanism to measure the degree to which a cache set is able to hold its working set. We call this value the saturation level of the set and we measure it by means of a counter with saturating arithmetic that is modified each time the set is accessed. If the access results in a miss, the counter is incremented, otherwise it is decremented. We call this counter saturation counter.

The fact that different sets can experience very different levels of demand has already been discussed in the bibliography [12][14]. This fact, which is the base for our proposal, can be illustrated with the saturation counters. Figure 1 classifies the sets in a 8-way 2MB cache with lines of 64 bytes during the execution of the *astar* benchmark, from the SPEC CPU2006 suite. The classification is a function of their saturation level as measured by saturation counters whose maximum value is 15 in this case. The levels of saturation considered are low (the counter is between 0 and 5), medium (between 6 and 10) and high (between 11 and 15). We can see how after the initialization stage there are some sets that are little saturated, while others are very saturated. These sets of opposite kinds could be associated, moving lines from highly saturated sets to little saturated ones in order to balance their saturation level and avoid misses. This also gives place to make second searches, or in general up to n-th searches if n sets are associated, if a line is not found in the set indicated by the cache indexing function and this set is known to have shifted lines to other set(s). As a result, the operation of the Set Balancing Cache we propose involves, besides the saturation counters explained, an *association algorithm*, which decides which set(s) are to be associated in the displacements, a *displacement algorithm* which decides when to displace lines to an associated set, and finally, modifications to the standard cache *search algorithm*. We will now explain them in turn.

2.1 Association algorithm

This algorithm determines to which sets can displace lines a given one. Although the number of sets involved could be any, and it could change over time, we have started studying the simplest approach, in which each cache set is statically associated to another specific set in the cache. That is the reason why we call this first design of our proposal static SBC (SSBC). This design minimizes the additional hardware involved as well as the changes required in the search algorithm of the cache. We have decided the associated set to be the farthest set of the considered one in the cache, that is, the one whose index is obtained complementing the most significant bit of the index of the considered set. This decision is justified by the principle of spatial locality, as if a given set is highly saturated, it is probable its neighbors are in a similar situation. A consequence of this decision is that given to sets X and Y associated by this algorithm, sometimes lines will be displaced from X to Y, and vice versa, depending on the state of their saturation counters. Notice also that when the associativity of a cache design is multiplied by 2, this is equivalent to merging in a single set the same two sets that our policy associates, i.e., those that differ in the most significant bit of the index.

2.2 Displacement algorithm

A first issue to decide is when to perform displacements. In order to minimize the changes in the operation of the cache and take advantage of line evictions that take place in a natural way in the sets, we have chosen to perform the displacements when a line is evicted from a highly saturated set. Since the replacement algorithm we consider for the cache sets is LRU, as it is the most extended one, this means that the LRU line will not be sent to the lower level of the memory hierarchy; rather it will be actually displaced to another set.

It is intuitive that displacements should take place from sets with a high saturation level to sets little saturated. A concrete range for the saturation counter, from which value of the counter we consider that displacements should take place, and under which value we consider a set to be little saturated are the parameters to choose for this policy. We have observed experimentally that a good upper limit for a saturation counter in a cache with associativity K is $2K-1$, thus the counters used in this paper work in the range 0 to $2K-1$.

Regarding the triggering of the displacement of lines from a set, when its saturation counter has a value under its maximum it means that there have been hits in the set recently, thus it is possible its working set fits in it. Only when the counter adopts its maximum value will have most recent accesses (and particularly the most recent one) resulted in misses and it is safer to presume that the set is under pressure. Thus our SBC only tries to displace lines from sets whose saturation counter adopts its maximum value, a decision taken based on our experiments.

Finally, although it is the association algorithm responsibility to choose which is the set that receives the lines in a displacement, it is clear that displacing lines to such set if/when its saturation counter is high can be counterproductive, since that indicates the lack of underutilized lines. In fact we could end up saturating a set that was working fine when trying to solve the problem of excess of load on another set. Thus a second condition required to perform a displacement is that the saturation counter of the receiver is below a given limit we call *displacement limit*. We have determined experimentally that the associativity K of the cache is a good displacement limit for the counters in the range

0 to $2K - 1$ we have used. Notice that since displacements only take place as the result of line evictions, the access to the associated set saturation counter needed to verify this second condition can be made during the resolution of the miss that generates the eviction.

Concerning the local replacement algorithm of the set that receives the displaced line, the line is inserted as the most recently used one (MRU). The rationale is that since the displaced line comes from a stressed working set, while the working set of the destination set fits well in it, this line needs more priority than the lines already residing in the set. Besides this way n successive displacements from a set to another one insert n different lines in the destination set. If the displaced line were inserted as the least recently used one (LRU), each new displacement would evict the line inserted in the previous one. We have checked experimentally that the insertion in the MRU position yields better results than in the LRU one.

2.3 Search algorithm

In the SBC a set may hold both memory lines that correspond to it according to the standard mapping mechanism of the cache and lines that have been displaced from its associated set. Thus the unambiguous identification of a line in a set requires not only its tag, but also an additional *displaced bit* or d for short. This bit marks whether the line is native to the set, when it is 0, or it has been displaced from another set, when it is 1. Searches always begin examining the set associated by default to the line, testing for tag equality and $d = 0$. If the line is not found there, a second search is performed in the associated set, this time seeking tag equality and $d = 1$. If the second search is successful, a secondary hit is obtained.

Our proposal avoids unnecessary second searches by means of an additional *second search* (sc) bit per set that indicates whether its associated set may hold displaced lines. This bit is set when a displacement takes place. Its deactivation takes place when the associated set evicts a line, if the OR of its d bits changes from 1 to 0 as result of the eviction. Checking this condition and resetting the second search bit of the associated set is done in parallel with the resolution of the miss that generates the eviction. Without this strategy to avoid unnecessary second searches, the IPC for the static SBC (SSBC) would have been 0.6% and 1.0% smaller in the two level and the three level cache configurations used in our evaluation in Section 5, respectively.

2.4 Discussion

Figure 2 shows a simple example of the operation of a Set Balancing Cache with 4 sets. Line addresses of 7 bits are used for simplicity, the lower two bits being the set index and the upper 5 ones the tag. The first reference is mapped to set 0, where $sc = 0$, thus no second search is needed and a miss occurs. Checking saturation counters results in a displacement of the line that must be evicted from set 0, here the one with tag 10010, to set 2 ($\overline{0}0 = 10$), so it is actually the LRU line of set 2 the one that is evicted from the cache. The second reference is mapped again to set 0, where it misses. Since now its sc bit is 1, a second search is performed in set 2, where there is a hit, since the tag is found with the displaced bit $d = 1$.

As Section 2.1 explains, a K-way SSBC associates exactly each pair of sets of the cache that would have been merged in

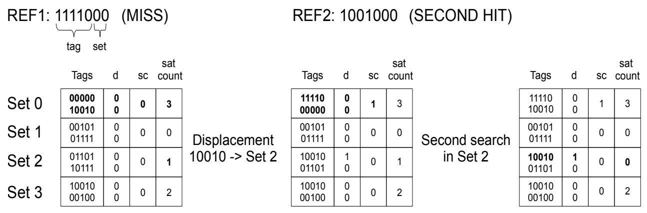

Figure 2: Static SBC operation in a 2-way cache with 4 sets. The upper tag in each set is the most recently used and the saturation counters operate in the range 0 to 3.

a single set in the $2K$-way cache with the same size and line size. Still, there are very important differences between both caches. While the $2K$-way cache unconditionally merges the sets and their working sizes, in the SSBC the merging is conditioned by the behavior of the sets. Namely, their resources are shared only when at least one of the sets suffers a stream of accesses with so many misses that its saturation counter reaches the maximum limit, while the other set shows to be large enough to hold its current working set, which is signaled by a value of its saturation counter smaller than the displacement limit. This smarter management of the sharing of resources in the cache leads to better performance for the SSBC even when it leads to second accesses when lines have been displaced from their original sets.

Contrary to other cache designs that lead to sequential searches in the cache [1][3] the SBC does not swap lines to return them to their original set when they are found displaced in another set. This simplifies management and does not hurt performance because our proposal, contrary to those ones, is oriented to non first level caches. Thus once a hit is obtained in a line, the line is moved to the upper level of the memory hierarchy, where successive accesses can find it. Experiments performing swapping of lines in the SBC to return displaced lines to their original set under a hit proved that this policy had a negligible impact on performance.

Finally, in principle, the tag and data arrays of an SBC can be accessed in parallel. Still, we recommend and simulate a sequential access to these arrays for two reasons. One is that the SBC is oriented to non-first level caches, where both arrays are often accessed sequentially because in those caches the tag-array latency is much shorter than the data-array one, and the sequential access is much more energy-efficient than the parallel one [5][17]. The other is that since the SBC may lead to second searches, the corresponding parallel data-array accesses would further increase the waste of energy.

3. DYNAMIC SET BALANCING CACHE

The SSBC is very restrictive on associations. Each set only relies on another prefixed set as potential partner to help keep its working set in the cache. It could well happen that both sets were highly saturated while others are underutilized. When a cache set is very saturated, it would be better to have the freedom to associate it to the more underutilized (i.e. with the smallest saturation value) non-associated set in the cache. This is what the dynamic SBC (DSBC) proposes. We now explain in turn the algorithms of this cache.

3.1 Association algorithm

The DSBC triggers the association of sets when the saturation counter of a set that is not associated with another set reaches its maximum value, which is $2K-1$ in our experiments, where K is the associativity of the cache. When this happens, the DSBC tries to associate it with the available set (i.e. not yet associated with another one) with the smallest saturation level. An additional restriction is that the association will only take place if this smallest saturation level found is smaller than the displacement limit, described in Section 2.2. The reason is that it makes no sense to consider as candidate for association a set whose saturation counter indicates that lines from other sets should not be displaced to it.

In principle this policy would require hardware to compare the saturation counters of all the available sets in order to identify the smallest one. Instead we propose a much simpler and cheaper design that yields almost the same results, which we call *Destination Set Selector* (DSS). The DSS has a small table that tries to keep the data related to the less saturated cache sets. Each entry consists of a valid bit, which indicates whether the entry is valid, the index of the set the entry is associated to, and the saturation level of the set. Comparers combined with multiplexes in a tree structure allow to keep updated a register *min* with the minimum saturation level stored in the DSS (*min.level*), as well as the number of its DSS entry (*min.entry*) and the index of the associated set (*min.index*). This register provides the index of the best set available for an association when requested. Similarly, a register *max* with the maximum saturation counter in the DSS (*max.level*) and the number of the DSS entry (*max.entry*) is kept updated. The role of this register is to help detect when sets not currently considered in the DSS should be tracked by it, which happens when their saturation level is below *max.level*.

When the saturation counter of a free set (one that is not associated to another set) is updated, the DSS is checked in case it needs to be updated. The index of this set is compared in parallel with the indices in the valid entries of the DSS. Under a hit, the corresponding entry is updated with the new saturation level. If this value becomes equal to the displacement limit, the entry is invalidated, since sets with a saturation level larger or equal to this limit are not considered for association. If the set index does not match any entry in the DSS and its saturation level is smaller than *max.level*, this set index and its saturation value are stored in the DSS entry pointed by *max.entry*; otherwise they are dismissed.

Any change or invalidation in the entries of the table of the DSS lead to the update of the *min* and *max* registers. Invalidations take place when the saturation value reaches the displacement limit or when the entry pointed by *min* is used for an association. In this latter case the saturation value of the entry is also set to the displacement limit. This ensures that all the invalid entries have the largest saturation values in the DSS. Thus whenever there is at least an invalid entry, *max* points to it and *max.entry* equals the displacement limit, which is the limit to consider a set for association with a highly saturated set.

The operation of the DSS allows to provide the best candidate for association to a highly saturated set most of the times. The main reason why it may fail to do this is because all its entries may be invalidated in the moment the association is requested. When this happens no association takes place. Obviously, the larger the number of entries in the DSS, the smaller the probability this situation arises. The efficiency of the DSS as a function of its number of entries will be analyzed in Section 5.

The DSBC has a table with one entry per set called *Association Table* (AT) that stores in the i-th entry $AT(i).index$, the index of the set associated with set i, and a source/destination bit $AT(i).s/\overline{d}$ that indicates in case of being associated, whether the set triggered the association because it became saturated ($s/\overline{d}=1$) or it was chosen by the Destination Set Selector to be associated because of its low saturation ($s/\overline{d}=0$). When a set is not associated, its entry stores its own index and $s/\overline{d}=0$.

3.2 Displacement algorithm

Just as in the SSBC, displacements take place when lines are evicted from sets whose saturation counter has its maximum value. In the DSBC, sets are not associated by default to any other specific set, thus another condition for the displacements to take place is that the saturated set is associated to another set. Another important difference with respect to the SSBC is that displacements are unidirectional, that is, lines can only be displaced from the set that requested the association (the one whose counter reached its maximum value), which we call source set, to the one that was chosen by the Destination Set Selector to be associated to it, which we call destination set. The rationale is that the destination set was chosen among all the ones in the cache to receive lines from the source one because of its low level of saturation. For the same reason, displacements will not depend on the level of saturation of the destination set: once it is designated as destination set, it will continue to receive lines displaced from the source until the association is broken. If the same policy as in the SSBC were applied, that is, displacements only take place when the destination set saturation counter is smaller than K, the average miss rate in our experiments would have been on average 0.6% larger, and the resulting IPC would have been 0.38% worse.

3.3 Search algorithm

Just as in the SSBC, there is a displaced bit d per line that indicates whether is has been displaced from another set. The cache always begins a search looking for a line with the desired tag and $d=0$ in the set with the index i specified by the memory address sought. Simultaneously the corresponding i-th entry in the Association Table, $AT(i)$ is read. Upon a hit, the LRU of the set (and the dirty bit if needed) is modified. Otherwise, the access is known to have resulted in a miss if $AT(i).s/\overline{d}=0$, as this means that either the set is not associated or this set is the destination set of an association, which cannot displace lines to its associated set. In any case the saturation counter is updated and if it has reached its maximum and the set is not yet associated, a destination set can be requested from the Destination Set Selector while the miss is resolved.

If $AT(i).s/\overline{d}=1$ the destination set indicated by $AT(i).index$ is searched for an entry with the tag requested and $d=1$. Here we can get a secondary hit or a definitive miss. In both cases the set saturation counter will be updated, although this will not influence the association. If there is a miss, the LRU line of the destination set will be evicted, and the LRU line from the source set will be moved to the destination set to replace it. This happens in parallel with the resolution of the miss, whose line will be inserted in the source set.

3.4 Disassociation algorithm

The approach followed to break associations is very similar to the one used to avoid unnecessary second searches in the SSBC. A disassociation can take place upon a first search miss (i.e., a native miss) in a destination set i. If the OR of the d bits of this set changes from 1 to 0 as result of the eviction triggered by the miss, the association is broken. This can be calculated once the line to be evicted is decided, as this condition is equivalent to requiring that the OR of the d bits of all the lines but the one to evict is 0. This way, the detection of the disassociation and the changes it involves take place in parallel with the eviction itself and the resolution of the miss. The disassociation requires accessing the AT of the source set of the association, as provided by $AT(i).index$, and clearing the association there. The entry for the destination set is then also modified setting $AT(i).index = i$.

3.5 Discussion

Figure 3 shows an example of the DSBC operation with the same references as in Figure 2. The first reference is mapped to set 0, where a miss occurs. Since this set is not associated ($AT(0).index = 0$) but its saturation counter has its maximum value, a destination set for an association is requested. The figure assumes the Destination Set Selector provides set 3 as candidate, proceeding then to evict the LRU line in set 3 to replace it with the LRU line in set 0. When the line missed arrives from memory it is stored in the block that has been made available in set 0. The second reference is mapped again to the set 0 resulting in a miss. A second search is initiated in set $AT(0).index = 3$, where it is found.

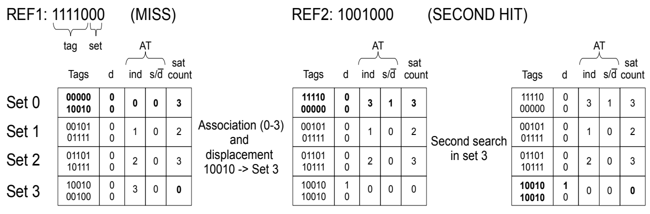

Figure 3: Dynamic SBC operation in a 2-way cache with 4 sets. The upper tag in each set is the most recently used and the saturation counters operate in the range 0 to 3.

The greater flexibility of the DSBC allows it to apply a more aggressive displacement policy, as Section 3.2 explains. Section 5 will show it also achieves better results. Beyond performance measurements, graphical representations also help explain the net effect of SBC on a cache. Figure 4 illustrates it showing the distribution of the saturation level across the sets of the L2 cache of the two-level cache configuration of Table 1 during part of the execution of the *omnetpp* benchmark of the SPEC CPU 2006 suite. The level is measured with a saturation counter in the range 0 to 15. The baseline in Figure 4(a) has a high ratio of highly (level 11 to 15) and lowly (level 0 to 5) saturated sets. The SSBC in Figure 4(b) basically turns highly-saturated sets into medium-saturated sets. The DSBC in Figure 4(c) alleviates more highly-saturated sets without generating medium-saturated sets.

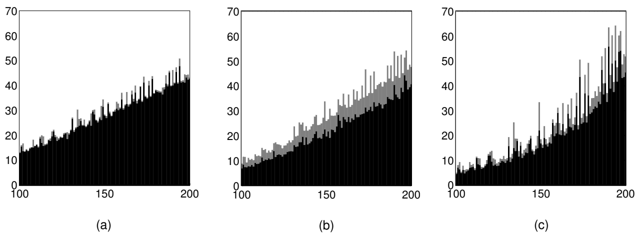

Figure 4: Distribution of the sets with a high saturation level(black), medium saturation level(gray) and low saturation level(white) during a portion of the execution of omnetpp in the L2 cache of the two-level configuration. (a) Baseline (b) Static SBC (c) Dynamic SBC. Samples each $5 * 10^5$K accesses.

4. SIMULATION ENVIRONMENT

To evaluate our approach we have used the SESC simulator [15] with two baseline configurations based on a four-issue CPU clocked at 4GHz, with two and three on-chip cache levels respectively. Both configurations assume a 45 nm technology process and are detailed in Table 1. The tag check delay and the total round trip access are provided for the L2 and L3 to help evaluate the cost of second searches when the SBC is applied. Our three-level hierarchy is somewhat inspired in the Core i7 [8], the L3 being proportionally smaller to account for the fact that only one core is used in our experiments. Both configurations allow an aggressive parallelization of misses, providing between 16 and 32 Miss Status Holding Registers per cache.

4.1 Benchmarks

We use 10 representative benchmarks of the SPEC CPU 2006 suite, both from the INT and FP sets. They have been executed using the reference input set (*ref*), during 10 billion instructions after the initialization. Table 2 characterizes them providing the number of accesses to the L2 during the 10^{10} instructions simulated, the miss rate in the L2 cache both in the two-level (2MB L2) and the three-level (256kB) configurations, and whether they belong to the INT or FP set of the suite. It is a mix of benchmarks that vary largely both in number of accesses that reach the caches under the first level and in miss ratios in the L2 cache.

5. PERFORMANCE EVALUATION

The SBC has been applied, for both static and dynamic versions, in the second level for the two-level configuration and in the two lower levels for the three-level configuration. The dynamic SBC uses a Destination Set Selector (described in Section 3.1) with four entries based on our experiments, which we detail next.

Table 1: Architecture. In the table RT, TC and MSHR stand for round trip, tag directory check and miss status holding registers, respectively.

Processor	
Frequency	4GHz
Fetch/Issue	6/4
Inst. window size	80 int+mem, 40 FP
ROB entries	152
Integer/FP registers	104/80
Integer FU	3 ALU,Mult. and Div.
FP FU	2 ALU, Mult. and Div.
Common memory subsystem	
L1 i-cache & d-cache	32kB/8-way/64B/LRU
L1 Cache ports	2 i/ 2 d
L1 Cache latency (cycles)	4 RT
L1 MSHRs	4 i / 32 d
System bus bandwidth	10GB/s
Memory latency	125ns
Two levels specific memory subsystem	
L2(unified) cache	2MB/8-way/64B/LRU
L2 Cache ports	1
L2 Cache latency (cycles)	14 RT, 6 TC
L2 MSHR	32
Three levels specific memory subsystem	
L2(unified) cache	256kB/8-way/64B/LRU
L3(unified) cache	2MB/16-way/64B/LRU
Cache ports	1 L2, 1 L3
L2 Cache latency (cycles)	11 RT, 4 TC
L3 Cache latency (cycles)	39 RT, 11 TC
MSHR	32 L2, 32 L3

Table 2: Benchmarks characterization. MR stands for miss rate.

Bench	L2 Accesses	2MB L2 MR	256kB L2 MR	Comp.
bzip2	125M	9%	41%	INT
milc	255M	71%	75%	FP
namd	63M	2%	5%	FP
gobmk	77M	5%	10%	INT
soplex	105M	8%	15%	FP
hmmer	55M	10%	41%	INT
sjeng	32M	26%	27%	INT
libquantum	156M	74%	74%	INT
omnetpp	100M	28%	91%	INT
astar	192M	23%	48%	INT

5.1 Destination Set Selector efficiency

A request for a destination set made to the Destination Set Selector (DSS) may result in four outcomes. If the DSS provides a candidate, this cache set can (A) actually have the smallest level of saturation among the available sets in the cache or (B) not. The DSS will not provide a candidate if all its entries are invalid. This may happen either because (C) there are actually no candidates in the cache (all the sets are either associated or too saturated), or (D) there are candidates in the cache, but not in the DSS. Figure 5 shows the evolution of the average percentage of times each one of these four situations happens during the execution of our benchmarks in the L2 cache of the two-level configuration as the number of entries in the DSS varies from 2 to 128. The outcomes are labeled A, B, C and D, following our explanation. We see that even with just two entries the DSS has a quite good behavior, since outcomes A and C, in which the DSS works as well as if it were tracking the behavior of all the sets, add up to 80%. With 4 entries A+C behavior improves to 90%, and after that there is a slow slope until almost 100% of the outcomes are either A or C with 128. Based on this we have chosen a 4-entry DSS to optimize the balance between hardware and power required and benefit achieved. In this graph we can also see that under the conditions requested in the DSBC, around 35% of the association requests are satisfied.

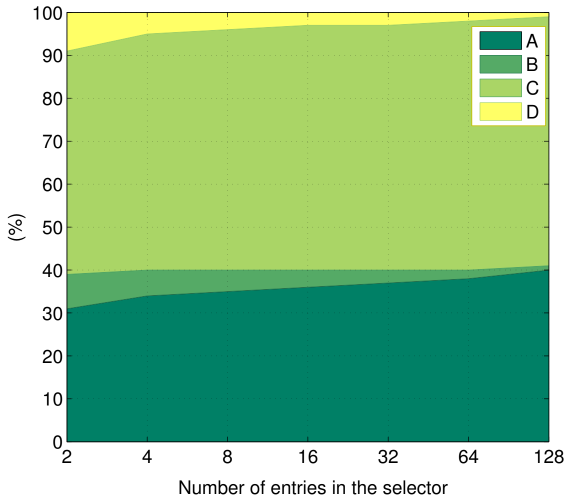

Figure 5: Percentage of association requests made to the Destination Set Selector (DSS) in the L2 cache of the two-level configuration that (A) are satisfied with a set with the minimum level of saturation, (B) are satisfied with a set whose level of saturation is not the minimum available, (C) are not satisfied because there are no candidate sets in the cache, and (D) are not satisfied because none of the existing candidate sets is in the DSS, depending on the number of entries in the DSS.

5.2 Performance comparison

Figure 6 shows the ratio of accesses that result in a miss, a hit, and a secondary hit in the L2 and L3 caches in the two memory hierarchies tested, using standard caches, SSBC, and DSBC for each one of the benchmarks analyzed. The last group of columns (mean), represents the arithmetic mean of the rates observed in each cache. We can see that the SBCs basically keep the same ratio of first access hits as a standard cache, and they turn a varying ratio of the misses into secondary hits. When the baseline miss rate is small or there are few accesses, the SBCs seldom perform displacements of lines and second searches happen also infrequently. Also, the DSBC achieves better results than the SSBC, as expected. Hit and miss rates are not the best characterization for SBCs because they involve second searches that make secondary hits more expensive than first hits, and which delay the resolution of misses that need the second search to be confirmed. This is better measured in Figure 7, which shows the average data access time improvement of the static and dynamic SBC with respect to the baseline caches for each benchmark.

Despite the overhead of the second searches, the SBC almost never increases the average access time of any benchmark. There is only a small 1% slowdown in the L2 cache in the two-level configuration for 444.namd and 445.gobmk,

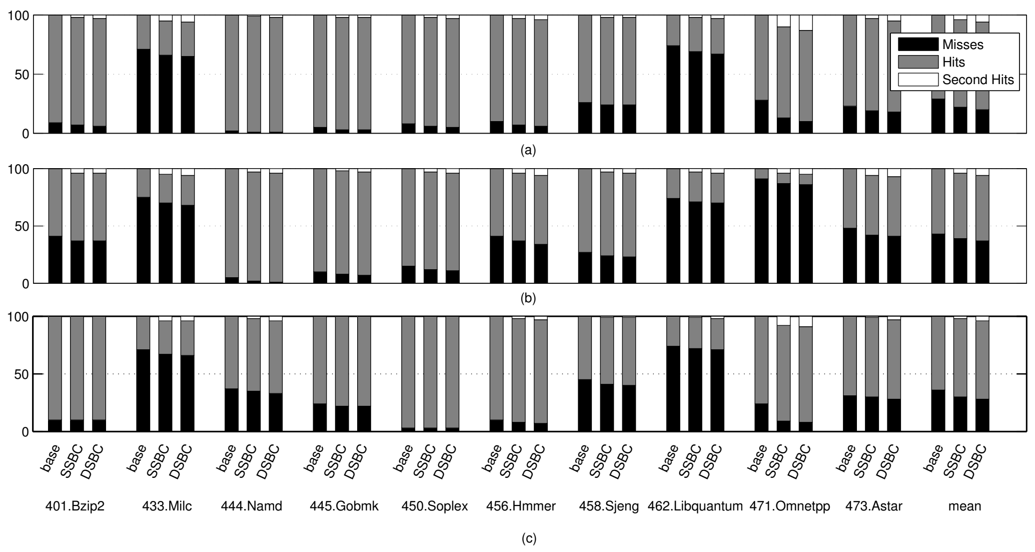

Figure 6: Miss, hit and secondary hit rates for the (a) L2 cache in the two-level configuration, (b) L2 cache in the three-level configuration, and (c) L3 cache in the three-level configuration.

because their second searches contribute very little to reduce its already minimal miss rate. Not surprisingly the greater flexibility of the DSBC allows it to choose better suited cache sets for the displacements than the SSBC, leading to better average access times. The average improvement (geometric mean) of the access time in the L2 of our two-level configuration is 4% and 8% for the SSBC and the DSBC, respectively. In the three-level configuration the average reduction is 3% and 6% for the L2, and 10% and 12% for the L3, for the SSBC and the DSBC, respectively.

Figures 8 and 9 show the performance improvement in terms of instructions per cycle (IPC) for each benchmark in the two level and the three level configurations tested, respectively. The figures compare the baseline not only with the SSBC and the DSBC, but also with the baseline system where the L2 and the L3 have duplicated their associativity. This latter configuration is tested to show the difference between associating two sets of K lines following the SBC strategy and using sets of $2K$ lines. The bar labeled *geomean* is the geometric mean of the individual IPC improvements seen by each benchmark.

In the two-level configuration the SBC always has a positive or, at worst, negligible effect on performance. Two kinds of benchmarks get no benefit from the SBC: those with a

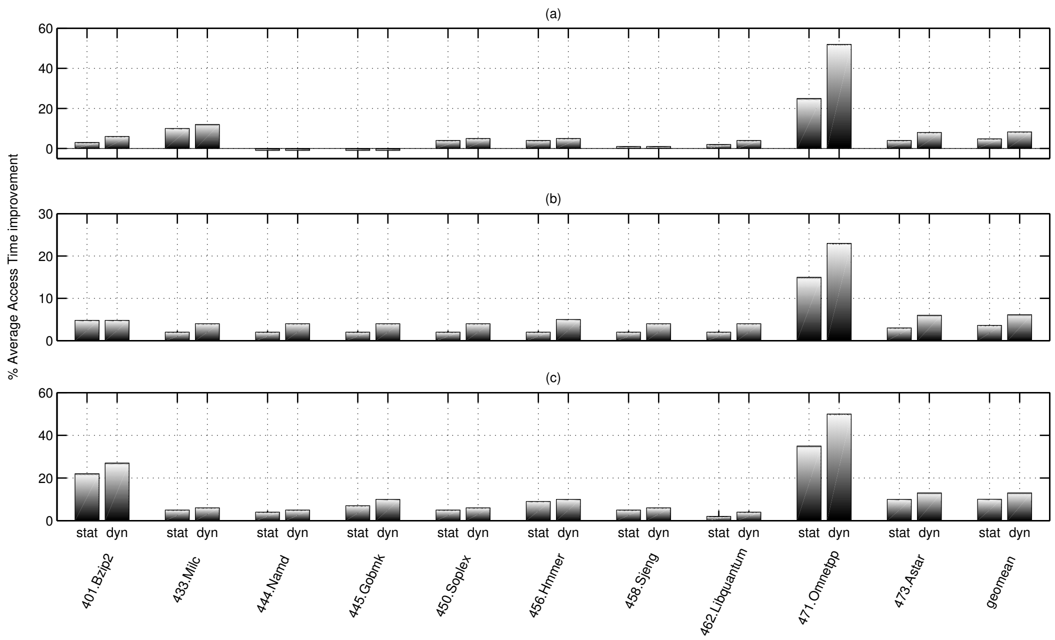

Figure 7: Average access time reduction achieved by the static and the dynamic SBC in the (a) L2 cache in the two level configuration, (b) L2 cache in the three level configuration, and (c) L3 cache in the three level configuration.

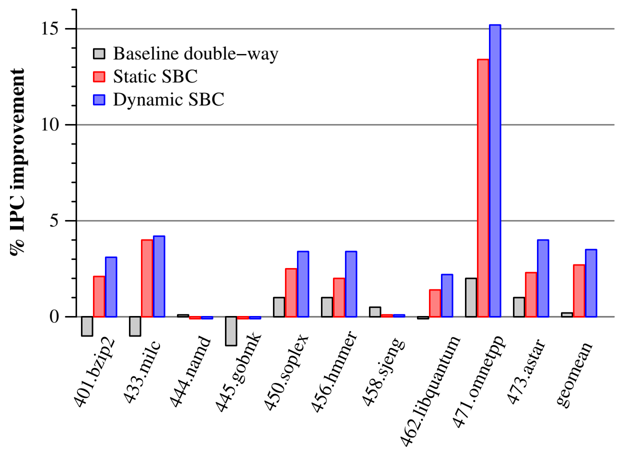

Figure 8: Percentage IPC improvement over the baseline in the two-level configuration duplicating the L2 associativity or using SBC.

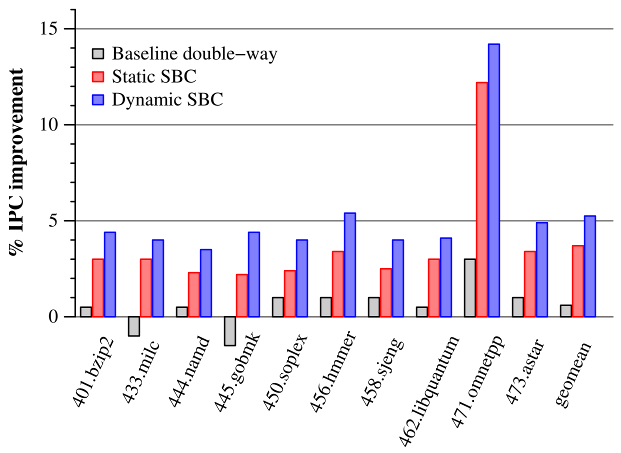

Figure 9: Percentage IPC improvement over the baseline in the three-level configuration duplicating the L2 and L3 associativity or using SBC in both levels.

small miss rate, like 444.namd or 445.gobmk, in which our proposal can do little to improve an already good cache behavior; and 458.sjeng, which has very few accesses to the L2, just 3.2 each 1000 instructions, as Table 2 shows. The small number of accesses reduces the influence of the L2 behavior in the IPC, and more importantly it reduces the frequency of triggering of the SBC mechanisms.

In the three-level configuration the improvement is larger and applies to all the benchmarks. The benchmarks that did not benefit from the SBC in the two-level configuration benefit now for two reasons. One is the larger local miss ratios either in the L2 or in the L3. The other is that in this 256 kB L2 cache (modeled after the one in the Core i7) the accesses are spread on 8 times less sets than in the 2MB cache of the two-level configuration. This increases the working set of each set, generating more SBC-specific activity. The DSBC systematically outperforms the SSBC, which in its turn achieves much better results than duplicating the associativity of the caches. Since the SSBC associates exactly the two same sets that a duplication of the associativity merges, these results outline the benefit of sharing resources among sets under the control of a policy that triggers this sharing only when it is likely it is going to be beneficial and disables it when the feedback is not good.

6. COST

In this section we evaluate the cost of the SBC in terms of storage requirements, area and energy, which has been estimated using CACTI 5.3 [7].

The SBC requires additional hardware because of the need of a saturation counter per set to monitor its behavior and additional bits in the directory to identify displaced lines (d bit). The SSBC has an additional bit per set to know whether second searches are required. The DSBC instead requires an Association Table with one entry per set that stores a $s/\overline{d}$ bit to specify whether the set is the source or the destination of the association, and the index of the set it is associated to. It also requires a Destination Set Selector (DSS) to choose the best set for an association, a 4-entry DSS being used in our evaluation. Based on this, Table 3 calculates the storage required for a baseline 8-way 2 MB cache with lines of 64B assuming addresses of 42 bits. As we can see, the SSBC and the DSBC only have an overhead of 0.31% and 0.58% respectively, compared to the baseline configuration. The energy consumption overhead on average per access calculated by CACTI is less than 1% for SBC and 79% for the baseline with double associativity, and the corresponding area overhead is shown in Table 4. We see that the SBC not only offers more performance, but also requires less energy and area than duplicating the associativity.

Table 3: Baseline and SBC storage cost in a 2MB/8-way/64B/LRU cache. B stands for bytes.

	Base	Static SBC	Dynamic SBC
Tag-store entry:			
State(v+dirty+LRU+[d])	5 bits	6 bits	6 bits
Tag ($42 - \log_2 sets - \log_2 ls$)	24 bits	24 bits	24 bits
Size of tag-store entry	29 bits	30 bits	30 bits
Data-store entry:			
Set size	512B	512B	512B
Additional structs per set:			
Saturation Counters	-	4 bits	4 bits
Second search bits	-	1 bit	-
Association Table	-	-	12+1 bits
Total of structs per set	-	5 bit	17 bits
DSS (entries+registers)	-	-	10B
Tag-store entries	32768	32768	32768
Data-store entries	32768	32768	32768
Number of Sets	4096	4096	4096
Size of the tag-store	118.7kB	122.8kB	122.8kB
Size of the data-store	2MB	2MB	2MB
Size of additional structs	-	2560B	8714B
Total	**2215kB**	**2222kB**	**2228kB**

7. ANALYSIS

In this section we evaluate how the performance and cost of the SBC vary with respect to the parameters of the cache. We also analyze how it compares to the usage of a victim cache whose cost is comparable to the overhead of the SBC. Along all this section we will always use as baseline the 2MB/8-way/64B/LRU L2 cache of our two-level configuration.

Table 4: Baseline and SBC area. Percentages in the Total column are related to the Baseline configuration.

Configuration	Components	Details	Subtotal	Total
Baseline	Data + Tag	2MB 8-way 64B line size + tag-store	12,57 mm^2	**12,57** mm^2
Baseline with double associativity	Data + Tag	2MB 16-way 64B line size + tag-store	14,52 mm^2	**14,52** mm^2 (> 3%)
Static SBC	Data + Tag	2MB 8-way 64B line size + tag-store (with additional d bit)	12,58 mm^2	**12,60** mm^2 (< 1%)
	Counters	4096*4 bits	0,01 mm^2	
	Second search bits	4096 bits	$< 0,01$ mm^2	
Dynamic SBC	Data + Tag	2MB 8-way 64B line size + tag-store (with additional d bit)	12,58 mm^2	**12,64** mm^2 (< 1%)
	Counters	4096*4 bits	0,01 mm^2	
	Association Table	4096*12 bits	0,04 mm^2	
	DSS (entries + regs)	4*(1+12+4)+2*(2+4) bits	$< 0,01$ mm^2	

Table 5: Cost-benefit analysis of the static and the dynamic SBC as a function of the cache size.

Cache size	Baseline miss rate	SSBC miss rate	DSBC miss rate	SSBC miss rate reduction	DSBC miss rate reduction	SSBC storage overhead	DSBC storage overhead
256KB	45.13%	40.81%	39.24%	9.6%	13.1%	0.28%	0.51%
512KB	39.07%	35.54%	34.47%	9.2%	11.8%	0.29%	0.53%
1MB	33%	30.84%	29.14%	6.55%	9.3%	0.30%	0.55%
2MB	25.6%	23.25%	22.3%	9.2%	12.8%	0.30%	0.58%
4MB	20.7%	19.6%	19.3%	5.4%	6.8%	0.31%	0.60%

Table 6: Cost-benefit analysis of the static and the dynamic SBC as a function of the line size.

Line size	Baseline miss rate	SSBC miss rate	DSBC miss rate	SSBC miss rate reduction	DSBC miss rate reduction	SSBC storage overhead	DSBC storage overhead
64B	25.6%	23.25%	22.31%	9.2%	12.8%	0.30%	0.58%
128B	27.46%	25.6%	25%	6.5%	9%	0.16%	0.29%
256B	24.6%	23.2%	22.3%	5.7%	9.3%	0.08%	0.14%

Table 7: Cost-benefit analysis of the static and the dynamic SBC as a function of the associativity.

Line size	Baseline miss rate	SSBC miss rate	DSBC miss rate	SSBC miss rate reduction	DSBC miss rate reduction	SSBC storage overhead	DSBC storage overhead
8-ways	25.6%	23.25%	22.31%	9.2%	12.8%	0.30%	0.58%
16-ways	25.1%	22.8%	21.88%	9.2%	12.83%	0.26%	0.38%
32-ways	24.6%	22.28%	21.43%	9.4%	12.88%	0.23%	0.29%

7.1 Impact of varying cache parameters

Table 5 shows the miss rate reduction achieved by the static and the dynamic SBC as well as the storage overhead it involves as the cache size varies between 256kB and 4MB. Both kinds of SBC always reduce the average miss rate obtained, but as the cache size increases the working set of some benchmarks fits better, reducing the opportunities of improving it.

Table 6 studies the cost-benefit of both SBC proposals comparing the miss rate reduction achieved by them versus the additional storage cost they incur as a function of the line size in the baseline cache. The increase in the line size reduces proportionally both the number of sets and lines, being the SBC cost mostly proportional to it as we can see. The reduction of the number of sets and the fact their lines keep more data also makes more probable the SSBC finds the static pairs of sets it is able to associate are too saturated to trigger displacements. The greater flexibility of the DSBC allows to overcome better this problem. This is why the DSBC behaves better than the SSBC as the line size increases.

Table 7 makes the same study from the point of view of the associativity considering values of 8, 16 and 32. The increase of associativity reduces the number of sets, and thus the relative cost of the SBC, but increases the tag size. Miss rates and their reduction stay very flat. Also, just as the experiments in Section 5.2 considering caches that duplicated the associativity, this table shows that making shared usage of the lines of two sets under heuristics like the ones proposed by the SBC is much more effective than organizing the cache lines in sets with twice or even four times larger. This way, even the 8-way static and dynamic SBC have 5.5% and 9.31% less misses than the 32-way baseline, respectively.

7.2 Victim cache comparison

Figure 10 shows a comparison of the L2 cache miss rates among a static SBC, a dynamic SBC, and the cache extended with a fully-associative victim cache [10] of either 8kB or 16kB of data store, relative to the L2 two-level base-

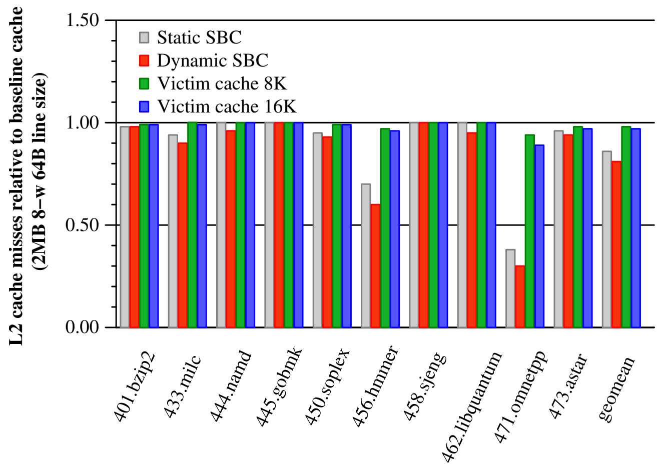

Figure 10: Comparison of the static SBC, the dynamic SBC, a victim cache of 8kB and a victim cache of 16kB in terms of miss rate relative to the one in the L2 in the two-level baseline configuration.

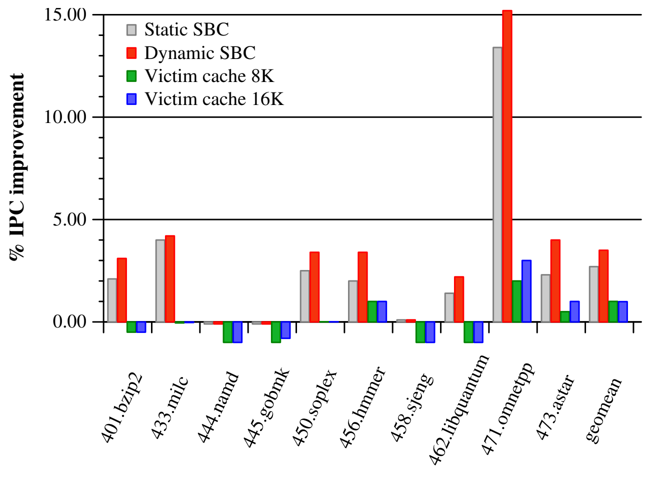

Figure 11: Comparison of the static SBC, the dynamic SBC, a victim cache of 8kB and a victim cache of 16kB in terms of IPC relative to the one in the L2 in the two-level baseline configuration.

line configuration. We have chosen these sizes because as Table 3 shows, the storage overhead for the L2 cache configuration considered is about 7kB for the static SBC and about 13 kB for the dynamic SBC. Thus, the 8kB and the 16kB victim caches are larger than the static and the dynamic SBC respectively. If their tag-store were considered too, they would be even more expensive in comparison. We see how with less resources, any SBC performs better than the largest victim cache. Figure 11 makes the same comparison based on the IPC.

7.3 SBC Behavior

While comparisons in miss rate, average access time or IPC allow to assess the effectiveness of the SBC with respect to other designs, measurements on its internal behavior allow to understand better how it achieves these results. Thus, we analyze here this behavior based on measurements in the L2 cache of our two-level configuration. This way, the hit rate observed in the second searches, that is, the ratio of second searches that result in a secondary hit, is on average 36.3% and 47.7% for the SSBC and DSBC, respectively. The SSBC is more conservative in displacing lines than the DSBC because of its restriction on the associated set. As a result, less lines are displaced, leading to a smaller second access hit ratio. In fact the SSBC displaces an average of 1.7 lines per association (i.e. since the *sc* bit in an association is activated until it is reset), while the DSBC displaces an average of 2.15 lines before the association is broken. On the other hand, the conservative policy of the SSBC leads it to make safer decisions than the DSBC on which lines it is interesting to displace to the associated sets, that is, the lines it displaces are more likely to be referenced again. The result of this is that the average number of secondary hits per line displaced is 3.64 in the SSBC, while it decreases to 3.29 in the DSBC.

It is also interesting to examine the frequency of second searches, as they may generate contention in the tag-array. On average only 10.3% and 10.2% of the accesses to the cache require second searches in the SSBC and the DSBC, respectively.

8. RELATED WORK

There have been several proposals to improve the architecture of caches to deal with the problem of the non-uniform distribution of memory accesses across the cache sets. Alternative indexing functions have been suggested [16][11] that succeed at achieving a more uniform distribution, but they do not attempt to identify underutilized lines or working sets that cannot be retained successfully in the cache. The idea underlying most proposals to improve the capability of the caches to keep the working set is the increase with respect to the standard design of the number of possible places where a memory block can be placed. In general, the smaller the associativity of the cache, the greater the imbalance in the demand on the individual set of the cache. Thus it was in the context of direct-mapped caches where the first approaches of this kind appeared. Pseudo-associative caches belong to this family of proposals. Initially they provided the possibility of placing the blocks in a second associated cache line, providing a performance similar to that of 2-way caches [1][3], but they were also generalized to provide larger associativities [19]. These proposals present search structures based at the line level as they perform searches line by line, unlike SBC that performs searches set by set. Besides they do not provide mechanisms to inhibit line displacements: whenever a cache line is occupied by a memory block mapped to it and a second memory block of this kind is requested, there is an automatic displacement to an associated cache line. In our proposal this depends on the value of the saturation counters, and in the case of the dynamic SBC, whether there is associated set or not, and the set is considered a source or a destination of lines. Another important difference is that pseudo-associative caches swap cache lines under non-first hits in order to place them back in their major location according to the default mapping algorithm of the cache, so that successive searches will find them in the first search. The SBC performs no swaps because it is oriented to non-first level caches, thus the effect of successive accesses is blurred, as many will be satisfied from the upper levels in the memory hierarchy.

The adaptive group-associative cache (AGAC) [12] tries to detect underutilized cache frames in direct-mapped caches in order to retain in them some of the lines that are to be replaced. Contrary to the SBC, AGAC records the location of each line that has been displaced from its direct-mapped position in a table, which is accessed in parallel with the tag-storage. Besides, AGAC needs multiple banks to aid the swappings triggered by hits on displaced lines. Also, the decision on what to do with a line on a miss in its location depends on whether it is among the most recently used ones or not. If it has been recently used, it is displaced to a location that is not among the most recently used or displaced ones, the selection being then random in that subset.

The Indirect Index Cache (IIC) [6] seeks maximum flexibility in the placement of memory blocks in the cache. Its tag-store entries keep pointers so that any tag-entry can be associated to any data-entry. The tag-store is made up of a primary 4-way associative hash table that under a miss proceeds to the traversal of the collision set for its hash entry in a direct-mapped collision table. Each entry of this table points to the next entry in the collision set. The IIC swaps entries from the collision table to the primary hash table to speed up future accesses, resulting in increased port contention and power consumption. Finally, the IIC management algorithms are much more complex than those of the SBC, in particular the generational replacement run by software.

The NuRAPID cache [4] provides a flexible placement of the data-entries in the data array in order to reduce average access latency, allowing the most recently used lines to be in the fastest subarrays in the cache. This requires decoupling data placement and tag placement, which NuRAPID achieves through the usage of pointers between tag-entries and data-entries. This flexibility does not exist in the tag array, which is completely conventional in its mapping and replacement policies. Thus NuRAPID does not target miss rate and has the same problems of workload imbalance among sets as a standard cache.

The B-Cache [18] tries to reduce conflict misses balancing the accesses to the sets of first-level direct-mapped caches by increasing the decoder length and incorporating programmable decoders and a replacement policy to the design. There is no explicit notion of pressure on the sets or displacements between them as a result of it.

The V-Way cache [14] adapts to the non-uniform distribution of the accesses on the cache sets by allowing different cache sets to have a different number of lines according to their demand. It duplicates the number of sets and tag-store entries, keeping the same associativity and number of data lines. Data lines are assigned dynamically to sets depending on the access pattern of the sets and a global replacement algorithm on the data lines. Namely, the V-Way cache reassigns the less reused data lines to sets with empty tag-store entries that suffer a miss, which is the origin of the variability of the set sizes. When a set reaches its maximum size, it stops growing and replacements take place under a typical replacement algorithm such as LRU. The structure to allow any data line to be assigned to any tag-entry requires the storage for forward and reverse pointers between the tag-store and the data-store entries, besides the reuse counters used by the global replacement algorithm. A comparison with IIC shows similar miss rate reductions, particularly for few (1 or 2) additional tag-store accesses per hit by the IIC. Finally, the V-Way cache outperforms largely AGAC in their tests.

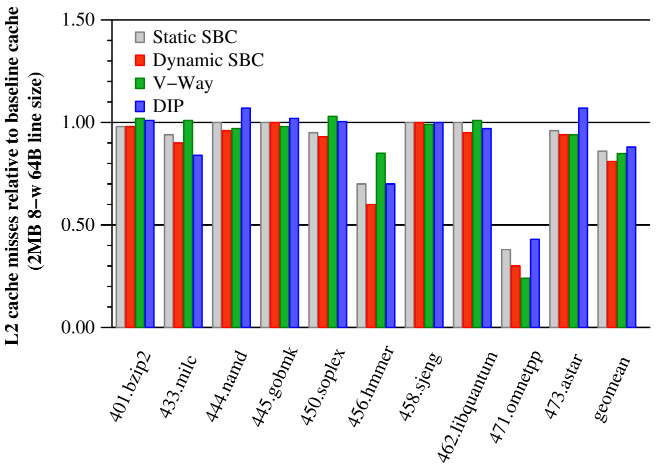

Figure 12: Comparison with recent proposals in terms of number of cache misses relative to the L2 cache of our two-level configuration.

More recently, Scavenger [2] has been proposed, which unlike SBC is exclusively oriented to last level caches and partitions the cache in two halves. One half is a standard cache, while the other half is a large victim file (VF) organized as a direct-mapped hash table with chaining, in order to provide full associativity. The VF tries to retain the blocks that miss more often in the conventional cache, which are identified by a skewed Bloom filter based on the frequency of appearance of each block in the sequence of misses. If a block evicted from the standard cache is predicted by the filter to have more misses than the block with the smaller priority in the VF, this latter block is replaced by the one evicted from the standard cache. This policy requires a priority queue that maintains the priorities of all the VF blocks. Accesses take place in parallel in both halves of the cache. When a block is found in the VF, it is moved to the standard cache.

These two proposal achieve good results, but their cost is much larger than that of the SBC. This way, while the V-Way cache and Scavenger require about an additional 11% and 10% storage on the L2 cache of our two-level configuration, respectively, the overhead for the static and dynamic SBC is 0.3% and 0.58%, respectively (Table 3). Something similar happens with the area required, which we estimate to the about 4% for the V-Way cache and more than 12% for Scavenger, while it is below 1% for the SBC (Table 4).

We compare here the performance of the SBC with that of the V-Way cache and the Dynamic Insertion Policy (DIP) [13] because their cost also scales well with the cache size. DIP is a proposal to adapt dynamically the policy of insertion of new lines in sets, alternating between marking the most recently inserted lines in a set as most recently used lines (the traditional policy) or the least recently used ones, the replacement policy being LRU. Notice that if the latter case, only if the block is accessed again in the cache will it become the MRU in its set. Otherwise the next miss will trigger its eviction. This system helps keep the most important part of the working set in the cache when the size of this set is much larger than the cache.

Figure 12 compares the miss rates among SSBC, DSBC, V-Way cache and DIP in the L2 cache for the two-level configuration used previously. Data shown are relative to miss rate of the baseline configuration. DIP has been simulated with 32 dedicated sets and $\epsilon = 1/32$ (see [13]). The last group of bars correspond to the geometric mean of the ratios of reduction of the miss rate for the four policies. The results vary between the 19% reduction for the dynamic SBC and the 12% reduction for DIP, which is the simplest and cheapest alternative. The V-way cache achieves a 15% reduction, slightly better than the 14% one of the static SBC. Benchmark by benchmark, the V-way cache is the best one in three of them, DIP in one, and the dynamic SBC in the other six ones. We must take into account that DIP and the V-Way cache turn misses into hits, while the SBC turns them into secondary hits, which suffer the delay of a second access to the tag array. On the other hand, the duplication of tag-store entries, the addition of one pointer to each entry and a mux to choose the correct pointer increases the V-Way tag access time around 39%, while the SBC has very light structures (up to 17 bits per set plus one bit per tag-store entry), thus having a negligible impact on access time.

9. CONCLUSIONS

We have presented the Set Balancing Cache (SBC), a new design aimed at non-first level caches with a good cost-benefit relation. This cache associates sets with a high demand with sets that have underutilized lines in order to balance the load among both kinds of sets and thus reduce the miss rate. The identification of the degree of pressure on a set, which we call level of saturation, is performed by a counter per set called saturation counter. The balance is materialized in the displacement of lines from cache sets with a high level of saturation to sets that seem to be underutilized, the displaced lines being found in the cache in subsequent searches. Two designs have been presented: a static one, which only allows displacements between preestablished pairs of sets, and a dynamic one that tries to associate each highly saturated set with the less saturated cache set available. The selection of this less saturated set is made by a very cheap hardware structure we call Destination Set Selector (DSS), which yields near-optimal selections.

Experiments using 10 representative benchmarks of the SPEC CPU2006 suite achieved an average reduction of 9.2% and 12.8% of the miss rate for the static and the dynamic SBC, respectively, or 14% and 19% computed as the geometric mean.

This led to average IPC improvements between 2.7% and 5.25% depending on the type of SBC and the memory hierarchy tested. Furthermore, the SBC designs proved consistently to be better than increasing the associativity, both in term of area and performance.

In this paper we have explored the feasibility of using information at the set level to adopt decisions on cache management. Future directions for research include complementing it with information at the line level beyond the recency of use provided by the set local replacement policy and analyzing improvements to the SBC strategy for multicore shared caches.

10. ACKNOWLEDGMENTS

This work was supported by the Xunta de Galicia under projects INCITE08PXIB105161P and *"Consolidación e Estructuración de Unidades de Investigación Competitivas"* 3/2006 and the MICINN, cofunded by the Fondo Social Europeo, under the grant with reference TIN2007-67536-C03-02. The authors are also members of the HiPEAC network.

11. REFERENCES

[1] A. Agarwal and S. D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. *ISCA*, pages 179–190, May 1993.

[2] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martínez. Scavenger: A new last level cache architecture with global block priority. *MICRO*, pages 421–432, December 2007.

[3] B. Calder, D. Grunwald, and J. S. Emer. Predictive sequential associative cache. *HPCA*, pages 244–253, February 1996.

[4] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. *MICRO*, pages 55–66, December 2003.

[5] Digital Equipment Corporation. Digital semiconductor 21164 alpha microprocessor product brief, March 1997.

[6] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. *ISCA*, pages 107–116, June 2000.

[7] HP Labs. CACTI 5.3. cacti.5.3.rev.174.tar.gz. Retrieved in November, 2008, from `http://www.hpl.hp.com/research/cacti/`.

[8] Intel Corporation. Intel core i7 processor extreme edition and intel core i7 processor datasheet, 2008.

[9] A. Jaleel. Memory characterization of workloads using instrumentation-driven simulation. Retrieved on December 18, 2008, from `http://www.glue.umd.edu/~ajaleel/workload/`.

[10] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers. *ISCA*, pages 364–373, June 1990.

[11] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. *HPCA*, pages 288–299, February 2004.

[12] J. Peir, Y. Lee, and W. W. Hsu. Capturing dynamic memory reference behavior with adaptative cache topology. *ASPLOS*, pages 240–250, October 1998.

[13] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., and J. S. Emer. Adaptive insertion policies for high performance caching. *ISCA*, pages 381–391, June 2007.

[14] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way Cache: Demand-Based Associativity via Global Replacement. *ISCA*, pages 544–555, June 2005.

[15] J. Renau et al. SESC simulator. sesc_20071026.tar. Retrieved on May 18, 2008, from `http://sesc.sourceforge.net`.

[16] A. Seznec. A case for two-way skewed-associative caches. *ISCA*, pages 169–178, May 1993.

[17] D. Weiss, J. Wuu, and V. Chin. The on-chip 3-mb subarray-based third-level cache on an itanium microprocessor. *IEEE Journal of Solid State Circuits*, 37(11):1523–1529, November 2002.

[18] C. Zhang. Balanced cache: Reducing conflict misses of direct-mapped caches. *ISCA*, pages 155–166, June 2006.

[19] C. Zhang, X. Zhang, and Y. Yan. Two fast and high-associativity cache schemes. *IEEE MICRO*, 17:40–49, 1997.