Dynamic Last-Level Cache Allocation to Reduce Area and Power Overhead in Directory Coherence Protocols

Mario Lodde[1], Jose Flich[1], and Manuel E. Acacio[2]

[1] Universitat Politècnica de València, Spain
[2] Universidad de Murcia, Spain

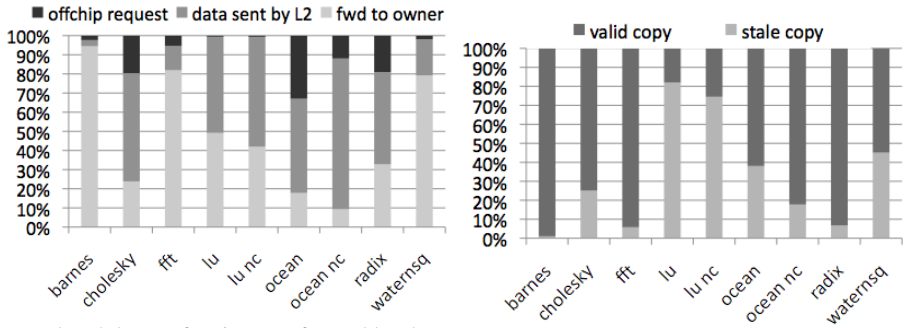
Abstract. Last level caches (LLC) play an important role in current and future chip multiprocessors, since they constitute the last opportunity to avoid expensive off-chip accesses. In a tiled CMP, the LLC is typically shared by all cores but physically distributed along the chip, thus providing a global banked capacity memory with high associativity. The memory hierarchy is orchestrated through a directory-based coherence protocol, typically associated to the LLC banks. The LLC (and directory structure) occupies a significant chip area and has a large contribution on the global chip leakage energy. To counter measure these effects, we provide in this paper a reorganization of the LLC cache and the directory by decoupling tag and data entry allocation, and by exploiting the high percentage of private data typically found in CMP systems. Private blocks are kept in L1 caches whereas LLC area is reorganized to reduce L2 entries while still allowing directory entries for private data, thus, maximizing on-chip memory reuse. This is achieved with no performance drop in terms of execution time. Evaluation results demonstrate a negligible impact on performance while achieving 45% of area saving and 75% of static power saving. For more aggressive designs, we achieve 80% area and 82% static power savings, while impacting performance by 10%.

1 Introduction

Tiled chip multiprocessors (CMPs) have been advocated as the most effective way of organizing future many-core CMPs with dozens of processor cores [1]. These tiled architectures provide a scalable solution for managing the design complexity, and effectively using the resources available in advanced VLSI technologies. A tiled CMP is built by replicating the same tile structure on the chip surface. Each tile typically includes one (or more) processor core, one (or more) level of private caches, part (one bank) of a shared but distributed last-level cache (LLC) and a router to connect the tiles building a network-on-chip (NoC). The shared LLC is typically inclusive with respect to all of the private caches. This means that, at all times, the LLC contains a superset of the blocks in the private caches (Intels Core i7 is a good example [2] of the latter).

Private caches in these designs are kept coherent by means of a directory-based cache coherence protocol implemented in hardware. The directory structure is distributed between the LLC banks, usually included into the tags portion of every cache entry [3]. In this way, each tile keeps the sharing information of the blocks mapped to the LLC bank that it contains. This sharing information comprises two main components: the *state bits* used to codify one of the possible states that the directory can assign to the cache block, and the *sharing code*, that holds the list of current sharers.

C. Kaklamanis et al. (Eds.): Euro-Par 2012, LNCS 7484, pp. 206–218, 2012.



(a) breakdown of actions performed by the LLC when an L1 request is received

(b) % of L2 replacements

Fig. 1. Breakdown of actions and replacements at LLC

Last level caches (LLC) play an important role in ensuring performance since they constitute the last opportunity to avoid expensive off-chip accesses. In this way, current and future CMPs will be equipped with increasingly larger LLCs, occupying a significant fraction of the total chip area and having a large contribution on the global chip leakage energy (as an example, the size of the LLC of the Intel Core i7 can reach up to 15MB as by today [2]). In this work we focus on the design of an effective LLC for future many-core CMPs. In particular, we propose to reorganize the LLC structure to allow for the dynamic allocation of entries at this cache level depending on the specific necessities of every address.

Our work is motivated by the observation that for private blocks (memory blocks requested by a single core) the LLC contains stale data (or, more precisely, the data portion of the cache entry is not needed). While all the cached copies of a shared block have the same value, both in the private and the LLC caches, in the case of a private block (the block is owned by a single private cache) the values of the LLC and the private copies may differ (the block could be in modified state in the private cache). Therefore, the LLC copy of a private block is then probably stale, and useless until the corresponding private cache performs a *writeback* operation on the block. In that situation, the only valid copy of the block would be moved to the LLC. In this way, there are two cases in which the data portion of each LLC entry is needed: a) the block is shared by several cores, and b) a private block has been replaced by the owner private cache. In the rest of situations, keeping the data portion of the LLC entries can be seen as a waste of resources. Figure 1.(a) plots the breakdown of actions performed by the LLC of the 16-core CMP assumed in this work (details about the evaluation environment can be found in Section 3). More than 80% in some cases of the requests are forwarded to the L1 caches, thus do not involve the data portion of the LLC. Therefore, we can see that a large percentage of area and power is wasted in LLC to store private blocks.

On the other hand, LLC replacements (due to limited capacity and associativity) lead to invalidating data in private caches. Figure 1.(b) shows the percentage of LLC replacements of private blocks. As derived from the figure, a large percentage of replacements in the LLC (more than 40% in some applications) are for private blocks. Private data is

only requested the first time the processor wants to operate on it and then is written and read without sending any request to the LLC, thus becoming older in the LLC set and thus becoming quickly selected by the LRU replacement algorithm (even if it is being actively referenced by a processor core).

Taking those results as a reference, we can conclude that an effective organization for the LLC should combine two types of entries: entries with just the tag's portion for private blocks, and regular entries for the rest of the blocks. In this work, we propose to reorganize the LLC structure to allow the dynamic allocation of blocks depending on the block being *private* or *shared*. In particular, we redesign the associated directory with a different (higher) associativity than the L2 data array. The tag and directory array contain information about all the cached blocks, while the data array contains only shared and replaced private blocks. This allows for a smaller LLC with the same performance as private blocks will be kept only at private caches. With our approach, we achieve large savings in static power while not hurting performance. Evaluation results demonstrate average savings of 45% in area and 75% in static power.

Our proposal can be combined with previous works that aim for reducing static power at L2 caches by dynamically powering down cache lines. This is the case of [4]. Notice that in that situation entries in L2 for private blocks (once the L1 cache writes on the block) are powered down. This means extra power saving through a line-level power gating mechanism (similar to cache decay for L1 caches [5]). These strategies are orthogonal to our approach. We provide results for the two mechanisms combined together.

The rest of the paper is organized as follows. In Section 2 we describe our proposal and its impact on area and power overhead. In Section 3, we perform a detailed analysis of performance and power savings. Then, related work is described in Section 4 and the paper is concluded in Section 5.

2 Dynamic L2 Cache Line Allocation

In this work, and without losing generality, we focus on CMPs made of 16 tiles connected through a 2D mesh NoC. Each tile includes a processor core, its private L1 cache and a bank of the L2 cache, which is shared by all the tiles. The whole L2 cache is, thus, distributed and made of 16 banks, each managing a subset of the global address space. From now on, we use the term L2 cache and LLC interchangeably.

We assume MESI states for the L1 caches. The *M* state is used when the block is private and has been modified; *E* is used when the block is private and not modified; *S* means read-only permissions over the block (the block is potentially shared); and *I* denotes a block that is invalid or not cached. Additionally, each directory entry (associated with each L2 cache entry) will have the following fields: 1) *Cache line state* field being *P* when the block is private, *S* when the block is shared with no owner, *C* when the block is only cached in L2, and *I* when the block is not cached (invalid entry). 2) *Owner* field being a pointer to the owner L1, which can provide the block to future requestors. 3) *Sharers List* field, being the list of the L1 caches which share the block.

Figure 2 shows how an L2 block switches between states depending on the requests received by L1 caches (transient states are omitted for the sake of clarity). At first, the block is not cached on chip (state *I*). Upon a write (GETX) or a read (GETS) request, the

block is fetched from main memory and sent to the L1 requestor, which is now the owner of the block and holds a private copy (state *P*). At this point, a write request from another core will be forwarded to the owner, which will send the data to the requestor and invalidate its copy (the requestor will become the new owner). A read request (GETS) will also be forwarded to the owner, but in this case it will not invalidate its copy. Instead, it will provide the data to the requestor but also keep a copy of the block with read permission. However, the block state in the L2 bank will switch from *P* to *S*.

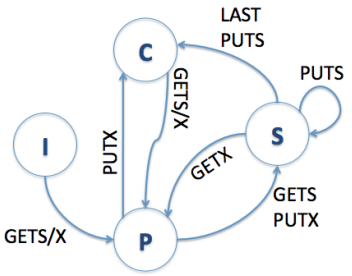


Fig. 2. Simplified FSM for the L2 cache (MESI protocol)

If the block is in *P* state and the owner replaces the block (PUTS or PUTX), then the only on-chip copy of the block lies in the corresponding L2 bank, which switches to state *C*. Further requests will be served using the L2 copy of the block.

Usually, there is a 1:1 relationship between the tag/directory and the data portions of the LLC. A different design can be chosen, with fewer data entries than tag/directory entries. If the block is private (state *P* in the directory), then only the tag/directory is allocated, as the only valid copy will be held by the owner L1. If, on the contrary, the block is shared or only cached in the L2 bank (state *S* and *C* in the directory), then both the tag/directory and the cache line are allocated.

We can reorganize an L2-directory bank by reducing the associativity in L2 and keeping or using that area for more tag/directory entries in the directory structure, so we break the 1:1 relationship. This will reduce replacements of blocks in *P* state. To do this, we keep the associativity of the tag/directory array to 16 while reducing the L2 data array associativity from 16 down to 8. This means that only the first eight ways of the tag array can store information about a block in state *S* or *C*, which data will be saved in the corresponding way of the data array, and all the 16 ways can store information about a private block. We also stress the inequality by further reducing L2 data associativity down to four and even two. Notice that this will constraint the area devoted to a shared data in L2 but will not compromise private data, as will be tracked by the directory and will be allocated in the L1 caches. With these ratios (from 1:2 to 1:8) large savings in L2 area are expected.

It has to be noted that one could think of reducing L2 size by reducing the number of sets, instead of the number of ways (see Figure 3). However, this could compromise cache capacity depending on the data set. Indeed, having less sets would lead to have less entries for shared data, while by reducing the associativity, cache capacity for

shared data will not be compromised (as long as shared data does not conflict in the same set). Notice that we reduce ways as those are, expectedly, used by private blocks.

Another direction one could take is reducing in the same degree both the associativity in the L2 and in the directory (see Figure 3). However, in that case we would incur in high performance penalty (as will be seen later) as the associativity in the directory must be proportional and in line with the associativity in L1 caches. Simply, the directory will not have enough ways to avoid conflicts between both shared and private blocks.

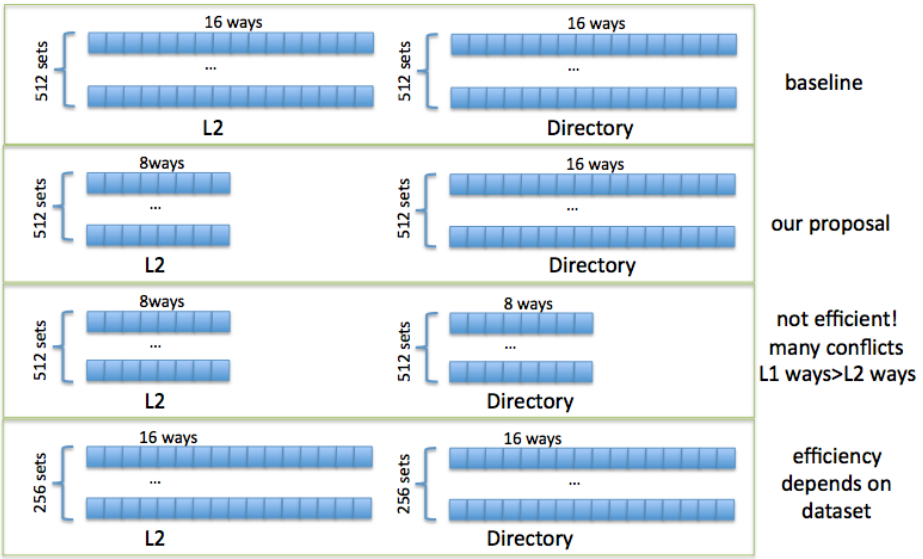


Fig. 3. Different LLC configurations by changing the number of sets and the associativity of the L2 and directory structures

Figure 4 shows an example of an L2 cache reorganized. Only the first four ways of the tag array may keep information about shared or cached blocks, which will be saved in the same way of the data array. The remaining four sets of the tag array are devoted to private blocks. However, private blocks can also be mapped in the first four ways of the tag array. In this case, the information included in the corresponding set of the data array would not be useful.[1]

Assuming an L2 cache with 4 ways and a directory with 8 ways, when a block switches from *P* to *S* or *C*, its entry must be moved if it doesn't lie in the first four ways. This may trigger the replacement of another block if all the first four ways are already in use, which means that the data array set is full. Thus, the automata to manage the L2 and directory needs to be slightly modified.

[1] With orthogonal power saving techniques as [4] these entries can be powered down. Its impact is later analyzed.

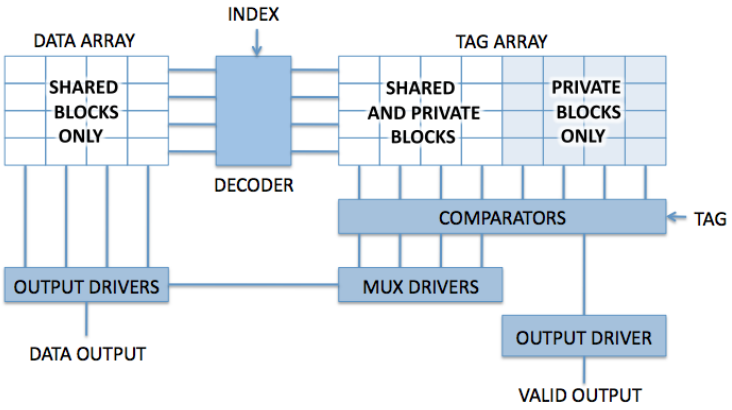


Fig. 4. Example of LLC reorganization

2.1 Replacement Policy

An LRU counter per way is used to implement the replacement policy. The counters are used in the classical way: each time the L2 receives a request for a block, all the counters with a lower value are incremented and the block counter is set to zero. When a new block is requested and all the set entries are already in use, the entry with the highest LRU counter value is selected. With the organization we propose, replacements can be triggered also when a block which is already cached must be saved in the L2 cache. As shown in Figure 2, this happens when an owner invalidates its private copy or a read request is received for a private block: the L2 state switches from P to C in the first case or from P to S in the second case. In both cases, a data entry must be allocated for the block in the L2 cache. If the first four ways are already allocated to other blocks, the replacement policy will choose the way with the highest LRU counter only between the first four entries (even though the entry with the highest LRU value could be one of the remaining ones). Notice, that LRU counters are updated for all the directory entries, thus not having two entries with the same number.

2.2 Dynamic Power Techniques

With our technique, truly shared data is promoted in the L2 data array, and private data is just tracked in the directory. However, it may happen that for a given L2-directory set more than half of the entries are for private blocks, thus not all the entries in the L2 cache will be used. In such situation, these L2 data entries are wasting energy. To mitigate this effect, our approach can be complemented with dynamic power-off strategies as [4], in which private blocks that could be allocated in the L2 cache lead to powering down the L2 entry. This can be achieved using ”sleep transistors” at each cache line to eliminate the most part of the leakage current, as proposed by Kaxiras et al. [5] for L1 caches. As in [5], we use Powell’s gated Vdd design [6] at cache line level, inserting a transistor

between the ground and each L2 cache line to reduce the leakage current to a negligible level.

When combined, the different transitions of a block *A* in the L2 cache can be summarized as follows:

- When an L1 cache requests block *A*, which is not cached on-chip, the L2 issues an off-chip request, allocates a tag entry to the block (which can be anyone of the entries) and marks the block as private; if the tag entry is one of the first half, its corresponding data entry is powered-off.
- Subsequent write requests will cause a change of the owner but the state of the block in L2 cache will remain P.
- When the owner replaces its copy, it must be saved in the L2 cache. The same happens if the L2 receives a read request for a private block, which will become shared. If *A* is mapped in the first half of the tag ways, the corresponding data entry must be powered-on. If, on the contrary, *A* is mapped in the second half, it will trigger a potential internal swap, selecting one entry from the first half (using the LRU algorithm). The selected entry can be in one of the following states:
  - Private: this block and *A* are swapped, the data entry is powered on to save the write back copy of *A*.
  - Shared/Cached: this entry will be replaced and allocated to *A*.
- If a write request (in case *A* is in state S) or any request (in case *A* is in state C) is received, the block will be again treated as a private block and the data entry must be powered-off.

3 Performance Evaluation

We evaluate our proposal by using gMemNoCsim and Graphite [7] simulators. gMemNoCsim is an in-house event-driven cycle accurate cache hierarchy and NoC simulator. gMemNoCsim is embedded in Graphite, which allows execution-driven simulation of applications. Application's memory accesses are tracked by Graphite and fed into gMemNoCsim for an accurate memory coherency and on-chip network modeling. Graphite threads are blocked until memory accesses are resolved by gMemNoCSim. We implemented a two-level MESI coherence protocol, and the modifications needed to implement our block allocation and replacement policy (later we do the same for a MOESI protocol). Five different L2 designs have been evaluated and compared (block size is set to 64 bytes):

1. **L2-512-16_D-512-16**. An 512KB L2 bank with 512 sets and 16 ways. The directory also has 512 sets and 16 ways (1:1 ratio). This is the baseline for comparison.
2. **L2-512-8_D-512-16**. An 256KB L2 bank with 512 sets and 8 ways. The directory also has 512 sets but keeps 16 ways (1:2 ratio).
3. **L2-512-4_D-512-16**. An 128KB L2 bank with 512 sets and 4 ways. The directory keeps the same with 512 sets and 16 ways (1:4 ratio).
4. **L2-512-2_D-512-16**. An 64KB L2 bank with 512 sets and 2 ways. The directory keeps the same with 512 sets and 16 ways (1:8 ratio).

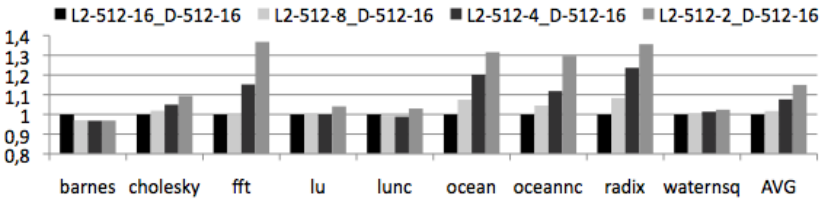


Fig. 5. Normalized execution time. MESI protocol with L2 banks with 512 sets.

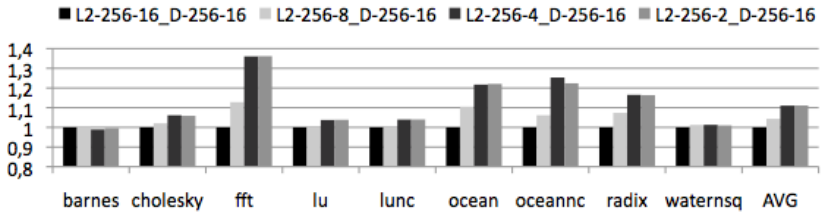


Fig. 6. Normalized execution time. MESI protocol with L2 banks with 256 sets.

In addition, we use a smaller L2 cache of 256KB as the baseline. In this case (**L2-256-16_D-256-16**), 256 sets and an associativity of 16 is used for both the L2 and the directory. Our designs on top of this baseline are **L2-256-8_D-256-16** (1:2 ratio, 128KB), **L2-256-4_D-256-16** (1:4 ratio, 64KB), and **L2-256-2_D-256-16** (1:8 ratio, 32KB).

The system is made of 16 tiles with one processor in each tile and with a private 32KB L1 data cache (with 256 sets and 4 ways). Each tile includes also the L2 bank and the associated directory. All the tiles are connected through a 2D mesh topology using the XY routing algorithm. In a first test every configuration does not incorporate any sleep transistor technology. Later we evaluate the combination of our technique with those. We ran various SPLASH-2 applications with these cache organizations.

Figure 5 shows the execution time for the different applications, normalized to the case of the first baseline L2-512-16_D-512-16. As can be seen, some applications have no impact on execution time when L2 banks are reduced. Indeed, in BARNES, CHOLESKY, LU, LUNC, and WATERNSQ, the L2 could be reduced by a factor of 8 (L2-512-2_D-512-16) with practically no impact on performance. On the other hand, other applications can be sensitive to L2 cache capacity to shared blocks. Anyway, by averaging, we can see that a good tradeoff is reducing L2 cache by half, which on average leads to only 1.7% performance decrease. Further reductions in area will tend to 7.5% performance degradation for an L2 reduction factor of 4 and to 15% for a reduction factor of 8.

For the case of smaller L2 banks (those with 256 sets), Figure 6 shows the execution time of applications. We can see similar trends with large savings (up to a factor of 8x) in area with no performance degradation, and others with some impact (up to 35%). On average, a reduction of 2x in L2 size have no large impact.

We use Cacti 5.3 [8] to compute area and leakage for the different L2-directory configurations. In Figure 7 we can see how area needs compare to the different evaluated

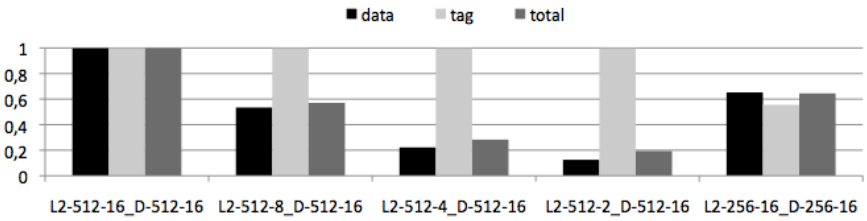


Fig. 7. Normalized LLC area occupancy

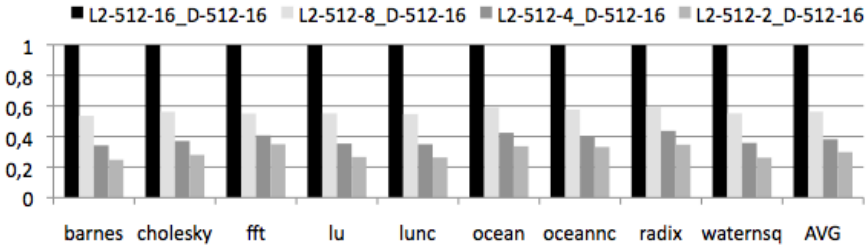


Fig. 8. Normalized L2 leakage energy. MESI protocol.

designs. Each component is normalized to the baseline design (L2-512-16_D-512-16). Tag array's area is the same for the first four designs, while in L2-256-16_D-256-16 tags take roughly half the area as the overall associativity is reduced. As far as data array is concerned, the area needs decrease with the associativity of each design. Even though L2-512-8_D-512-16 and L2-256-16_D-256-16 have the same data array size, the area of the former is lower than the area of the later, due to its lower associativity (lower number of comparators).

Figure 8 shows the leakage energy consumed by the L2 banks taking into account the entire execution of each application, and normalized to the baseline (L2-512-16_D-512-16). Leakage is reduced up to 80% due to the cache reorganization. This saving is proportional to the reduction ratio performed.

Figure 9 shows leakage savings when our proposal is combined with [6]. We compare the previous four cases and a baseline 256KB cache (L2-256-16_D-256-16). In both baselines, caches have all data entries powered on during the whole execution time, while the proposals power-on the L2 data entries only when needed, as perviously described. The average leakage energy is reduced on average by 75% (for 1:2 ratio), 83% (for 1:4 ratio) and 89% (for 1:8 ratio). Depending on the application, we achieve up to 98% in leakage energy savings (BARNES).

3.1 Benefits When Using MOESI Protocol

Another appealing protocol which can be implemented in L1 caches is MOESI. It behaves like the MESI protocol but when an owner (which has its block in state M or E) receives a forwarded GETS its state becomes O (and not S like in MESI protocol).

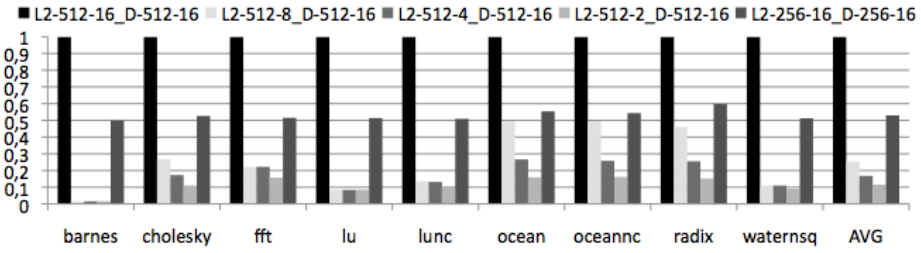


Fig. 9. Normalized L2 leakage energy (MESI) with sleep transistors

This means it remains the owner of the block with read permission, and when the L2 receives a request, it will still be forwarded to this L1. This is inefficient in a typical memory hierarchy since it takes one more step to provide the data to the requestor, but can largely benefit from our approach. Indeed, shared blocks in state O do not need to be allocated in L2 banks, thus, being all the requests forwarded to the owner. The L2 state diagram when a MOESI protocol is used in L1 caches is shown in Figure 10. A block keeps switching between states P and O until the owner invalidates its copy, and only then an L2 cache line must be allocated. In Figure 11 we compare the execution time for the different cache configurations with L1s that use a MOESI protocol (the figure shows the cases for 512 sets). However, for the sake of fairness, we use the MESI

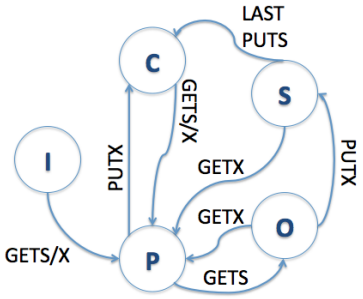


Fig. 10. Simplified FSM for the L2 cache (MOESI protocol)

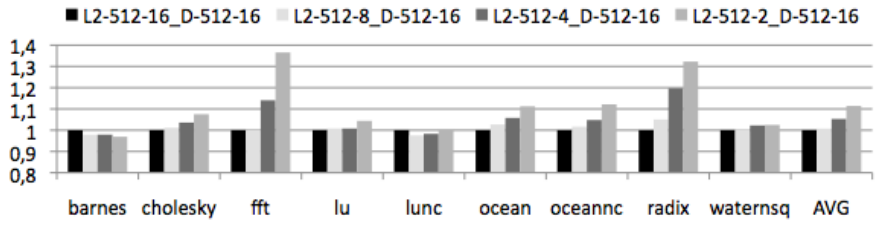


Fig. 11. Normalized execution time. MOESI (for 1:x ratio proposals) and MESI (for baseline).

protocol for the baseline configuration (which works better than when using MOESI, due to the extra indirection). For our proposals, however, we use the MOESI protocol.

As shown, our organization takes advantage of the O state since less blocks in state S and C must be invalidated. The average penalty when using a data array with lower associativity is now 0.7% (with a 1:2 ratio), 5.2% (with a 1:4 ratio) and 11.4 % (with a 1:8 ratio), while saving 43%, 72% and 81% of area and 75%, 82% and 83% of static power, respectively (these results are not shown due to space constraints).

4 Related Work

As the cache hierarchy is currently the chip component which has more impact in area and power, many efforts have been made in reducing its energy requirements. At circuit level, different techniques have been proposed to reduce cache leakage. Powell et al. [6] propose the gated-Vdd technique which powers down the L2 entries used for private blocks. Similar techniques have been proposed to reduce the supply voltage enough to reduce leakage without destroying the content of the cell, as done with drowsy [9] and superdrowsy [10] caches. The latter techniques have various drawbacks compared to the destructive technique proposed by [6], being more difficult to implement and saving less leakage since a certain voltage has to be provided to the cell to keep the data.

At higher level, alternative cache architectures have been proposed to reduce static and/or dynamic power. Savings can be achieved by modifying cache parameters like cache size and cache associativity [12, 13]. Other efforts have been made to reduce the number of cache accesses, by using snoop filters [14, 15], way predictors [16] or filter caches [17]. Various proposals turn off L1 or L2 cache ways based on different prediction techniques. As an example, Kaxiras et al [5] propose to turn off L1 cache entries which are not expected to be reused. Abella et al [11] propose a different predictor to turn off unused L2 cache entries. Li et al [4] use different policies to turn off L2 cache entries when block copies are replicated in an L1, and evaluate both conservative and destructive voltage gating techniques.

All these works always assume a 1:1 relationship between L2 entries and directory entries. In addition, our proposal is orthogonal to all these works, since in our work we simply remove the area devoted mainly to private blocks in L2. Indeed, we reduce L2 cache size by reducing its associativity, while keeping directory associativity, which is vital to keep track all the on-chip blocks, either shared or private. Indeed, we demonstrated in this paper our technique can be complemented with the one presented in [6].

5 Conclusions

In this paper we have proposed an effective method to significantly reduce the area of LLC caches in a CMP system. The reduction comes from the idea of keeping private blocks in L1 caches and not using L2 entries for such blocks. Although the idea is not new, in our approach we physically redimension the LLC cache and its associated directory and provide a 1:x approach, where different associativity degrees are used in both L2 and directory caches. The L2 cache associativity is lower in order to accommodate

only shared blocks, while directory associativity is kept to keep track of all the blocks in the chip, either private or shared blocks.

Results demonstrate large savings in L2 area and static power consumption. With a MESI procotol, the L2 cache size can be halved with no impact on performance and with a rough reduction of leakage of 50%. More aggressive designs (75% of L2 size reduction) lead to low performance penalties but with very large savings in power. We also demonstrate in this paper that our technique can be complemented with power-gating mechanisms in the L2 cache.

Acknowledgement. This work has been supported by the VIRTICAL project (grant agreement n 288574) which is funded by the European Commission within the Research Programme FP7.

References

1. Tile-gx processors family, http://www.tilera.com/products/TILE-Gx.php
2. Intel Core i7 technical Specifications, http://www.intel.com/products/processor/corei7ee/specifications.htm
3. Zhang, M., Asanović, K.: Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In: Proc. of the 32nd Int'l Symposium on Computer Architecture (ISCA-32), pp. 336–345 (2005)
4. Li, L., Kadayif, I., Tsai, Y.-F., Vijaykrishnan, N., Kandemir, M., Irwin, M.J., Sivasubramaniam, A.: Leakage energy management in cache hierarchies. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (2002)
5. Kaxiras, S., Hu, Z.: Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In: Proc. of the 28th Int'l Symposium on Computer Architecture (ISCA 2001) (May 2001)
6. Powell, M., Yang, S.-H., Falsafi, B., Roy, K., Vijaykumar, T.N.: Gated-Vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In: Proceedings of the International Symposium on Low Power Electronics and Design, ISLPED 2000 (2000)
7. Miller, J.E., Kasture, H., Kurian, G., Gruenwald III, C., Beckmann, N., Celio, C., Eastep, J., Agarwal, A.: Graphite: A Distributed Parallel Simulator for Multicores. In: The 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA) (January 2010)
8. Cacti 5 Technical Report, http://www.hpl.hp.com/techreports/2008/HPL-2008-20.html
9. Flautner, K., Kim, N.S., Martin, S., Blaauw, D., Mudge, T.: Drowsy caches: Simple techniques for reducing leakage power. In: Proceedings of the ACM/IEEE 29th International Symposium on Computer Architecture, ISCA 2002 (2002)
10. Kim, N.S., Flautner, K., Blaauw, D., Mudge, T.: Single-VDD and Single-VT super- drowsy techniques for low-leakage high-performance instruction caches. In: Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design, ISLPED 2004 (2004)
11. Abella, J., Gonzales, A., Vera, X., O'Boule, M.F.P.: IATAC: A Smart Predictor to Turn-off L2 Cache Lines. ACM Transactions on Architecture and Code Optimization 2(1) (March 2005)

12. Drophso, S., Buyuktosunoglu, A., Balasubramonian, R., Albonesi, D.H., Dwarkadas, S., Semeraro, G., Magklis, G., Scott, M.L.: Integrating adaptive on-chip storage structures for reduced dynamic power. In: Proceedings of the IEEE 11th International Conference on Parallel Architectures and Compilation Techniques, PACT 2002 (2002)
13. Zhang, C., Vahid, F., Najjar, W.: A highly configurable cache architecture for embedded systems. In: Proceedings of the ACM/IEEE 30th International Symposium on Computer Architecture, ISCA 2003 (2003)
14. Moshovos, A., Memik, G., Falsafi, B., Choudhary, A.: Jetty: Filtering snoops for reduced energy consumption in SMP servers. In: Proceedings of International Symposium on High Performance Computer Architecture (HPCA 2001) (January 2001)
15. Salapura, V., Blumrich, M., Gara, A.: Design and implementation of the Blue Gene/P snoop filter. In: Proceedings of International Symposium on High Performance Computer Architecture (HPCA 2007) (February 2007)
16. Inoue, K., Ishihara, T., Muruami, K.: Way-predictive set-associative cache for high performance and low energy consumption. In: Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design, ISLPED 1999 (1999)
17. Kin, J., Gupta, M., Mangione-Smith, W.-H.: The filter cache: An energy efficient memory structure. In: Proceedings of the ACM/IEEE 30th International Symposium on Microarchitecture, MICRO 1997 (1997)