Elastic Cooperative Caching: An Autonomous Dynamically Adaptive Memory Hierarchy for Chip Multiprocessors

Enric Herrero[1] eherrero@ac.upc.edu
José González[2] pepe.gonzalez@intel.com
Ramon Canal[1] rcanal@ac.upc.edu

[1]Dept. d'Arquitectura de Computadors, Universitat Politècnica de Catalunya
[2]Larrabee Architecture Group, Intel Barcelona

ABSTRACT

Next generation tiled microarchitectures are going to be limited by off-chip misses and by on-chip network usage. Furthermore, these platforms will run an heterogeneous mix of applications with very different memory needs, leading to significant optimization opportunities. Existing adaptive memory hierarchies use either centralized structures that limit the scalability or software based resource allocation that increases programming complexity.

We propose Elastic Cooperative Caching, a dynamic and scalable memory hierarchy that adapts automatically and autonomously to application behavior for each node. Our configuration uses elastic shared/private caches with fully autonomous and distributed repartitioning units for better scalability. Furthermore, we have extended our elastic configuration with an Adaptive Spilling mechanism to use the shared cache space only when it can produce a performance improvement. Elastic caches allow both the creation of big local private caches for threads with high reuse of private data and the creation of big shared spaces from unused caches. Local data allocation in private regions allows to reduce network usage and efficient cache partitioning allows to reduce off-chip misses.

The proposed scheme outperforms previous proposals by a minimum of 12% (on average across the benchmarks) and reduces the number of offchip misses by 16%. Plus, the dynamic and autonomous management of cache resources avoids the reallocation of cache blocks without reuse which results in an increase in energy efficiency of 24%.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design Styles-*Cache memories*; C.1.4 [Processor Architectures]: Parallel Architectures-*Distributed architectures*

General Terms: Design, Experimentation, Performance.

Keywords: Chip multiprocessors, memory hierarchy, elastic cooperative caching, tiled microarchitectures.

1. INTRODUCTION

Chip multiprocessors are now widely used and, due to the limited parallelism of most commercial applications, are expected to run multiple heterogeneous workloads simultaneously. In this environment the behavior of each of these applications can be very different and therefore lead to different cache requirements. Recent studies [16] show that cache partitioning has a significant performance impact in runtime execution and that dynamic configurations can adapt to the program's time-varying phase behavior and improve performance.

One example of the optimization opportunities that adaptive configurations can exploit is the simultaneous execution of streaming and cache bound applications. For instance, audio and video streaming are commonly executed simultaneously with text editors, web browsers or antivirus in desktop computers. In general, streaming applications do not take advantage of the upper levels of the memory hierarchy and introduce a high amount of data in the caches that is not going to be reused. This inefficiency is aggravated in shared caches by the eviction of blocks from other applications that could be eventually reused. Therefore, it is interesting to have a memory hierarchy that provides some kind of intelligent control to distribute resources fairly and take advantage of the differences among applications. Ideally, the reallocation of resources should be managed autonomously through hardware arbiters to avoid adding extra complexity to the software layer. In addition, next generation memory hierarchies should provide the elasticity to offer the low latency advantages of private caches and the low off-chip miss rate of shared caches. Future tiled microarchitectures also need scalable structures to allow increased levels of parallelism.

Prior work has been done to distribute cache resources dynamically either by software or hardware. Organizations with software allocation [4, 13, 21] increase the programming complexity and, in some cases, use centralized structures that limit the scalability. On the other hand, organizations with hardware allocation either use snooping mechanisms [9] or have a centralized structure, like the last-level cache [23, 24] or the directory [12], that limit the scalability. In addition, most existing solutions do not consider applications with shared data since resources are divided among threads without considering if replication is desirable.

To solve the scalability issues of existing solutions, we propose the Elastic Cooperative Caching (ElasticCC). ElasticCC uses distributed cache partitioning units able to operate autonomously with local information, allowing the redistribution of cache resources without centralized structures or intensive communication among nodes. In our configuration, Shared/Private caches are used in each node together with its own repartitioning unit. Shared regions of all caches create a big shared cache to store evicted blocks from the most active private regions and try to reduce the number of off-chip misses. On the other hand, private regions allow big local private caches when all applications have similar cache requirements. The hardware overhead of the proposed solution is minimal and it does not require software support. Furthermore, we have extended our dynamic configuration with Adaptive Spilling, a technique to improve the usage of the shared cache space.

The proposed scheme outperforms previous proposals such as the Distributed Cooperative Caching and the Adaptive Selective Replication, achieving an average speed-up of 27% and 12% respectively. At the same time, it reduces the number of off-chip misses by 19% and 16% due to the more efficient cache allocation and it increases energy-efficiency by 71% and 24% due to the reallocation avoidance of non-reused cache blocks.

This paper makes the following contributions:

- We introduce Elastic Cooperative Caching, a distributed cache partitioning mechanism that allows autonomous, scalable, and elastic cache behavior based on application needs for next generation tiled microarchitectures.

- We present the first distributed cache repartitioning mechanism able to operate only with local information.

- We propose a mechanism to further optimize the reallocation of evicted blocks in the shared cache space that is also scalable and relies only on local information.

2. BACKGROUND

There is extensive prior research on the memory hierarchy of chip multiprocessors. This work can be divided between static and dynamic resource partition mechanisms.

In static resource partition mechanisms [2, 3, 6, 11, 14, 19, 25, 28] all threads have the same priority and the amount of cache assigned to each thread is changed through the coherence protocol and replacement mechanisms. These organizations either have inter-thread interferences (e.g. Cache-intensive threads may degrade performance of other applications by replacing their blocks) or are not able to give all the cache space to a single thread if the others are not using the cache. This is logical if we consider that the cache space is statically mapped to threads, and can lead to a non-optimal usage of resources in unbalanced workloads.

If we want to optimize cache allocation to reduce off-chip misses, a repartitioning mechanism is desirable to be able to change the cache size assigned to every thread. Dynamic resource partition mechanisms dynamically modify the amount of memory that is assigned to every node and eliminate inter-thread cache conflicts by allocating independent partitions of resources. These resources can be divided in banks, sets or ways and require an arbitration mechanism that can be software or hardware based.

Software-based dynamic configurations delegate resource allocation to the OS. Most of these organizations divide resources in independent sections to be able to provide QoS [4, 13, 21, 10]. Virtual Private Caches [21] divide resources in ways and implement a hardware arbiter to dynamically distribute the unallocated space in accordance to a fairness policy. In the Cooperative Cache Partitioning [4] resources are not only partitioned spatially, but also in time. They apply Multiple Time-sharing Partitions to expand the cache capacity of some threads at a given time and increase throughput. Iyer [13] studies different mechanisms to allow QoS in the memory hierarchy. However, it is focused on the distribution of a unified shared last-level cache. Liu et al. present the Shared Processor-Based Split L2 [17], a cache configuration that also allows software-based distribution of cache resources. The purpose of this configuration is not to provide QoS but to be able to select private or shared caches depending on the application. This organization is limited by a snoop based protocol to access cache data that requires broadcast messages to all cache banks. Finally, R-NUCA proposes a variable block mapping depending on the kind of data; allocating private data close to the requesting node and replicating shared read-only blocks. However, it requires OS support and works at a page-level granularity. These configurations are limited by centralized structures in some cases and by a software-based arbitration that increases programming complexity.

Hardware-based dynamic organizations [9, 12, 23, 24], on the other hand, are able to implement the repartitioning policy in hardware, reducing the programming complexity. They are based on performance counters to measure the benefit of increasing the cache size for each thread. Adaptive Set Pinning [24] divides resources of a last level cache through sets by granting the replacement ownership to a thread. This ownership is varied dynamically to optimize the cache usage. Extra small private caches are required to allocate blocks of threads that don't have the ownership of the corresponding set. In the case of the NUCA Substrate [12], a centralized directory and a shared pool of small cache banks with different degrees of sharing is proposed. Dynamic mapping allows data to be stored in multiple banks but requires a tag check of all the possible destinations. The Dynamic Spill-Receive (DSR) [22], on the other hand, uses private L2 caches to allocate evicted blocks from other caches if a performance improvement is expected. The ability to reallocate or accept evicted blocks is decided through set dueling using miss information of all caches. Therefore, for a given cache, miss information of all caches must be provided for the tested sets to decide its behavior. This makes this technique interesting for a small number of nodes but unfeasible for a large number of them due to the communication overhead that this would entail. And finally, several works [9, 23] have appeared recently that adapt the cache size through the column caching technique [5]. Column caching is a cache partitioning mechanism that restricts the available number of sets when allocating a block, therefore enabling the cache to be partitioned. In the case of Utility-Based Cache Partitioning (Utility) [23] a single last-level cache is partitioned and assigned among threads. On the other hand, in the Adaptive Shared/Private NUCA (ASP-NUCA) [9] a cache for each node is divided into shared and private cache space according to thread requirements.

Cache partitioning is controlled by the repartitioning unit, depicted in Figure 1 for ASP-NUCA. Repartitioning in ASP-NUCA is based on the expected reduction in misses for each thread. This technique, also used in the Utility-Based Cache Partitioning [23], stores the tags of the evicted blocks on a

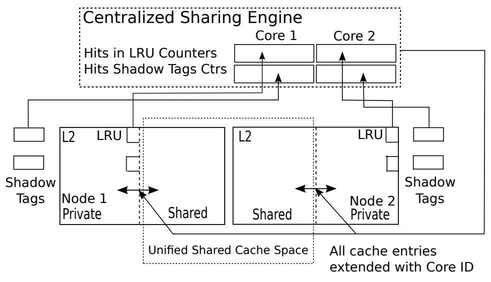

Figure 1: Adaptive Shared/Private NUCA Repartitioning Unit.

replacement in the private region. These tags are known as shadow tags. On a cache miss, the shadow tag is checked to detect the potential benefit of increasing the cache size. On the other hand, on a hit in the LRU block the second counter is updated to detect the potential degradation of reducing the cache size.

ASP-NUCA uses all the shared cache space as a centralized shared cache where all nodes can store as many blocks as indicated by a variable set for each of them. To be able to track the number of blocks from each core, ASP-NUCA also requires a CoreID field for each cache entry. When a block needs to be allocated in the shared region all shared partitions are checked to count the number of allocated blocks for that core. If there are less than allowed, the LRU block from all shared partitions is evicted. If not, the LRU block from that core is evicted.

The usage of shadow tags in all sets implies a significant hardware overhead and, therefore, only a few sets are monitored (6%). Even after doing this reduction the hardware overhead of ASP-NUCA is: a CoreID field in all cache entries, a shadow tag for the 6% of all cache sets, and two counters per node.

In addition to that, ASP-NUCA never changes the amount of private and shared cache space since every time that a private region in one node is increased, it is also decreased in another node. Therefore, this method is not optimal when all nodes execute independent tasks and big private caches would be desirable for each of them or when all threads share the data and a big shared cache would be better.

As we have seen, none of these techniques is suitable for large tiled microarchitectures because they require either a centralized cache or a centralized repartitioning unit that limits the scalability.

3. TAXONOMY OF APPLICATIONS

The design of dynamic memory hierarchies requires the implementation of policies to distribute cache resources. Applications can benefit from cache memory up to a certain point and the allocation mechanism must know the individual needs of each of them. In this section we study the behavior of applications in order to analyze the potential performance benefit of reallocating cache resources.

Figure 2 shows a characterization of the SpecOMP benchmarks for varying number of ways and sizes. It is important to note that application behavior is highly dependent on the section of code being executed and the cache size. We have evaluated the SpecOMP benchmark set with 8 nodes, private 16kb-4way L1s and private L2s of varying size and associativity.[1] Size is incremented with the addition of extra ways to evaluate the benefits in way-partitioned caches. Execution is started in each benchmark's most important parallel regions once all data structures are initialized. Three different parameters are shown in Figure 2; normalized performance compared to the 8way 256kb configuration, number of misses per instruction, and the sharing relation. Each row uses different graph scales to allow a better clarity in application behavior variations. Sharing relation shows the average number of nodes that share a block before its eviction and indicates if threads have independent data sets. Therefore, this parameter shows the potential savings in cache space a shared cache would provide by reducing replication.

Previous studies of dynamic cache repartitioning [23] have divided applications in three categories; low utility, high utility and saturating utility. However, after evaluating all applications we have detected that these categories do not consider the amount of sharing, which is important for an efficient cache partitioning.

Therefore we have divided the benchmarks into four categories:

Saturating Utility: These type of applications are characterized by having a small working set that fits in the cache. Therefore, granting more cache space to them has no impact on their performance. These applications are characterized by improving performance with each increase of cache size until the working set fits in it. If extra cache space is provided performance is not affected (e.g. Equake).

Low Utility: This category is for benchmarks with low temporal locality that make an intensive use of the memory hierarchy but do not have reuse. These applications are especially harmful for competing benchmarks that would benefit from more cache space. They are characterized by not improving performance when we increase cache sizes (e.g. Gafort).

Shared High Utility: In these applications, there are several threads that share a large number of blocks. Therefore, to optimize cache usage replication should be reduced for shared blocks reducing private regions. Replication in highly reused blocks, however, is still granted by L1 caches. These applications are characterized by a high sharing (e.g. Ammp).

Private High Utility: Finally, Private High Utility applications are those that benefit from larger levels of memory hierarchy but do not share data between threads. They are characterized by always improving performance when cache size is increased and by low data sharing among nodes (e.g. Swim).

Table 1 shows the classification of the SpecOMP benchmarks in the previously defined categories with tested benchmarks in bold. A multiprogrammed benchmark set is used to see the influence of memory intensive benchmarks over applications with lower cache requirements. We have selected benchmarks from each category to study the behavior of our dynamic configurations under all possible combinations.

4. ELASTIC COOPERATIVE CACHING

In this section we describe the Elastic Cooperative Caching (ElasticCC), a distributed and dynamic memory hierarchy that adapts autonomously to application behavior. We have

[1]Further details of the simulation environment can be found in Section 4.

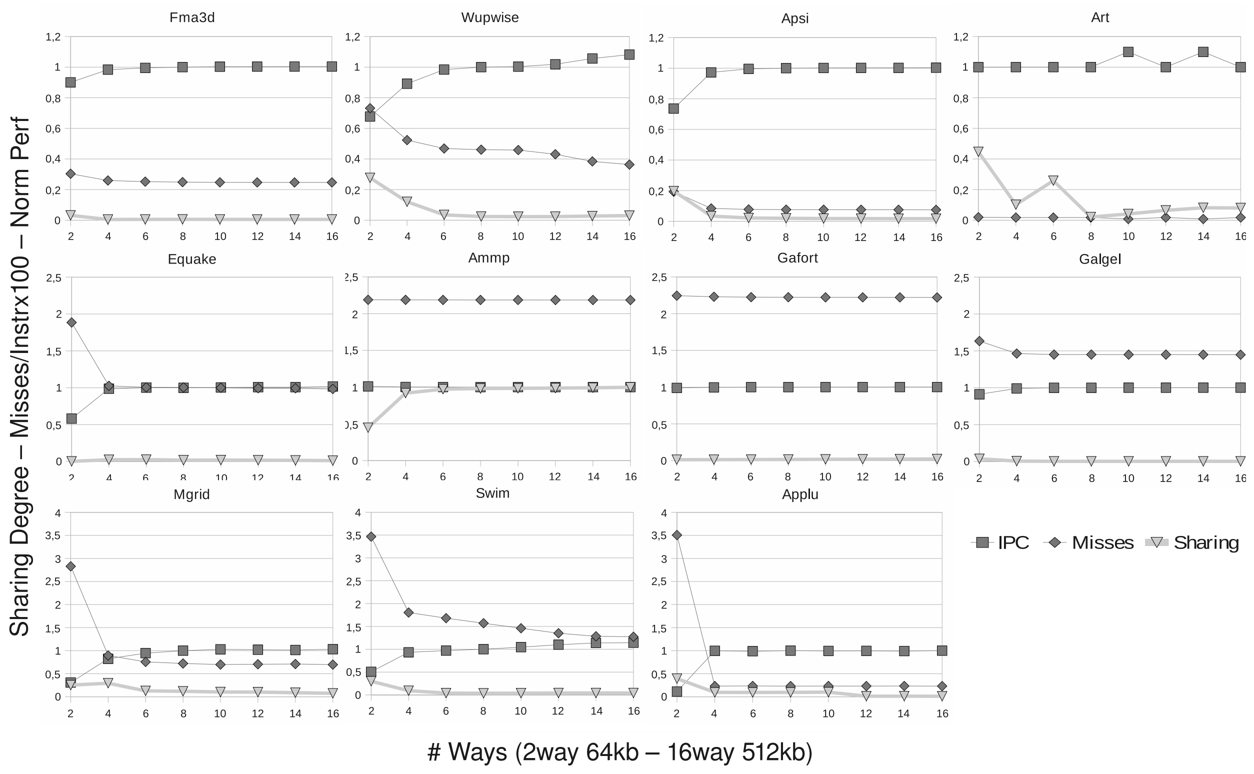

Figure 2: Spec OMP 2001 benchmark characteristics for different L2 sizes.

Type	Benchmark
Saturating Utility	Applu, **Apsi**, **Art**, **Equake**, Fma3d
Low Utility	**Gafort**, Galgel
Shared High Utility	**Ammp**
Private High Utility	Mgrid, **Swim**, Wupwise

Table 1: Benchmark Classification

focused on designing a scalable solution suited for next generation tiled microarchitectures.

4.1 Memory Coherence

To be able to have a scalable memory hierarchy organization it is important that all parts can be distributed to avoid bottlenecks. Therefore, in this paper we use the Distributed Coherence Engines (DCEs) [11] to grant coherence in the ElasticCC and in the evaluated version of Adaptive Selective Replication. DCEs are directory caches responsible for maintaining coherence for a portion of the address space. These directories can be distributed across the chip and have been used in the ElasticCC and in the evaluated distributed version of Adaptive Selective Replication.

Figure 3 shows the working principle of DCEs. When a request from a given node has missed in local L1 and L2 caches the request is forwarded to the DCE where the block address is mapped. If the requested block is located in another cache the DCE will forward the request to that node so the data is transfered to the requester.

Distributed Cooperative Caching (DCC) and ElasticCC also use the N-Chance forwarding mechanism [7] to reduce off-chip misses. This mechanism reallocates evicted cache blocks in neighboring caches when the block is the last on-chip copy. Forwarding (also known as spilling) is handled by the DCE since it stores sharers information and also is able to avoid coherence races during the transmission. Spilling can be repeated N times for a given block. Since replication control is already enforced by the coherence protocol, in this paper N is set to 1 like in [3] and [11]. Therefore, if the spilled block is evicted again, it is evicted from the chip.

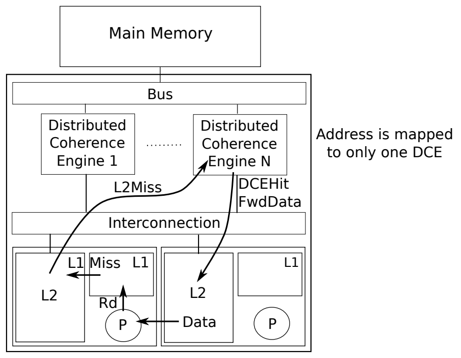
Main Memory
Bus
Distributed Coherence Engine 1
....
Distributed Coherence Engine N
Address is mapped to only one DCE
L2Miss
DCEHit FwdData
Interconnection
L1
Miss
L1
L2
Rd
P
L2
Data
L1
P

Figure 3: DCE working example.

4.2 ElasticCC Structure

The ElasticCC framework consists of several independent L2 cache memories that are logically divided into a shared and a private region that compete for the cache space. Private regions store all the evicted blocks from the local L1 and

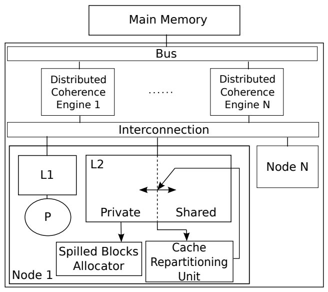

Figure 4: ElasticCC Node Structure.

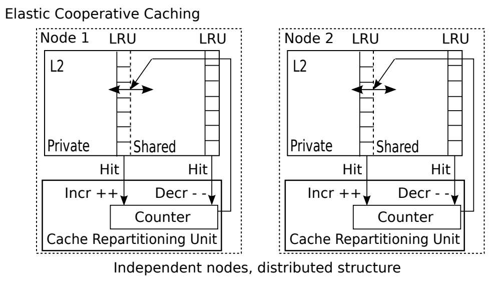
Elastic Cooperative Caching
Node 1
LRU
LRU
L2
Private
Shared
Hit
Hit
Incr ++
Decr - -
Counter
Cache Repartitioning Unit
Node 2
LRU
LRU
L2
Private
Shared
Hit
Hit
Incr ++
Decr - -
Counter
Cache Repartitioning Unit
Independent nodes, distributed structure

Figure 5: ElasticCC Repartitioning Unit.

shared regions store spilled blocks from neighboring caches. This allows the creation of big local private caches if all applications have similar cache requirements and a big shared cache if only a few take advantage of extra cache space. ElasticCC also adjusts the level of replication by repartitioning caches. Shared data is replicated when requested in the corresponding private regions but is never replicated in the shared region since it stores only unique blocks. Therefore, bigger private regions allow a higher replication and bigger shared regions limit it. To achieve the separation between private and shared cache space we have used the column caching technique proposed by Chiou [5] that allows separation and dynamic repartitioning without having to invalidate any block.

The extra hardware required for each node is a Repartitioning Unit and a Spilled Block Allocator, described in detail in the next subsections. The Cache Repartitioning Unit is responsible for dynamically adjusting the amount of cache that is going to be private or devoted to spilled blocks. Since not all nodes have the same shared cache space, the Spilled Block Allocator is responsible for deciding to which node a locally evicted block is spilled. This part is also important because more blocks should be spilled to the nodes with more shared space and less to the ones highly used by the local node. Figure 4 shows the structure of the proposed configuration.

4.3 Cache Repartitioning Unit

Our Cache Repartitioning Unit, depicted in Figure 5, adjusts the proportion of private and shared space locally for every L2 cache, avoiding centralized structures that limit the scalability. As can be seen, the hardware overhead of the repartitioning unit is minimal. The Repartitioning Unit only needs one counter per node that is incremented with Private LRU block hits and decremented with Shared LRU block hits.

Our cache repartitioning is done every fixed number of cycles and the decision is based on the number of hits on the Least Recently Used (LRU) blocks of the shared and the private parts of the cache. If the resulting value of the counter exceeds an Upper Threshold (UT) then the size of the private region is increased. If the value does not reach a Lower Threshold (LT) then the size of the shared region is increased. For all the middle values the size is not changed to avoid oscillatory states. Repartitioning is done at a given number of cycles to match program phases behavior and thresholds are control registers set at boot time.

```
If Private_LRU_Hit then
   Increase Counter
EndIf
If Shared_LRU_Hit then
   Decrease Counter
EndIf
If Repartition_Cycle then
   If Counter > Upper_Threshold then
      Add_Private_Way
      Send Repartition_Info_Msg
   ElseIf Counter < Lower_Threshold then
      Add_Shared_Way
      Send Repartition_Info_Msg
   EndIf
   Clear Counter
EndIf
```

Figure 6: Cache Repartitioning Algorithm.

Figure 6 shows the cache repartitioning algorithm, where hits in the LRU blocks of the private region increase a counter and hits in the LRU blocks of the shared region decrease it. Because we use LRU hits to decide to change the cache size, we can increase the private size of the cache even when the working set already fits in it. However, this is only going to happen when the shared region is not used since in the other cases the LRU hits in the shared region will compensate the counter value.

Repartitioning does not require the eviction of all cache blocks of the reassigned way since this would degrade performance unnecessarily. Once a cache is repartitioned partition sizes are updated and this information is used by the replacement mechanism. Therefore, new blocks will gradually replace the ones belonging to the other partition. Private and Shared portions must always have at least 1 way to simplify the coherence protocol and avoid race conditions. Therefore, partitioning only affects the replacement and allocation mechanisms. For accesses, the L2 cache can be seen as a normal shared cache, directly accessed by the local processor or indirectly accessed by other nodes through the DCEs. This organization also means that in-flight coherence messages of blocks being shifted among regions are not going to be affected.

4.4 Spilled Block Allocator

The second important part of the Elastic Cooperative Caching is the distribution of spilled blocks across the chip. Due to the dynamic behavior of caches some nodes can be mostly private and without cache space for spilled blocks and other nodes can be completely shared. Therefore it is important to have a Spilled Block Allocator in each node to distribute spilled blocks efficiently.

Type	Working Set size	Sharing	Local Reuse	Private Cache size	Spilling
Saturating Utility	Small	H/L	H/L	Small	No
Low Utility	Big	Low	Low	Small	No
Shared High Utility	Big	High	H/L	Small	Yes
Private High Utility	Big	Low	High	Big	Yes

Table 2: Application Types Behavior

Every time that a cache is repartitioned, a message is broadcast to all nodes with the partitioning information. This information is later used by the Spilled Block Allocator to distribute data among caches in a more efficient way: sending more evicted blocks to caches with more shared space. Since this mechanism is only used to balance the amount of spilled blocks among nodes, and the maximum size variation on each repartitioning is going to be a single way, we allow the use of stale information when caches are repartitioned. Thus, it is possible to separate the repartitioning information from the critical paths and broadcast it in a low priority network channel.

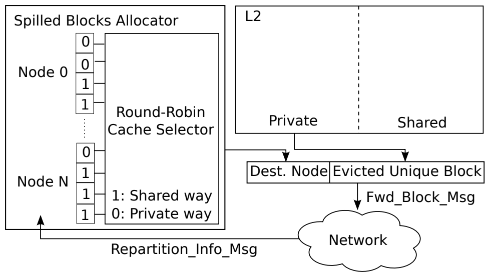

Figure 7: Spilled Block Allocator.

The proposed Spilled Block Allocator depicted in Figure 7 uses a Round-Robin arbiter with a bit vector containing the cache partitioning information of each way of each cache. The bit vector is updated every time that a message with partitioning information is received, with a bit representing the shared or private state of each way. When a block is evicted from the local private partition the arbiter selects the next shared way and spills the block to the corresponding node. Therefore, all shared ways are used equally and in circular order and nodes with more shared space receive more spilled blocks.

4.5 Adaptive Spilling mechanism

Elastic Cooperative Caching, as shown in previous sections, distributes available cache space to reduce the number of off-chip misses. Spilling, however, is done regardless of application properties. This may cause interferences between applications in the shared cache space. We propose an extension to the Elastic Cooperative Caching that takes advantage of its fully distributed organization and decides locally (i.e. autonomously) whether spilling is needed or not depending on the type of application.

Table 2 shows the most important characteristics of each of the application categories defined in section 2 and the last two columns show the desired behavior of our system. Low Utility applications, for example, minimize private regions due to their low reuse but still are able to use the shared region. Since this type of applications may have a very high number of misses, the shared cache is going to be filled by blocks that are not going to be reused. The desired behavior in this case would be to forbid spilling. It can be seen that only High Utility applications benefit from the shared cache space. Therefore, our Adaptive Spilling mechanism detects high utility applications and allows them to use the shared cache space while making it unavailable to others.

We will allow spilling, then, for both Private and Shared High Utility applications, but our mechanisms detect each of these applications differently. Private High Utility applications are detected through block reuse. Since applications with high reuse will have a high number of private ways, spilling is allowed when 75% of the cache (6 ways) is private. For Shared High Utility applications, sharing is detected by monitoring cache-to-cache transfers. The spilling decision is done on a per-block basis, allowing us to spill only the truly shared blocks. To track the sharing history of data, one bit is added to each entry of the DCE and is set when there is a cache-to-cache transfer (1= Block was shared, 0= Block was never shared). This bit is later used to detect the corresponding utility type together with the size of the private and shared regions. Our studies have shown that if a block was shared only once it is worth while to spill that block. This information is stored in the DCEs because it provides a global view of the sharing history of the block and avoids the influence of variations in the L2 cache size. The hardware overhead of this improvement is 64 Bytes of extra memory per DCE, which is negligible.

5. EXPERIMENTAL SETUP

We have evaluated our proposed framework with Simics [18], a full-system execution-driven simulator extended with the GEMS [20] toolset that provides a detailed memory hierarchy model. We have added a power model to the simulator based on Orion [27] and Cacti [26] to evaluate the energy efficiency of our proposal. Our configuration uses simple cores with small primary caches to improve the aggregate thread throughput by a high number of processors [8]. Table 3 shows the values for the most important configuration parameters.

Previously selected benchmarks of the SPECOMP2001 workload set have been simulated with the reference input sets and in pairs, each of them executing 8 threads.

The Elastic Cooperative Caching framework has been compared against traditional organizations such as shared or private last level cache. As we have seen, existing cache partitioning techniques rely on a centralized cache or a centralized repartitioning unit with information from all caches that works very well for a small number of nodes. However, for a high number of cores, the scalability of these techniques is very limited. For instance, ASP-NUCA has a max number of blocks in a set for the private and the shared partition. This means that for replacements in the shared region, all

Parameter	Value
Number Processors	16
Instr Window/ROB	16/48 entries
Branch Predictor	YAGS
Technology	70 nm
Frequency	4 GHz
Voltage	1.1 V
Block size	64 bytes
L1 I/D Cache	16 KB, 4-way
L2 Cache	256 KB, 8-way
DCE Size	8192 entries
Network Type	Mesh with 2 VNC
Hop Latency	3 cycles
Link BW	16 bytes/cycle
Memory Bus Latency	250 cycles

Table 3: Configuration Parameters

shared partitions are considered as a unique cache. Therefore a distributed version of this technique would require snooping all nodes on every cache replacement, potentially saturating the network. Therefore we consider it more fair to compare ElasticCC to scalable state-of-the-art cache organizations like the Adaptive Selective Replication [2] and the Distributed Cooperative Caching [11].

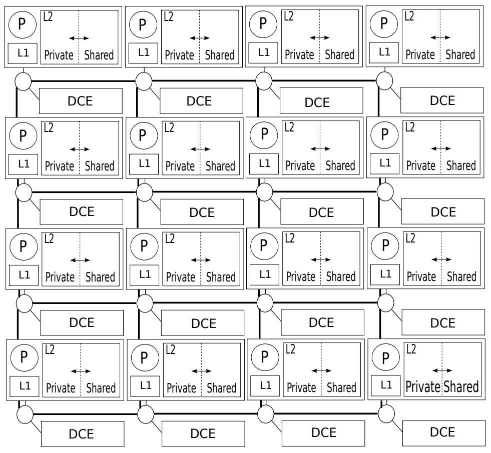

Figure 8: ElasticCC Memory Structure.

Since our work is intended for large tiled microarchitectures we have used Distributed Coherence Engines (DCEs) [11] to grant coherence in the ASR technique and in the ElasticCC to avoid centralized directories or snoop based protocols. To the best of the authors knowledge, this is the first time a realistic distributed version of the ASR technique is implemented and evaluated. Network interconnect also plays an important role in the overall performance [15]. Therefore, we have used a mesh interconnect to have a scalable solution. Figure 8 shows the memory structure of ElasticCC with one DCE per node. In all the tested configurations two levels of cache are used, as well as a MOESI protocol to grant coherence between nodes. All simulations use a local and private L1 cache and a shared/private L2 cache for every processor. Evaluated configurations are:

Shared Memory. This configuration assumes a Non-Uniform Cache Access (NUCA) architecture. L2 cache is physically distributed across the nodes and logically unified. Addresses are mapped to cache banks in an interleaved way to try to distribute requests in the network. L1 and L2 caches are inclusive and the L2 also includes the directory information for the allocated entries. On an L1 miss, the L2 bank corresponding to the address is accessed. If the block is located in another L1 in read-only mode, then it is replicated in the requesting node L1. Otherwise, the owner is invalidated without having to access the off-chip directory. This configuration tries to optimize cache usage and reduce off-chip accesses.

Private Memory. In this design, an L2 cache bank is assigned to every processor. On an L2 cache miss, memory must be accessed to check if the block is shared and to retrieve the data. This configuration makes little usage of the on-chip network and tries to optimize the access latency by placing all cache blocks in the local L2.

Distributed Cooperative Caching. (DCC) [11] This configuration has been used as a baseline, therefore, results are normalized over it. With this configuration it is possible to see the influence of interferences produced by the usage of a common cache space for private and spilled blocks. It uses 1 DCE for each node/processor with 2 R/W ports and 8-way associativity.

Adaptive Selective Replication. (ASR) [2] We have evaluated a distributed version of the ASR technique where coherence is granted through DCEs. This configuration uses 8 K entry 8 way NLHBs and 512 entry 8 way VTBs for each node.

Elastic Cooperative Caching. (ElasticCC) We have evaluated our dynamic proposal with a Cache Repartitioning Unit updated every 100k cycles and with a high threshold of 5 and a low threshold of 0. Thresholds are determined empirically. Since there is no previous information available on the behavior of applications, caches are initialized with 4 private and 4 shared ways.

ElasticCC + Adaptive Spilling. (ElasticCC+AS) This configuration evaluates the Elastic Cooperative Caching extended with the Adaptive Spilling mechanism, which only allows spilling when the private region occupies 75% of the cache (6 ways) or when the evicted block was shared. Thresholds for the Repartitioning Unit are the same as the previous configuration.

Ideal. Finally, a configuration that shows how much performance is possible to extract by increasing the cache size is evaluated. This configuration doubles the cache space in each node by using half private half shared 16 way 512kb L2 caches. In this case the shared/private division is static and grants 8 ways to each region. This situation emulates a configuration where there is -at the same moment- the maximum private and maximum shared space (i.e. twice the capacity of the evaluated cache).

6. RESULTS

6.1 Performance and Energy Efficiency

In this section, the Elastic Cooperative Caching is evaluated and compared to existing memory hierarchy configurations. Figure 9(a) shows the performance of the studied configurations normalized to Distributed Cooperative Caching, the power/performance relation that measures the energy efficiency of the proposed solutions, and the number of off-chip misses per instruction.

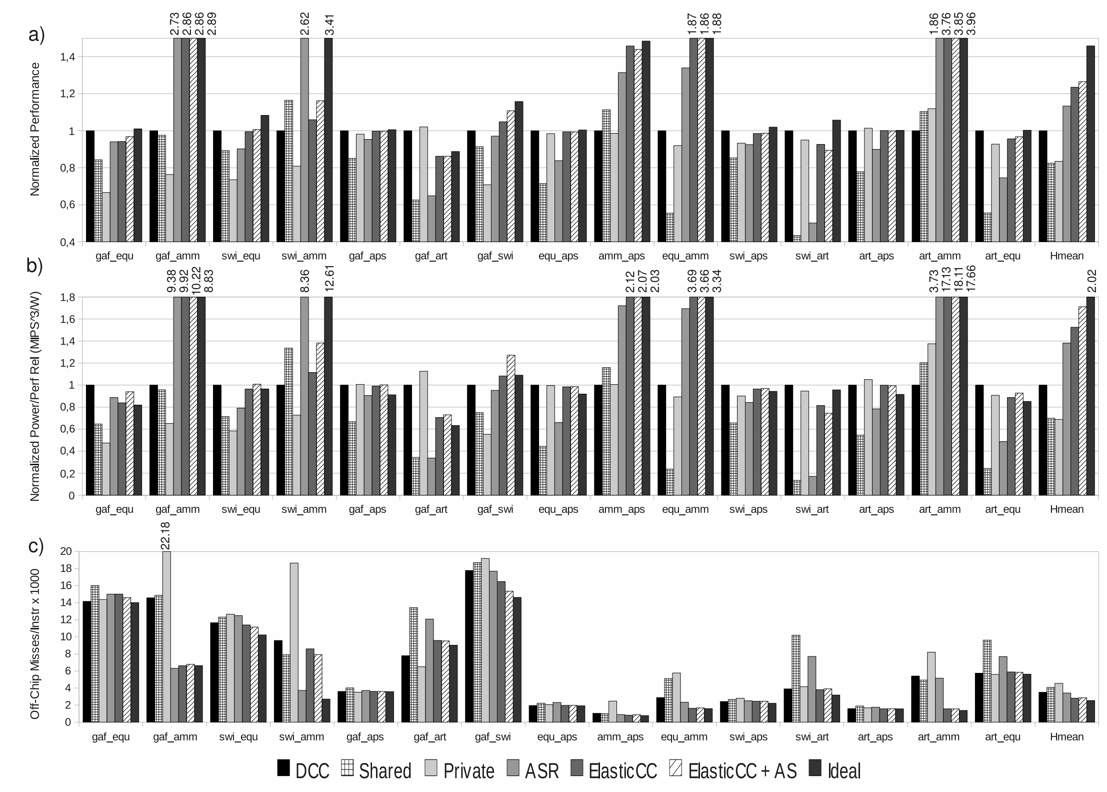

Figure 9: Normalized performance, Normalized energy efficiency and Off-Chip misses per Instr.

The performance graph shows that the Elastic Cooperative Caching outperforms the Distributed Cooperative Caching by an average of 27%, by 12% over the distributed version of Adaptive Selective Replication, by 52% over private caches, and by 53% over a distributed shared cache. Performance improvement in dynamic configurations is highly dependent on the characteristics of all the applications being executed simultaneously. Performance improvements can only come from High Utility benchmarks and in the other cases the adaptive mechanism must find the lowest amount of dedicated resources that does not degrade performance.

It can be seen that the Elastic Cooperative Caching is able to improve the performance of High Utility benchmarks when they are able to benefit from the extra cache space of neighboring applications. In some cases, it achieves almost the same performance as the Ideal configuration that has twice the cache space. The only configuration where performance is not similar to the ideal configuration is the one executing two High Utility benchmarks (Swim-Ammp), where ASR gets better performance. In this case both benchmarks benefit from the larger cache space and none of them leaves unused cache space. ElasticCC, however, outperforms ASR in all other cases thanks to its ability to repartition caches.

From the High Utility applications, Ammp is the one that improves its performance in a more significant way. This application is known for sharing a lock variable that leads to increased invalidations and cache misses[1]. Elastic Cooperative Caching is able to keep this data in the caches by avoiding replication of shared blocks and this leads to a high reduction in the number of off-chip cache misses. On the other hand, Swim requires much more cache space to improve its performance. Therefore, in this case, performance improvements depend much more on the other application that is executing simultaneously. However, when executing with a Low Utility benchmark (Gafort-Swim), it is able to increase performance by 10%.

The energy efficiency of Elastic Cooperative Caching is showed in Figure 9(b). Results in MIPS^3/W are normalized over the DCC configuration. ElasticCC+AS shows a 71% improvement over DCC and 24% over ASR. The more effective usage of shared regions with Adaptive Spilling can be seen in this graph. ElasticCC+AS improves the energy efficiency by 12% over ElasticCC without Adaptive Spilling. This improvement is produced by more effective spilling that reduces the network traffic and the avoids unnecessary reallocations of blocks without reuse.

Finally, Figure 9(c) shows the number of off-chip misses per instruction for each configuration. As expected the more efficient use of caches of ElasticCC brings an average reduction of 18.6% over DCC and 16.4% over ASR.

6.2 Dynamic behavior of ElasticCC

To see how the Elastic Cooperative Caching adapts to application behavior Figure 10 shows the average partitioning of the caches through the execution of the applications. Results show that the Repartitioning Unit is able to detect the application behavior and adapt cache sizes accordingly. The Low Utility application (Gafort) is granted in all cases less than 4 ways. The number of private ways assigned depends on the neighbor application. Therefore, when the second application is a Private High Utility one (Gafort-Swim), private ways are even reduced to less than 2. On the other

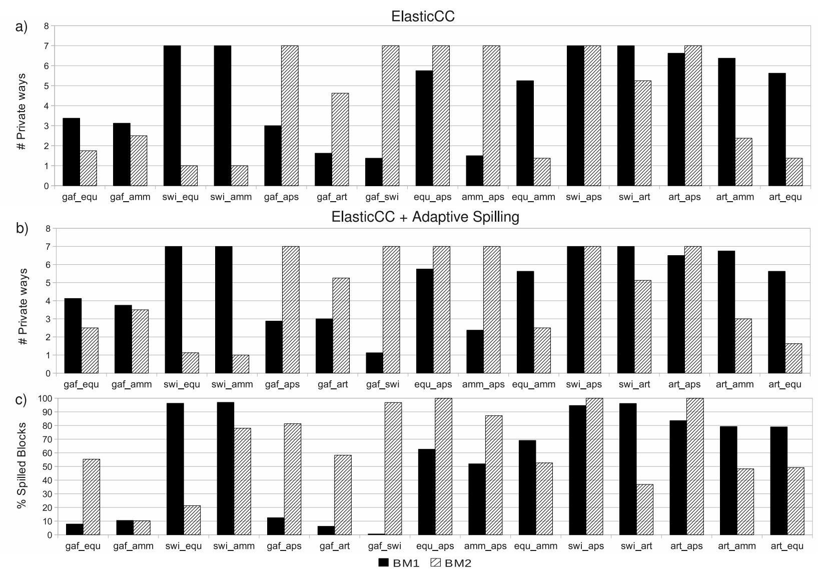

Figure 10: Average number of private ways per benchmark in ElasticCC and ElasticCC + AS and percentage of spilled blocks per benchmark in ElasticCC + AS compared to ElasticCC.

hand, the Private High Utility application (Swim) always uses all the available cache space and ends with 7 private ways. Finally, in the case of the Shared High Utility application (Ammp), it is possible to see that the amount of private cache is reduced to reduce replication and keep a bigger number of different cache blocks on-chip.

Although cache repartitioning is done correctly, the usage of the shared space is not optimized since Low Utility applications are going to corrupt it. Therefore a configuration with Adaptive Spilling also has been tested. Figure 10(b) shows the average final state for this configuration and also the percentage of evicted blocks that have been spilled to the shared cache. Caches are adapted to application behavior as well as without Adaptive Spilling but in this case private regions are slightly larger. This increase is compensated by the much more restrictive usage of shared regions. It is interesting to observe how the Adaptive Spilling is able to filter data that is not going to be reused. Figure 10(c) shows the percentage of spilled blocks. Spilling for the Low Utility application (Gafort) is significantly cut down in all cases compared with the ElasticCC alone where all evicted blocks are spilled. This reduction increases the available cache space for other applications.

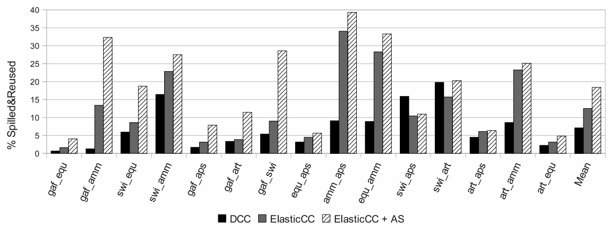

Figure 11: Percentage of spilled blocks that are reused.

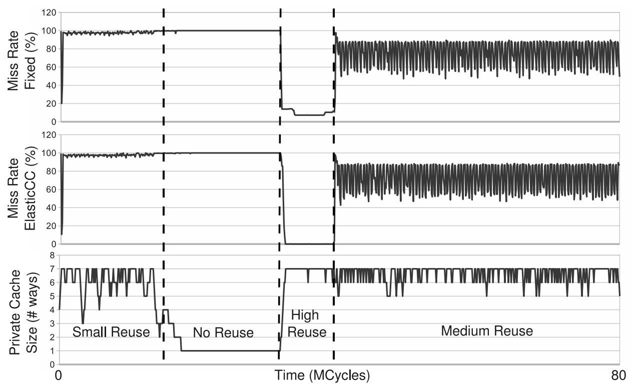

Figure 12: Cache behavior for thread 1 of Equake.

The effectiveness of Adaptive Spilling can bee seen in Figure 11 where the percentage of spilled blocks that are reused is shown. Elastic Cooperative Caching clearly increases the efficiency of spilling compared to Distributed Cooperative Caching. Reuse is increased on average from 7.1% to 12.5%, avoiding unnecessary reallocations of blocks without reuse. Adaptive Spilling is able to further increase this reuse up to 18.4%. This translates to fewer unnecessary messages across the network and improves the energy efficiency of the proposed configuration.

Finally, Figure 12 shows the temporal behavior of the elastic Shared/private cache of the node executing thread 1 of Equake for the Gafort-Equake combination. The plot shows the miss rate of the private region for a fixed version with half private-half shared cache and for ElasticCC. When the local thread makes no reuse at all, the shared cache space is increased so other nodes can take advantage. On the other hand, when reuse is high, the cache behaves as private to reduce cache misses. An finally, when reuse is small or medium, the behavior depends on other threads requirements. Since Gafort is an application with low reuse, very few blocks are going to be reallocated in the shared space and the cache is kept mostly private. In addition, Figure 12 shows that when reuse is small the cache is more likely to repartition than when we have a medium reuse. It can be seen that ElasticCC is not only capable of detecting differences among applications but also variations within the same application, adjusting cache size accordingly.

7. CONCLUSIONS

A dynamic and scalable memory hierarchy is necessary for next generation tiled microarchitectures. Elastic Cooperative Caching is shown to be a good candidate due to its autonomy, distributed organization and adaptivity to applications. Its repartitioning unit is the first to allow independent resource allocation based on local information, making it suitable for tiles with a high number of cores. The proposed scheme is able to detect the different cache requirements among applications and distribute cache resources accordingly without software support. Furthermore, the sensitivity of the repartitioning unit is able to detect changes in the execution phase of an application and adapt the system to the new situation. ElasticCC achieves a speedup of 27% over Distributed Cooperative Caching, 12% over Adaptive Selective Replication, 52% over private caches, and 53% over a distributed shared cache. More effective cache allocation is responsible for this improvement since it leads to a 19% reduction of off-chip misses compared to DCC and 16% compared to ASR. Furthermore, the dynamic management of cache resources avoids the energy overhead of reallocating not-reused cache blocks and consequently increases energy efficiency by 71% over the DCC configuration and 24% over the ASR configuration.

8. ACKNOWLEDGEMENTS

This work has been supported by the Generalitat de Catalunya under grant 2009SGR1250, and the Spanish Ministry of Education and Science under grant TIN2007-61763. We would like to thank Dean Tullsen and Marc Lupon for their comments on this work.

9. REFERENCES

[1] V. Aslot and R. Eigenmann. Quantitative performance analysis of the spec ompm2001 benchmarks. *Scientific Programming*, 11(2):105–124, 2003.

[2] B. Beckmann, M. Marty, and D. Wood. Asr: Adaptive selective replication for cmp caches. *MICRO-39: 39th IEEE/ACM Int. Symp. on Microarchitecture*, Dec. 2006.

[3] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *ISCA '06: 33rd Int. Symp. on Computer Architecture*, pages 264–276, 2006.

[4] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS '07: 21st Int. Conf. on Supercomputing*, pages 242–252, 2007.

[5] D. Chiou. Extending the reach of microprocessors: Column and curious caching. *PhD Thesis, Massachusets Institute of Technology*, 1999.

[6] Z. Chishti, M. Powell, and T. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. *MICRO-36: 36th IEEE/ACM Int. Symp. on Microarchitecture*, pages 55–66, 3-5 Dec. 2003.

[7] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. *OSDI '94: 1st Conf. on Operating Systems Design and Impl.*, pages 267–280, Nov. 1994.

[8] J. Davis, J. Laudon, and K. Olukotun. Maximizing cmp throughput with mediocre cores. *PACT '05: 14th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 51–62, 17-21 Sept. 2005.

[9] H. Dybdahl and P. Stenstrom. An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. *HPCA '07: 13th Int. Symp. on High Performance Computer Architecture*, pages 2–12, 10-14 Feb. 2007.

[10] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive nuca: near-optimal block placement and replication in distributed caches. In *ISCA '09: 36th Int. Symp. on Computer architecture*, pages 184–195, New York, NY, USA, 2009. ACM.

[11] E. Herrero, J. Gonzalez, and R. Canal. Distributed cooperative caching. *PACT '08: 17th Int. Conf. on Parallel Architectures and Compilation Techniques*, 25-29 Oct. 2008.

[12] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. Keckler. A nuca substrate for flexible cmp cache sharing. In *ICS '05: 19th Int. Conf. on Supercomputing*, pages 31–40, 2005.

[13] R. Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *ICS '04: 18th Int. Conf. on Supercomputing*, pages 257–266, New York, NY, USA, 2004.

[14] N. D. E. Jerger, L.-S. Peh, and M. H. Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence. *MICRO-41: 41st IEEE/ACM Int. Symp. on Microarchitecture*, 8-12 Nov. 2008.

[15] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *ISCA '05: 32nd Int. Symp. on Computer Architecture*, pages 408–419, Washington, DC, USA, 2005. IEEE Computer Society.

[16] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA '08: 14th Int. Symp. on High Performance Computer Architecture*, pages 367–378, Feb. 2008.

[17] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for cmps. In *HPCA '04: 10th Int. Symp. on High Performance Computer Architecture*, page 176, Washington, DC, USA, 2004. IEEE Computer Society.

[18] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[19] M. Martin, M. Hill, and D. Wood. Token coherence: decoupling performance and correctness. *ISCA '03: 30th Int. Symp. on Computer Architecture*, pages 182–193, 9-11 June 2003.

[20] M. Martin, D. J. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[21] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07: 34th Int. Symp. on Computer architecture*, pages 57–68, New York, NY, USA, 2007. ACM.

[22] M. Qureshi. Adaptive spill-receive for robust high-performance caching in cmps. *HPCA '09: 15th Int. Symp. on High Performance Computer Architecture*, 14-18 Feb. 2009.

[23] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. *MICRO-39: 39th IEEE/ACM Int. Symp. on Microarchitecture*, pages 423–432, Dec. 2006.

[24] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. In *ASPLOS XIII: 13th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 135–144, New York, NY, USA, 2008. ACM.

[25] K. Strauss, X. Shen, and J. Torrellas. Uncorq: Unconstrained snoop request delivery in embedded-ring multiprocessors. *MICRO-40: 40th IEEE/ACM Int. Symp. on Microarchitecture*, Dec. 2007.

[26] D. Tarjan, S. Thoziyoor, and N. Jouppi. Cacti 4.0. Technical report, HP Labs Palo Alto, June 2006.

[27] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik. Orion: a power-performance simulator for interconnection networks. *MICRO-35: 35th IEEE/ACM Int. Symp. on Microarchitecture*, pages 294–305, 2002.

[28] M. Zhang and K. Asanovic. Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors. *ISCA '05: 32nd Int. Symp. on Computer Architecture*, pages 336–345, June 2005.