Adaptive Set-Granular Cooperative Caching

Dyer Rolán Basilio B. Fraguela Ramón Doallo

Computer Architecture Group
Electronic and Systems Department
Universidade da Coruña, Spain
{drolan, basilio.fraguela, doallo}@udc.es

Abstract

Current Chip Multiprocessors (CMPs) consist of several cores, cache memories and interconnection networks in the same chip. Private last level cache (LLC) configurations assign a static portion of the LLC to each core. This provides lower latency and isolation, at the cost of depriving the system of the possibility of reassigning underutilized resources. A way of taking advantage of underutilized resources in other private LLCs in the same chip is to use the coherence mechanism to determine the state of those caches and spill lines to them. Also, it is well known that memory references are not uniformly distributed across the sets of a set-associative cache. Therefore, applying a uniform spilling policy to all the sets in a cache may not be the best option. This paper proposes Adaptive Set-Granular Cooperative Caching (ASCC), which measures the degree of stress of each set and performs spills between spiller and potential receiver sets, while it tackles capacity problems. Also, it adds a neutral state to prevent sets from being either spillers or receivers when it could be harmful. Furthermore, we propose Adaptive Variable-Granularity Cooperative Caching (AVGCC), which dynamically adjusts the granularity for applying these policies. Both techniques have a negligible storage overhead and can adapt to many core environments using scalable structures. AVGCC improved average performance by 7.8% and reduced average memory latency by 27% related to a traditional private LLC configuration in a 4-core CMP. Finally, we propose an extension of AVGCC to provide Quality of Service that increases the average performance gain to 8.1%.

1 Introduction

Choosing either a private or shared configuration for the last level cache (LLC) is one of the key points of the design in CMPs. When the LLC is shared among all the cores, it requires a high bandwidth because every single request by any upper cache needs to access the interconnection network. Shared LLCs are usually distributed in tiles owned by different cores and, thus, needing different latencies depending on where the requested line is found. As the number of cores and cache banks increases, it becomes more difficult to hide wire delays. Even worse, harmful applications can hurt the performance of other concurrently executing applications. On the other hand, in private configurations, each core is assigned a static portion of the LLC, which provides lower latency, better scalability, isolation and makes the optimization of particular parameters, like power consumption, easier, at the cost of depriving the system of the ability of sharing underutilized resources. CMPs provide room for improving performance since multiple different applications may be executed concurrently, ones being short of cache resources while others can offer underutilized space. Therefore, it is interesting to track the global availability of resources and select the best policies to allocate them appropriately.

Several proposals have been presented in order to share resources in private configurations by displacing or spilling lines from one cache to another [1] [2]. A second way of addressing the impossibility of sharing resources in private caches as well as limiting the amount of space devoted to each application is to set partitions to both private and shared data. Many approaches which propose partitioning have appeared in the last years [3] [4], even existing mixed approaches like [5]. All these techniques uniformly apply the same policy to all sets in each private cache and they do not provide alternatives when the sharing of resources fails. Also, they classify caches in two groups, spillers or receivers, when sometimes it may be better to be neither spiller nor receiver.

In this paper we propose an approach to share resources between caches in a per-set level basis called Adaptive Set-

Granular Cooperative Caching (ASCC). This design measures the degree of stress of each set and spills lines from those sets with a high number of misses related to hits, which are thus unable to hold their working set, to sets with underutilized lines in another cache, where they can be found later using the coherence mechanism. Also, ASCC adds a third neutral state where the set is disabled for both spilling or receiving lines. This way each set applies the policy which best suits it.

Another novel idea explored by ASCC is that the metrics used to manage the sharing of resources can also drive changes in the local management of a set. The fact that the same metric drives both kinds of optimization eases their desirable coordination. ASCC first resorts to spills to alleviate high miss rates. If this does not suffice, the cache may be dealing with capacity problems. ASCC adopts the innovative feature of changing the insertion policy to one specially designed to deal with capacity problems when spilling is not enough to hold the working set in the CMP.

Furthermore, previous designs apply their policies using a static granularity to track the cache behavior. Experiments using different granularities for the ASCC indicate that granularity is an important point of the design. We propose an extension that dynamically adjusts the granularity of ASCC by virtually grouping adjacent sets for the sake of the tracking of their state and the application of the ASCC policies. We have called it Adaptive Variable-Granularity Cooperative Caching. Finally, we have improved this design adding support for Quality of Service.

The rest of this paper is organized as follows. The next section describes the motivation and background of our approach. Sections 3 and 4 explain Adaptive Set-Granular Cooperative Caching and Adaptive Variable-Granularity Cooperative Caching, respectively. The environment used in our experiments is described in Section 5, the results and analysis of our proposals being discussed in Section 6. The hardware cost is examined in Section 7. Section 8 introduces an extended design to reach Quality of Service afterwards. The last section is devoted to the conclusions and future work.

2 Motivation and Background

Novel approaches, many of which are not suitable for the traditional multiprocessors, have been proposed to exploit the new capabilities of CMPs. A great number of them has focused on the memory hierarchy due to its large impact in the overall performance. By using private LLCs, a system is able to provide the applications with isolation, low latency and minimum bandwidth. Also, it allows easier design extensions, as the tags are associated to each cache. Several approaches have appeared in the past years in order to provide private levels with shared capacity. Some techniques use partitioning in order to limit the amount of space for private and shared data while others try to make a better use of resources by spilling lines between caches, even existing mixed approaches like the Elastic Cooperative Caching (ECC) [5]. ECC splits sets in two different regions, a private one, to allocate lines evicted from the upper level, and a shared region, to hold lines spilled by neighbor caches.

Regarding the partitioning-oriented approaches, Adaptive Selective Replication [3] dynamically analyzes the workload behavior and adapts the degree of replication on a per block basis to match the application requirements. In Cooperative Cache Partitioning [4] resources are partitioned both in terms of time, giving different priorities of execution to different partitions, and space, setting the size of the different partitions depending on the application requirements. The main problem with the approaches that rely on partitioning is that they can waste space if the allocated ways in the shared or in the private region are not useful. These approaches often waste space by forcing to allocate at least one way for each type of data, even if it is not profitable, and they need large structures to operate. Additionally, the uneven demands experienced by different sets [6] [7] [8] [9] may result in some ways being wasted in the regions of some sets, while the same regions could benefit from more ways in other sets. As for the designs that rely on spills, Cooperative Caching (CC) [1] spills lines to other caches instead of directly evicting them to main memory if they are the only copy in the chip. CC disregards whether the spilling is going to benefit the cache. In a similar way, any cache can play the role of receiver even if it has no free space to share and the final candidate is chosen randomly. Dynamic Spill-Receive [2] (DSR) labels each cache as either spiller or receiver depending on a global counter per cache used by its set dueling mechanism. This global counter is updated by all the caches in order to determine whether the spillings are going to hurt receiver caches or not. A common limitation of CC and DSR is that they apply uniformly the same policy to all the cache sets, as they cannot detect whether a given set is going to perform better with more ways or applying a different policy. Even worse, all sets have to work always as spillers or receivers when sometimes it may be better to neither spill nor receive spilled lines. Furthermore, DSR restricts the number of spiller or receiver sets because they could be members of a Set Dueling Monitor (SDM) which uses a fixed policy even when a different one is performing better in the cache.

In summary, there are no approaches that use a fine-grain metric to profile the state of the caches and apply different policies to different portions of the cache depending on their status. Also, no approach allows a given set to be in a neutral state, not operating as spiller or receiver. Moreover, these approaches are not designed to deal with capacity problems. Our approach, called Adaptive Set-Granular Co-

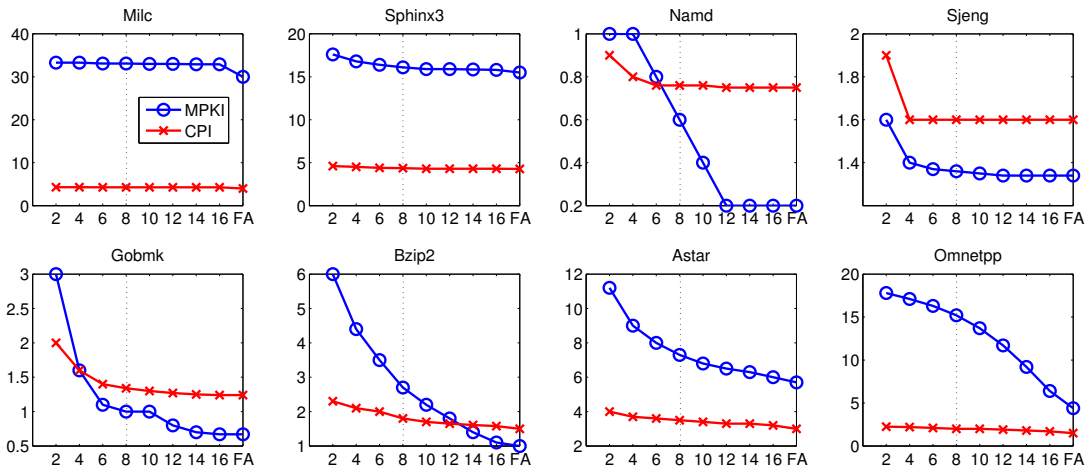

Figure 1. MPKI and CPI for SPEC CPU2006 benchmarks as the number of allocated ways varies. The X axis shows the number of ways allocated from a 16-way 2MB cache (the remaining ways are disabled). Benchmarks in the upper row can provide cache capacity and benchmarks in the lower row can benefit from allocating more ways gradually.

operative Caching (ASCC), is able to determine in a per-set basis whether the set should be a spiller, a receiver or neither of them while it tackles capacity misses at the same time. Furthermore, we have designed an extension called Adaptive Variable-Granularity Cooperative Caching (AVGCC), which dynamically changes the granularity of the ASCC policies.

Figure 1 shows the MPKI and CPI obtained in a 2MB 16 ways L2 cache enabling from 2 to 16 ways for 8 benchmarks from the SPEC CPU2006 suite. In our experiments in Section 6 we have used a 1MB 8 ways L2 cache as the baseline configuration, which is represented in the graphs with a dotted line. Statistics are gathered for the first 10 billion instructions executed after the initialization. We can see how benchmarks in the lower row significantly benefit from allocating more ways, while the ones in the upper row do not. In the first row we can deduce that *milc* and *sphinx3* are streaming applications because they have a high MPKI and they are barely affected by the increase in the number of allocated ways, *namd* has a small working set and *sjeng* is sensitive to cache capacity only up to $1/4th$ MB. All of them can offer cache capacity by increasing the number of allocated ways for the hungry applications. On the other hand, benchmarks in the lower row significantly benefit from allocating more ways. These benchmarks that are sensitive to cache capacity may take advantage of the simultaneous execution with applications that do not benefit from receiving extra space to hold their working sets if we provide a mechanism to reassign underutilized resources between applications. The last column in each graph shows the MPKI and CPI using full associativity in the cache. We can see how in many benchmarks there is still room for improving performance by reducing capacity misses.

Memory references are known to be non uniformly distributed across the sets of a set associative cache [6] [7] [8] [9]. Thus, there may be sets which might benefit from allocating a greater number of ways and sets that are able to hold their working set with their current assigned ways. This fact is illustrated by Figure 2, which shows the percentage of sets which benefit from enabling more ways and the percentage of sets which remain unaffected by this increase during the execution of the applications *astar* and *milc* of the SPEC CPU 2006 suite in subfigures (a) and (b), respectively. The simulation environment is the same as in the previous study. The classification is based on the MPKI of each set. If the MPKI does not decrease when the number of allocated ways increases, or if it decreases less than 1% related to the previous MPKI, calculated using 2 fewer ways, we mark this set as a constant set. Otherwise, it is a favored set. We can see how the percentage of sets, either favored or constant, changes considerably from 10 enabled ways on in Figure 2 (a) and from 6 to 12 in Figure 2 (b).

These results indicate that, depending on the application and cache associativity, a different number of sets may benefit from getting extra ways, while others do not. Our approach tries to detect these different behaviors in the sets to perform cache-to-cache transfers, from a cache set, which is not currently able to hold its working set and would benefit from more space, to another cache where the set is underutilized. Relatedly, we can see how having a neutral state can be beneficial, that is, it may be the case that the best for

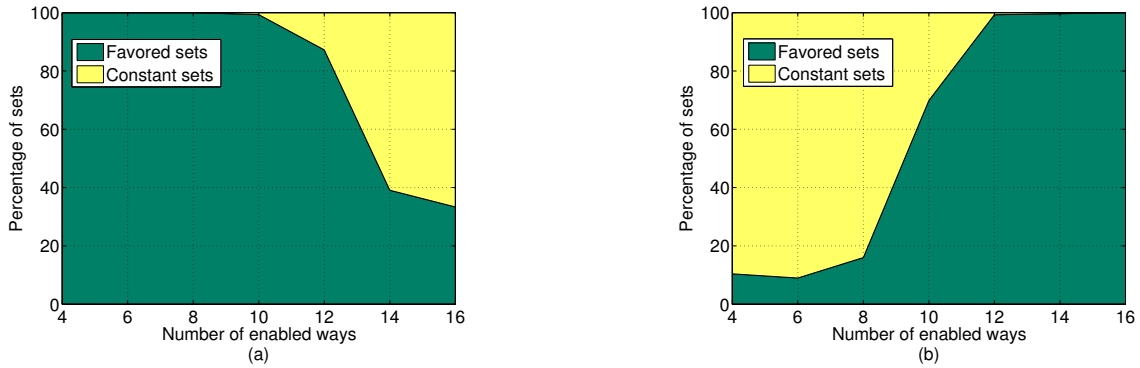

Figure 2. Percentage of sets that benefit from allocating more ways (favored) and percentage of sets which remain unchanged (constant) in the execution of the *Astar* (a) and *Milc* (b) benchmarks.

a set is to neither spill to nor receive lines from another set. For instance, increasing the number of ways in Figure 2 (a) from 10 to 12 leaves around 90% of favored sets, while 10% remain unaffected. When we increase the number of allocated ways up to 14, only 36% of the sets whose behavior improved when going from 10 to 12 lines keep taking advantage of having more ways. The other 64% has reached its optimum behavior using 12 ways and they do not benefit from getting more ways, while allocating fewer than 12 could be harmful. As a result, a neutral state, where the set is neither spiller nor receiver, is beneficial for this latter group of sets.

Furthermore, as we could see in Figure 1, a noticeable percentage of the miss rate is due to capacity misses. Since spills may not be enough to alleviate capacity problems, our approach tackles these problems changing the insertion policy of the sets as well.

3 Adaptive Set-Granular Cooperative Caching

We propose the Adaptive Set-Granular Cooperative Caching to promote a better distribution of cache resources in CMP platforms, which use private last levels for the cache memory hierarchy, by spilling lines from those caches which are short of space to other ones with underutilized resources and by adapting the insertion policy of cache sets to their demand. Our approach tries to balance the storage of working sets of each core between caches and to reduce capacity misses by displacing lines and applying a new insertion policy, respectively. In order to track the status of each set, our proposal uses, initially, one saturation counter per set [7]. This is a counter with saturating arithmetic that is increased when a miss occurs in the set, and decreased when a hit takes place. The value of this counter is called Set Saturation Level (SSL). We use the same design as in [7], where the saturation counters work in the range 0 to $2 * K - 1$, K being the associativity of the cache henceforth.

3.1 Balance of working sets

Our design classifies a set in one of three groups depending on its SSL. We have used thresholds for the different values of SSLs regarding the previous results in [7]. Thus, when the SSL is below K, the high recent proportion of hits indicates that the set can hold quite successfully its working set. In this situation it is likely there are underutilized lines in the set that could be used to store part of the working set of the set with the same index in other private caches. Therefore the set is classified as a receiver set. When $K \leq SSL < 2 * K - 1$, the set has some recent hits but given the degree of pressure on it, it might be unwise to devote lines of it to store lines of the working set of other sets. This way, the set is in a neutral state where it is neither a sender nor a receiver. Finally, when the saturation counter of a set reaches its maximum value, $2 * K - 1$, the high proportion of misses related to hits indicates that the set is not able to hold its working set and is thus classified as a sender. If the line to be evicted during a replacement operation in a sender set is the last copy in the chip, our proposal tries to optimize the usage of the caching resources by spilling it to a receiver set (with the same index) in another private cache in the same level instead of evicting it to a lower level. If there are several potential receiver sets, the one with the lowest value will be selected. If a tie occurs among several caches, the destination cache is selected randomly among the ones with the lowest value. This is the only point of the design which requires further access to the interconnection network. In order to scale the design, an intermediate structure per cache similar to the Spill Allocator proposed in [5] can be easily adapted. It would only require one entry per set and it would store the saturation counter value, which must be lower than K or K when there is no valid candidate, and the index of the current candidate

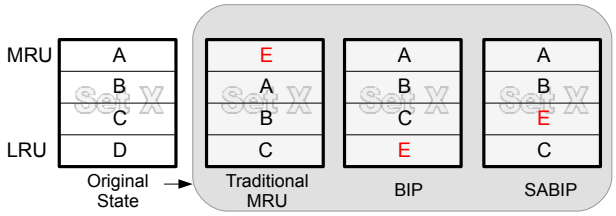

Figure 3. Different insertion policies for the new line E in set X in a 4-way cache.

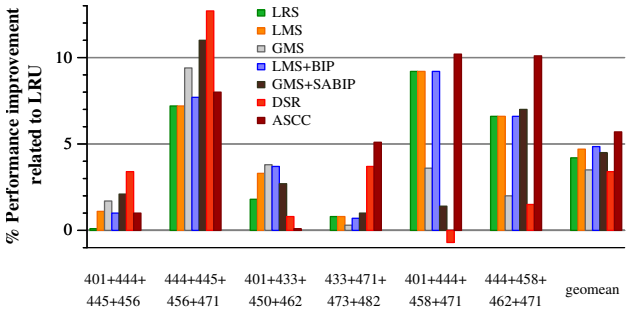

Figure 4. Performance improvement over the baseline for several intermediate designs of the ASCC using local (L) or global (G) policies to make decisions on spillings, for DSR and for ASCC itself running four applications.

cache. It should be updated with every miss in the other caches. Note that the low SSL in the receiver set favors that a previously spilled line is not likely to be spilled again in the near future, preventing inactive lines from being spilled repeatedly. As the spilling of lines is performed after a miss, the search of a candidate cache can be done simultaneously with the line search operation provided by the coherence mechanism, just as in [2].

3.2 Spilling-Aware BIP: A new insertion policy

Another novelty of our approach is that it changes a basic policy of the private caches in CMPs in response to the feedback of the cooperation mechanism. Namely, when a spiller set is not able to find a candidate receiver set, this indicates that spilling is not possible because the set has a high SSL in all caches, which indicates a global problem of capacity. Thus, in this situation our design changes the insertion policy of the spiller set in order to avoid capacity misses. The insertion policy reverts to the traditional MRU (Most Recently Used) one when the value of the saturation counter falls below K, indicating that the capacity problem has disappeared. The Bimodal Insertion Policy or BIP [10], which inserts new lines in the MRU position of the recency stack with a low probability, ε, while it inserts most lines in the LRU position, proved to be very effective to provide thrashing protection and thus reduce capacity misses. Our design uses a variation of BIP which inserts most lines not in the LRU position, but in the previous one in the recency stack, LRU-1, in order to discard temporary data. We have called this insertion policy Spilling-Aware BIP or SABIP.

Note that using the original BIP two harmful behaviors could happen. Firstly, destination sets working under BIP could evict just inserted lines, depriving them of a chance to be reused and consequently promoted to the MRU position, not only due to local misses, but also to make room for a spilled line. Our proposal avoids that behavior by applying the restriction on the SSL value of the destination sets so that destination sets always apply MRU insertion. But this situation could also happen in sets which were previously in BIP mode and which, due to a recent good behavior, become destination sets. Here only SABIP protects the most recently inserted line, which has probably good locality given the change of behavior of the set, from being evicted due to a spill from another set. Also, as SABIP gives more chances than BIP for lines to be reused before their eviction, it generates fewer spillings of lines with some locality, as they are retained more effectively in the set.

Our approach uses an *insertion policy bit* per set to determine the current insertion policy of the set. Figure 3 explains the behavior of different insertion policies. Also, our approach adds swapping of lines between caches when both the requested line found in another cache and the victim line selected by the replacement policy in the cache that performed the request, are the last copy on chip. This is done in order to keep these lines longer in the CMP.

3.3 Design breakdown

Figure 4 reasserts our design decisions and measures the contribution to performance of each one of them by showing some intermediate points of the design. The experiments consider 4 cores and multiprogrammed workloads using benchmarks from the SPEC CPU2006 suite. The characterization of these benchmarks and the simulation environment will be described in Section 5. In this figure, LRS or Local Random Spilling is ASCC without insertion policy modifications to tackle capacity misses and choosing randomly any cache with a value in the saturation counter of the current set lower than K as a candidate to receive a spilled line. LMS or Local Minimum Spilling selects the cache with the lowest value instead. GMS or Global Minimum Spilling uses only one counter to globally manage each cache (4 bits to represent the only saturation counter per cache with an associativity of 8), so that all the cache sets have the same behavior. LMS+BIP adds BIP to LMS

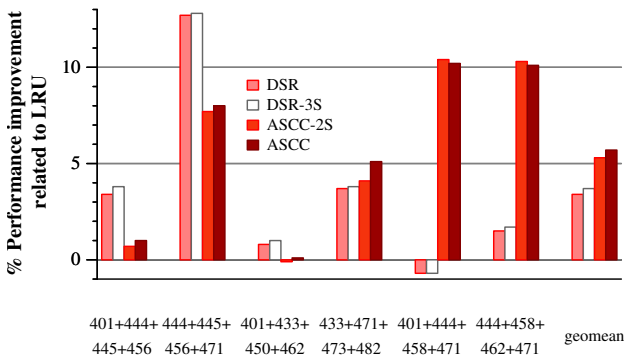

Figure 5. Performance improvement over the baseline for ASCC, a variation that uses only 2 states to classify the role of the sets, DSR and a variation that uses 3 states to classify the role of the cache, running four applications.

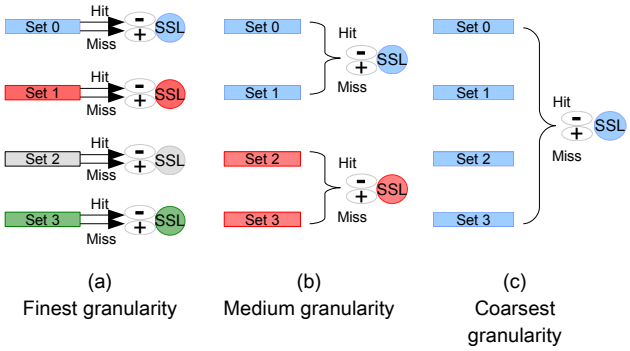

Figure 6. Different levels of granularity.

and GMS+SABIP (with an extra bit to determine the current insertion policy in the cache) adds SABIP to GMS. Also, we show the Dynamic Spill-Receive (DSR) [2] performance. Note that DSR is similar to GMS but using the set dueling mechanism instead of the SSL one, and ASCC is identical to LMS+BIP but using SABIP. We can see that LMS outperforms LRS thanks to the selection of the receiver cache with the lowest value for the saturation counter of the current set. In a similar way, LMS outperforms GMS thanks to its ability to handle each cache set separately. The benefits of SABIP, with respect to BIP in this environment, can be deduced comparing ASCC with LMS+BIP. It is worthy to point out the improvement of GMS+SABIP over DSR. This design, which uses half the negligible storage overhead of DSR, provides 30% more speedup over the baseline thanks to its management of the insertion policy. Furthermore, the comparison of GMS+SABIP with ASCC gives an idea of the value of a fined grain granularity in the cache management.

Figure 5 evaluates the usefulness of the neutral state of a set, in which it is neither a spiller nor a receiver. DSR with three states (DSR-3S) is a variation of DSR which uses the 2 most significant bits of its selector counter to decide if the cache is a spiller, when the 2 MSBs are 11, receiver, 00, or neutral, 01 or 10. This design achieves 9% more performance improvement over the baseline cache than the original DSR. Also, we have tried an ASCC design which uses only 2 states, ASCC-2S, in which a set is a spiller if its SSL is $>= K$ and a receiver otherwise, to see the effect of the neutral state in ASCC. We can see in the last column, which represents the geometric mean, that its performance improvement over the baseline is 10% smaller than that of ASCC.

4 Adaptive Variable-Granularity Cooperative Caching

In Figure 4 we could see how the designs that apply a global metric to trigger the spilling of lines perform better in the first two of the six multiprogrammed workloads. We have performed experiments applying different granularities in ASCC to study their influence on the performance. Applying a granularity for n consecutive sets using saturation counters simply involves updating a single counter and using its value to take the decisions for the whole group. Table 1 shows the percentage of performance improvement over the baseline when applying ASCC grouping sets 1 by 1, 4 by 4 and so on, up to 4096, i.e, from using 4096, 1024 counters and so on, to using only one counter for the whole cache. Note that the insertion policy relies on the saturation counter, so all the sets associated to one counter apply the same insertion policy. This is sensible, as if the number of counters is small, it means we are using a global metric and the insertion policy should be global as well. Figure 6 shows an example for a 4-set cache applying different levels of granularity using SSLs.

From these results we can infer that some workloads, or individual benchmarks within a workload, are better managed using a global metric, while others work better with a fine granularity of management. Increasing the granularity can help to correct ASCC overreactions to temporary outstanding behaviors of particular sets, while decreasing it allows to track the state of the cache sets and detect capacity problems in a more accurate way. Therefore, we have two options. The first one is to fix statically the granularity to some value that seems to work well overall. The drawbacks of this option are that it is not optimal, it does not adapt to the nature and the changes of behavior of the workloads, and there is no guarantee the granularity selected will be far from good for some workloads. The second option is to design a dynamic mechanism to adapt the algorithm granularity to the needs of the applications during the execution.

Table 1. Percentage of performance improvement of the ASCC from using 4096 counters (the original ASCC design, using 1 counter per set) to using only 1 counter (grouping all 4096 sets in the cache), related to the baseline configuration.

MW	ASCC	ASCC1024	ASCC256	ASCC64	ASCC16	ASCC4	ASCC1
445+401+444+456	1%	1%	1%	1%	1%	1.2%	2.1%
445+444+456+471	8%	13.6%	15.6%	17.9%	17.9%	16.4%	11%
433+462+450+401	0.1%	0.96%	1.4%	1.4%	1.4%	1.5%	2.7%
433+471+473+482	5.1%	6.17%	5.6%	6.4%	6.2%	6.1%	1%
458+444+401+471	10.2%	5.13%	6.6%	9%	9.1%	7.1%	1.4%
458+444+471+462	10.1%	5.45%	7.9%	9.2%	9.1%	7.6%	7%
geomean	**+5.7%**	**+5.2%**	**+6.2%**	**+6.9%**	**+6.8%**	**+6.5%**	**+4.5%**

Our proposal entails starting with one counter for the whole cache. The number of counters used is increased, duplicating the current number, if more than half the saturation counters in use have a value lower than K. When this happens, it means that most sets in the cache can provide sets in other caches with space, so we use a finer granularity to allow the sets to exchange lines in a more accurate way. Also, the current number of counters in use is decreased, halving it, when every pair of neighbor counters has a similar value, specifically when there is an absolute difference between their values of two at the most, and their embraced sets are working under the same insertion policy. This is sensible, since we can save storage space using one counter to track their SSL simultaneously, as they are similar. Note that different caches in the same CMP can be applying different granularities. We have called this technique Adaptive Variable-Granularity Cooperative Caching or AVGCC.

4.1 Hardware description

In order to implement the described mechanism, AVGCC needs three counters per cache. The first one, D henceforth, stores the logarithm of the current number of sets per saturation counter to base 2, that is, D is the logarithm to base 2 of the granularity applied by AVGCC. For instance, if 4 sets map to a single saturation counter, D will store the value 2. The saturation counters are then accessed adding a shifter controlled by D. This way, given a set index I, the associated counter would be $I >> D$.

Secondly, AVGCC uses another counter, A, in order to track how many pairs of SSLs fulfill the conditions required when checking whether the number of counters in use must be halved. The condition is evaluated twice, before updating the accessed SSL and after doing it. A flip-flop is needed to check whether the evaluation of the condition between the given SSL and its adjacent one has changed after updating the former. Counter A is decreased if the evaluation of the condition turns from being fulfilled to not being, increased in the opposite case, and it remains unchanged otherwise. Finally, the number of counters in use must be duplicated if more than half the saturation counters have a low value (below K). A counter, B, is needed for this purpose. B is increased when the value of a saturation counter goes from K to K-1 and decreased when it changes from K-1 to K. Furthermore, as it was previously mentioned, the D counter is increased when $A = (S >> D)/2$, where S is the number of sets, as every pair of the $(S >> D)$ saturation counters in use fulfills the condition for it. Counter D is decreased when $B > (S >> D)/2$. Also, after updating the current number of counters, the new ones are initialized to $K - 1$ and the associated insertion policies are reset to the traditional MRU one. This process is performed periodically. We will see in Section 6 that this design provided an average improvement of 7.8% over the baseline for the same set of workloads and configuration used in Table 1, in comparison with the 6.9% of the best static approach.

5 Simulation environment

To evaluate our approach we have used the SESC simulator [11] with a baseline configuration based on four-issue out-of-order cores with two cache levels, both of which are private to each core. This configuration is detailed in Table 2. The ratio of LLC space per core is similar to that used in the related bibliography and modern processors [12] [13].

We have used 13 SPEC CPU2006 benchmarks with an MPKI of at least 1, as shown in Table 3, to make 14 multiprogrammed workloads of two applications and 6 of four.

Table 2. Architecture.

Processor	
Frequency	4GHz
Fetch/Issue	4/4
ROB entries	176
Integer/FP registers	96/80
Memory subsystem	
L1 i-cache & d-cache	32kB/2-4-ways/32B/LRU/WT
L2 (unified, inclusive) cache	1MB/8-way/32B/LRU/WB
L2 Cache latency (cycles)	9 **local** hits, 25 **remote** hits
Main memory latency	115ns
Coherence protocol	MESI-based broadcasting

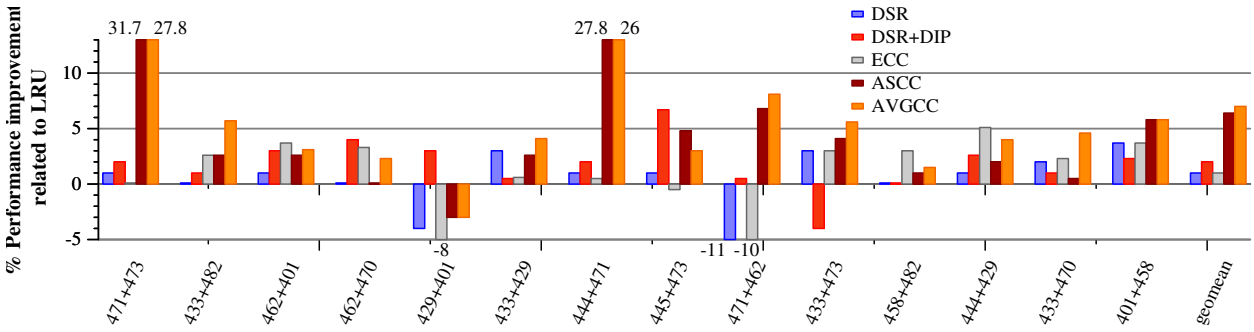

Figure 7. Performance improvement for DSR, DSR+DIP, ECC, ASCC and AVGCC running two applications over the baseline.

These workloads cover combinations between applications that benefit from allocating more ways and other ones that do not, workloads where no benchmark benefits from getting extra space and workloads where all the benchmarks would benefit from getting it. They have been executed using the reference input set (*ref*), during 10 billion instructions after the initialization. When each core commits this number of instructions it continues its execution until the last core finishes, in order to keep competing for the cache resources.

6 Performance evaluation

Our evaluation relies on the weighted speedup [14], which indicates execution time reductions, and the harmonic mean of CPIs [15], which balances fairness and performance. ASCC, as well as the other approaches, has been applied in the last level of the cache memory hierarchy. In our experiments we have also tested DSR with 32 sets per Set Dueling Monitor and 1 SDM per policy, a combination of DSR and DIP [10], where DIP decides the insertion policy for the global cache (either BIP or the traditional LRU one) depending on which policy is working better using also set dueling and, finally, the ECC approach described in Section 2. Our designs, as well as the DIP used in the combination with DSR, use a probability $\varepsilon = 1/32$ of inserting the new line in the MRU position using BIP. ECC uses the values proposed in [5] for the thresholds and we have implemented it without the distributed structures they propose, tracking the shared state of the lines with an additional bit per block. Note that this implementation provides the ECC design with more accuracy than the original design, which cannot track the information of all lines in the cache, specially if the degree of replication is low. Finally, AVGCC checks whether the number of counters must be updated every 100000 accesses to the cache.

Table 3. Benchmarks characterization.

Benchmark	L2 MPKI	CPI	Benchmark	L2 MPKI	CPI
401.bzip2	2.7	1.8	458.sjeng	1.36	1.6
429.mcf	40.1	10.4	462.libquantum	22.4	4.3
433.milc	33.1	4.28	470.lbm	29	2
444.namd	1	0.76	471.omnetpp	15.2	2
445.gobmk	1.1	1.34	473.astar	7.3	3.5
450.soplex	3.6	1	482.sphinx3	16.1	4.37
456.hmmer	3.4	1.3			

6.1 Speedup and fairness analysis

Figures 7 and 8 show the performance improvement over the baseline for the different approaches, measured as the weighted speedup of CPIs, using 2 and 4 cores, respectively. Numbers above or beneath the bars provide the percentages that are outside the scale range. The last column shows the geometric mean for each design. ASCC, which gets 6.4% and 5.7% of performance improvement over the baseline when executing 2 and 4 applications, respectively, and AVGCC, which achieves 7% and 7.8%, respectively, clearly outperform the other approaches. DSR adapts to the requirements of the applications thanks to the set dueling mechanism, but it is not able to take advantage of the state of each set as ASCC and AVGCC do. Also, it forces sets to be either spillers or receivers and lacks of a policy oriented to capacity problems. DSR+DIP outperforms DSR executing 2 applications because it tackles capacity misses as well. The problem with DSR+DIP is that its BIP insertion policy is not aware of the spillings, as we explained in Section 3.2. Thus, one just inserted line, which is not likely to be reused in the near future if the set is applying BIP, can be spilled to another cache and, as a result, a line with more locality could be evicted there. Even worse, a spilled line can evict a just inserted line in a set applying BIP depriving it of a chance to be reused and consequently promoted to the top of the recency stack. Let us recall that this particular

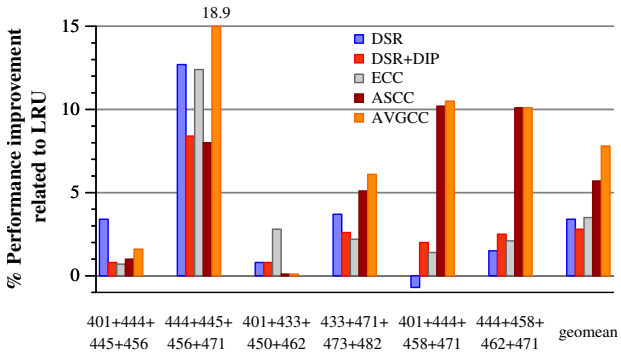

Figure 8. Performance improvement over the baseline for DSR, DSR+DIP, ECC, ASCC and AVGCC running four applications.

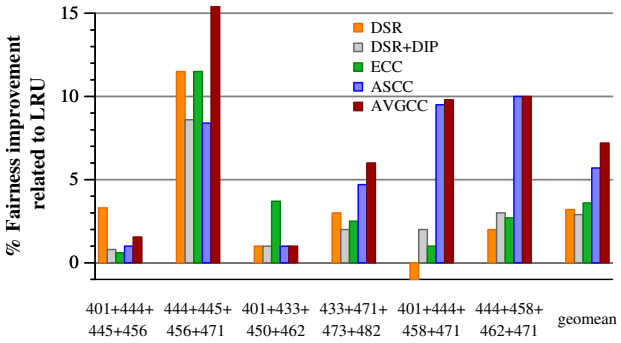

Figure 9. Fairness improvement over the baseline for DSR, DSR+DIP, ECC, ASCC and AVGCC running four applications.

behavior cannot happen in our designs. As the number of executing applications increases, the number of candidate caches to receive spilled lines increases as well, so the negative effects of BIP are likely to happen more frequently. That is why DSR+DIP degrades the performance of DSR executing 4 applications. The discrete and even negative behavior of DSR+DIP with respect to DSR emphasizes the need for modifying policies to adapt them to different environments, as we have done with SABIP, and for integrated management designs such as ASCC and AVGCC. The ECC performance improvement is modest compared to those of ASCC and AVGCC because it mainly relies in a high degree of replication, as it uses scalable structures that cannot hold information for all the lines in the cache and has the problems inherent in partitioning described in Section 2. Finally, AVGCC outperforms ASCC by adapting the granularity of its policies to the different requirements of the applications.

Figure 9 shows the percentage of fairness improvement with respect to the baseline system of each one of the considered approaches calculated as the harmonic mean of IPCs, using 4 applications. ECC gets better results than DSR and DSR+DIP because it is able to reduce the execution time for the longest applications. Overall the results and therefore the explanations for them, are similar to the ones obtained in the performance analysis. From these results we can conclude that ASCC and AVGCC do not hurt fairness when speeding up mixed workloads, AVGCC being again the leader thanks to its larger flexibility.

We have also simulated the usage by all the cores of an L2 shared cache of the same aggregated capacity in which addresses are mapped to banks in an interleaved way. This cache has been simulated using an average latency (almost twice the latency of a private L2 in the baseline for the 2-core experiments and almost four times using 4 cores) for the accesses to the different banks, assuming a uniform distribution of the accesses across the banks given by the interleaved mapping. Finally, all caches are write-back in this configuration. For the sake of space and clarity the results for these simulations are not shown in the figures. Globally, the 2MB shared cache outperformed the baseline private configuration in the 2-core experiments by 1.8% and 1.7% in terms of performance and fairness, respectively, laying quite far from the performance of AGCC and AVGCC. Using 4 cores, the 4MB shared cache got a 3% of improvement in both metrics. This means that, in general, and despite the automatic sharing of resources inherent in shared caches, private designs with additional sharing mechanisms can be more effective.

6.2 Average Memory Latency analysis

Figure 10 shows the average memory latency normalized to the baseline configuration (represented by the horizontal dotted line of 100) for all the approaches in the 2-core configuration. Each bar is broken down showing the percentage of accesses to the L2 which result in hits in the local L2, in a remote one or in main memory. The percentage due to the L1 hits is not shown because it is almost the same for all the approaches. The average memory latency has been calculated regarding that each access is sequentially processed, without overlaps between accesses. This bar graph provides some feedback on previous results. For instance, we can infer that ASCC and AVGCC degrade the baseline performance for the workload 429+401 (*mcf+bzip2*) because most local L2 hits in the baseline become remote L2 hits for both approaches. The last column shows the geometric mean. DSR gets a 5% of improvement, DSR+DIP 12%, ECC 1%, ASCC 18% and AVGCC 22% in the 2-core configuration. As for the 4-core configuration, not shown, DSR outperforms the baseline design by 10%, DSR+DIP by 14%, ECC by 11%, ASCC by 21% and finally AVGCC by 27%. This translates in average power consumption reductions in the memory hierarchy for the AVGCC of 25% and 29% in our 2 and 4-core experiments, respectively.

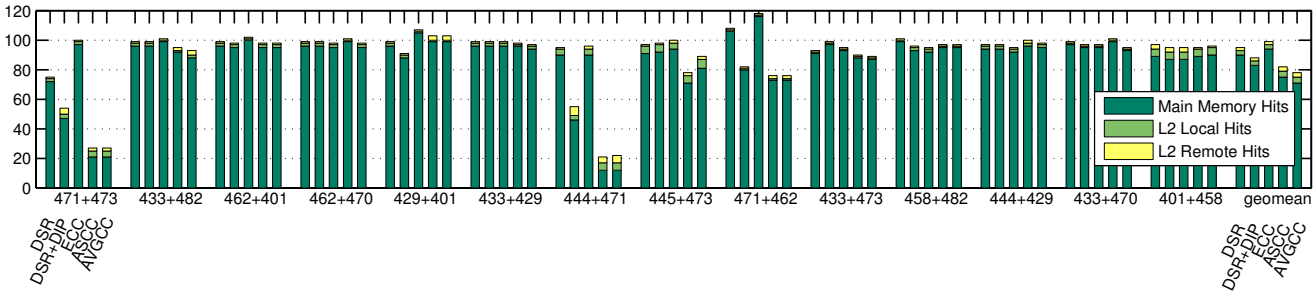

Figure 10. Percentage of improvement in the average memory latency and breakdown of the accesses for DSR, DSR+DIP, ECC, ASCC and AVGCC running two applications over the baseline.

6.3 Sensitivity analysis

We have performed experiments with **multithreaded** applications in order to evaluate our proposals in environments where sets tend to have a more uniform demand in all caches. For these experiments we used benchmarks from the SPLASH2 and PARSEC suites running them during 10 billion instructions (most of them until completion) and using the large input set for PARSEC and the appropriate input set for SPLASH2 using 4 threads. Most of these benchmarks are not hard memory demanding, so the L2 capacity was reduced to 512KB to get meaningful results. ASCC, with an average improvement of 5%, and AVGCC, with 6%, achieved the best results again in terms of reduction in execution time over the baseline. In this environment the spilling of lines can benefit even the receiver caches, which may need the line in the near future, so our policies aim to take advantage of this behavior as well.

Furthermore, we added a 16KB stride **prefetcher** to each LLC in the multiprogrammed experiments obtaining similar results. Namely, ASCC outperformed the baseline by 6% and 5.5% and AVGCC by 6.4% and 7.6% in the 2 and 4 core configurations, respectively. Prefetchers reduce the miss rate in the LLC, but this is done at the expense of consuming more bandwidth. This way, the impact of our policies is slightly reduced by the presence of prefetchers in the 2-core configuration. The reduction is negligible in the 4-core CMP because as the number of cores increases the bandwidth savings provided by ASCC and AVGCC are much more critical, particularly as the prefetchers consume more bandwidth.

Table 4. Cost-benefit analysis of the AVGCC as a function of the cache size.

Cache size	% Average reduction in off-chip accesses (4 / 2 cores)	Storage Overhead
1MB	27% / 14%	0.17%
2MB	12% / 9%	0.17%
4MB	12% / 9%	0.17%

Finally, Table 4 shows the percentage of reduction in the number of off-chip accesses achieved by AVGCC with respect to the baseline, as well as the overhead it involves, for **different cache sizes**. The computation of the overhead of our proposals will be discussed in Section 7. Since the miss rate decreases as the cache size increases, AVGG impact tends to be smaller the larger the cache is.

6.4 AVGCC Behavior

In this section we use measurements on the internal behavior of AVGCC and compare them to those of other techniques in order to better understand how it achieves these results. We have performed experiments on two parameters, the number of spillings and hits per spilled line. In the 2-core experiments AVGCC performed on average 13% fewer spills than the second best approach (DSR+DIP henceforth) and 60% fewer than the worst one (ECC). Also, its ratio of hits per spilling is 28% larger than that of the second best policy. Regarding the 4-core experiments, AVGCC performed 28% fewer spillings than the following best approach, and 70% fewer than the worst case. Also, its ratio of hits per spilling was 36% greater than the one of the DSR+DIP approach.

From these results we infer that AVGCC is not only able to achieve a higher ratio of hits per spilled line, but it also needs fewer spillings to get it. This means that AVGCC performs fewer useless spillings by relying, among other points of the design, on the neutral state of the sets.

7 Cost

In this section we evaluate the cost of AVGCC in terms of storage requirements. AVGCC requires a saturation counter per set to monitor its behavior and one additional bit per set in order to determine the insertion policy. Also, AVGCC needs three additional counters: one to track the current granularity for its policies (the D counter, explained in Section 4), another one to determine if all pairs of neighbor

Table 5. Baseline and AVGCC storage cost in a 1MB/8-way/32B/LRU cache.

	Baseline	AVGCC
Tag-store entry:		
State(MESI+LRU)	5 bits	5 bits
Tag ($42 - \log_2 sets - \log_2 32$)	25 bits	25 bits
Size of tag-store entry	30 bits	30 bits
Data-store entry:		
Set size	32*8*8 bits	32*8*8 bits
Additional structs per set:		
Saturation Counters	-	4 bits
Insertion policy bit	-	1 bit
Total of structs per set	-	5 bits
Halving/Duplicating counters mechanism:		
A,B & D counters	-	12+12+4 bits
Total	-	28 bits
N of tag-store entries	32768	32768
N of data-store entries	32768	32768
N of sets	4096	4096
Size of the tag-store	120kB	120kB
Size of the data-store	1MB	1MB
Size of additional storage	-	2560B+$\sim$4B
Total	**1144kB**	**1146kB** (0.17%)

sets have a similar value (the A counter) and a last counter to check whether more than half the saturation counters in use have a value lower than K (B counter). Based on this, Table 5 calculates the storage required for AVGCC related to an 8-way 1MB baseline cache with lines of 32 bytes, assuming addresses of 42 bits. Note that the ASCC storage cost would be the same as the AVGCC one except for the additional 4 bytes that A, B and D counters mean. We can see that the storage overhead of the AVGCC is very small, and it pays off for the performance benefits and power savings obtained.

Furthermore, as we could see in Section 4, it is not necessary to have one counter per set, i.e., to apply the finest granularity, in order to get the best overall results. Thus, we performed experiments limiting the maximum number of counters, that is, using a coarser granularity than the finest one in the AVGCC, in order to reduce the storage overhead. The speedups achieved using 4 cores go from 6.8% when limiting the number of counters to 128 (which only requires 83B) to 7.1% using 2048 counters at the most (1284B). While these results are lower than the 7.8% AVGCC obtains by having the finest granularity using 4096 counters, they need 97% and 50% less storage, respectively.

8 Quality of Service Aware AVGCC

In some CMPs losing performance may be unacceptable. AVGCC degrades the baseline performance sometimes. In order to solve this problem we propose an extended design to provide Quality of Service. Our design can inhibit AVGCC by stopping spillings and fixing the insertion policy to MRU. We leverage the fact that this inhibition can be done by limiting the increase that an update in the saturation counters means when a cache miss occurs. A harmful operation of AVGCC is detected when its number of misses is greater than in the baseline cache. The number of misses of the baseline cache (*MBC*) is estimated by tracking the misses (*SampledSetMisses*) in those sets (*SampledSets*) working under the MRU traditional insertion policy and which have a value for their saturation counters greater than $K - 1$, as these sets cannot receive lines. Then MBC is estimated as:

$$MBC = CacheSets * (SampledSetMisses/SampledSets) \quad (1)$$

As for the number of misses for AVGCC (*MissesWithAVGCC*), it is simply collected using a counter. Our design calculates a ratio called *QoSRatio* whose purpose is to adjust the saturation value of the sets in order to penalize or reward them depending on their behavior. It is calculated every 100000 cycles simultaneously with the recalculation of the number of counters following the equation:

$$QoSRatio = MBC/max(MBC, MissesWithAVGCC) \quad (2)$$

After the computation all the parameters are initialized and the saturation counters are updated after each miss by adding the *QoSRatio*, while they are decreased in 1 unit after a hit as usual. The QoS-Aware AVGCC requires a per-core storage overhead of 2 bytes for both miss counters (*SampledSetMisses* and *MissesWithAVGCC*), 4 bits to store the *QoSRatio* value (1.3 fixed point format) and 12 bits (assuming 4096 sets in the cache) to count the number of sampled sets. Finally, 3 additional bits are needed per saturation counter (4.3 fixed point format). Altogether, this design means a 0.35% of storage overhead over the baseline considering the finest granularity, that is, having one counter per set. Figure 11 shows the percentage of performance improvement over the baseline system of the Quality of Service Aware AVGCC using two cores. Our Quality of Service approach globally outperforms the original AVGCC, as stated in the last column of the bar graph. Using 4 cores, which has not been shown because AVGCC did not degrade the performance of any workload, our QoS approach gets 8.1% improvement over the baseline.

9 Conclusions

We have presented Adaptive Set-Granular Cooperative Caching (ASCC), a new design aimed at last level caches in

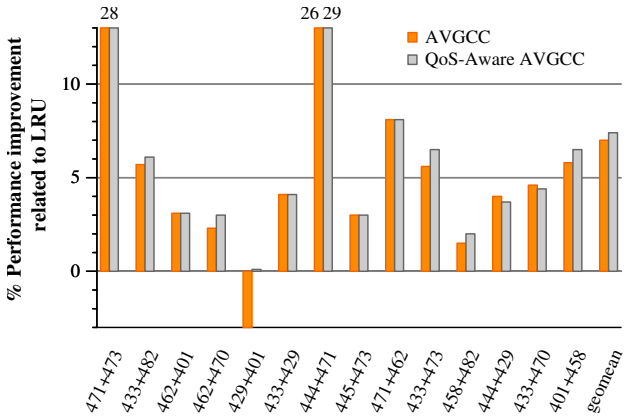

Figure 11. Percentage of performance improvement for QoS-Aware AVGCC and AVGCC over the baseline using 2 cores.

CMP private configurations with a good cost-benefit relation. This cache detects the different demand of sets in order to balance their working set by spilling lines to other caches where the set is underutilized. When the sharing of resources does not suffice, ASCC modifies the cache insertion policy in order to deal with the problem of capacity. Furthermore, we propose Adaptive Variable-Granularity Cooperative Caching (AVGCC), which is able to adapt the granularity with which the sets should be profiled and managed. Also, this design has been improved with a *Quality of Service* mechanism.

As far as we know this is the first approach that uses spills for sharing resources using a set level metric. It is also the first one able to adapt its granularity depending on the behavior of the cache and capable of coordinating a spilling mechanism with a local policy to tackle capacity misses. Finally, it demonstrates the benefits of operating portions of the cache, groups of sets in this case, neither as spillers nor as receivers of lines, that is, not participating in the spilling mechanism.

In a 4-core system running multiprogrammed workloads, AVGCC achieved a performance improvement of 7.8% with respect to the baseline. Also, it clearly outperformed recent proposals both in terms of speedup and fairness, while having a very small storage overhead. The 27% average memory latency reduction and 29% power consumption reduction it provided in these tests are also remarkable. Similar results were obtained in experiments with multithreaded applications.

Future directions for research include tuning the size and limits of saturation counters, as well as exploring other metrics, to obtain a more accurate picture of the state of the cache.

Acknowledgments

This work was supported by the Xunta de Galicia under projects INCITE08PXIB105161PR and *"Consolidación e Estructuración de Unidades de Investigación Competitivas"* 2010/06 and the MICINN, cofunded by FEDER funds, under the grant with reference TIN2010-16735. The authors are also members of the european HiPEAC network of excellence.

References

[1] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *ISCA*, pages 264–276, 2006.

[2] M. K. Qureshi. Adaptive spill-receive for robust high-performance caching in CMPs. In *HPCA*, pages 45–54, 2009.

[3] B. M. Beckmann, M. R. Marty, and D. A. Wood. ASR: Adaptive selective replication for CMP caches. In *MICRO*, pages 443–454, 2006.

[4] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS*, pages 242–252, 2007.

[5] E. Herrero, J. González, and R. Canal. Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors. In *ISCA*, pages 419–428, 2010.

[6] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way cache: Demand-based associativity via global replacement. In ISCA, pages 544–555, June 2005.

[7] D. Rolán, B. B. Fraguela, and R. Doallo. Adaptive line placement with the Set Balancing Cache. In MICRO, pages 529–540, December 2009.

[8] D. Rolán, B. B. Fraguela, and R. Doallo. Reducing capacity and conflict misses using set saturation levels. In *HiPC*, December 2010.

[9] D. Zhan, H. Jiang, and S. C. Seth. STEM: Spatiotemporal management of capacity for intra-core last level caches. In *MICRO*, pages 163–174, 2010.

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

[11] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[12] R. Golla. Niagara 2: A highly threaded server-on-a-chip, 2007.

[13] Model number methodology for the AMD Opteron 4100 and 6100 series processors, 2010.

[14] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *ASPLOS*, pages 234–244, 2000.

[15] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*, pages 164–171, 2001.