2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

Introducing a Data Sliding Mechanism for Cooperative Caching in Manycore Architectures

Safae Dahmani*,†, Loïc Cudennec* and Guy Gogniat†
* *CEA, LIST, Embedded Real Time Systems Laboratory*
Gif-sur-Yvette, France
firstname.name@cea.fr
† *Lab-STICC Laboratory*
University of Bretagne Sud
Lorient, France
guy.gogniat@univ-ubs.fr

***Abstract*—In future micro-architectures, the increase of the number of cores and wire network complexity is leading to several performance degradation. These platforms are intended to process large amount of data. One of the biggest challenges for systems scalability is actually the memory wall: the memory latency is hardly increasing compared to technology expectations. Recent works explore potential software and hardware solutions mainly based on different caching schemes for addressing off-chip access issues.**

In this paper, we propose a new cooperative caching method improving the cache miss rate for manycore micro- architectures. The work is motivated by some limitations of recent adaptive cooperative caching proposals. Elastic Cooperative caching (ECC), is a dynamic memory partitioning mechanism that allows sharing cache across cooperative nodes according to the application behavior. However, it is mainly limited with cache eviction rate in case of highly stressed neighborhood. Another system, the adaptive Set-Granular Cooperative Caching (ASCC), is based on finer set-based mechanisms for a better adaptability. However, heavy localized cache loads are not efficiently managed. In such a context, we propose a cooperative caching strategy that consists in sliding data through closer neighbors. When a cache receives a storing request of a neighbor's private block, it spills the least recently used private data to a close neighbor. Thus, solicited saturated nodes slide local blocks to their respective neighbors to always provide free cache space. We also propose a new Priority-based Data Replacement policy to decide efficiently which blocks should be spilled, and a new mechanism to choose host destination called Best Neighbor selector.

The first analytic performance evaluation shows that the proposed cache management policies reduce by half the average global communication rate. As frequent accesses are focused in the neighboring zones, it efficiently improves on-Chip traffic.

Finally, our evaluation shows that cache miss rate is enhanced: each tile keeps the most frequently accessed data 1-Hop close to it, instead of ejecting them Off-Chip. Proposed techniques notably reduce the cache miss rate in case of high solicitation of the cooperative zone, as it is shown in the performed experiments.

***Keywords*-Many-cores, Tiled Micro-architectures, Memory Hierarchy, Cooperative Caching, Cache Partitioning, Data Sliding, Priority-Based Replacement Policy, Best Neighbor Selector.**

I. Introduction

Nowadays, multi-core processors use is very prevalent, either in regular desktop end-products, high performance computing systems or even in embedded computing systems (smartphones, automotive industry). One of the trend that can be observed since the last five years is a massively grow of the number of cores embedded on a single chip [1]. These systems are expected to grow up in the same way for the next decades, leading to a generation of new massively parallel architectures called manycores. Today, manycores made of ten to a hundred of cores are already available [2], [3]. Upcoming chips for 2013 are expected to embed 256 cores [4], [5] and research projects target up to 4096 cores [6].

The manycore systems are intended to execute a set of workloads with different memory needs. Using on-chip data caching allows to decrease access latencies, and therefore improves application performances. Data that are the most often used are fetched into high speed access memory units, close to the processing core, such as L1 caches or shared L2 caches. This avoids expensive requests to the main memory. These mechanisms are transparent to the developer and to the application. The resulting cache hierarchy is one of the main issues that lead to several optimizations. As we move down in the cache hierarchy, the memory storage capacity grows up, as for the access latency. The way cache memories are managed directly shapes the number of cache misses, that is responsible of slow external memory accesses and general performance decrease.

One way to optimize memory management relies on cache partitioning, which refers to the sharing of low level caches (L2/L3) between several threads that run concurrently. There are different cache hierarchy organizations depending on both the number of cache levels and if each level is shared or private. For example, the Intel Tera-scale architecture [7] relies on a distributed first-level cache that leads to different tradeoffs in private or shared cache modes. The use of private caches leads to small access latencies, and allows

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.205

a better scalability, which is of major importance regarding the design of manycore architectures. The downside of this approach is that shared data are replicated in multiple tiles, making harder to deal with data consistency.

In the remainder of the paper, we focus on manycore architectures with flat memory hierarchy. These manycores are made of processing cores, each core hosting a single private cache: no on-chip memory is shared. It can be found, for example, in the Adapteva Epiphany IV [8] and the Intel MIC Knights Corner [9], and can be compared to a wireless ad-hoc network or an unstructured peer-to-peer system.

In such a context, the cooperative cache policy [10], [11] has been proposed to efficiently manage data over large-scale architectures with no shared memory. Cooperative caching consists in taking benefits of some unused memory blocks in the neighborhood's private caches. According to adopted cache organization strategy (ie: LRU), mostly used data are kept as long as possible in cooperative areas.

This virtually increases the size of the private cache in order to avoid off-chip evictions. Cooperative caching differs from distributed shared memory (DSM) in a sense it does not offer a global address space.

A particular class of high performance applications, once deployed on a manycore architecture, is able to locally saturate the chip with numerous reads and writes. This is particularly true regarding image processing applications that use convolution filters, streaming-based processing, or the deployment of multiple applications with on-chip locality constraints in order to minimize inter-process communications. In this scenario, private caches attached to core that run the application are heavily solicited. Therefore, data may have to be evicted out of the chip, even using a cooperative cache: the saturated neighborhood can not be of any help.

In order to deal with this situation, we propose a data sliding mechanism that offers some improvement over the regular cooperative cache system. This mechanism lets a core A use the private cache of its neighbor B, even if B is saturated. In order to host this data, core B chooses one of its own data to be stored on another neighbor C. The mechanism ensures that the data are not evicted off-chip, and always located close to the owner core for performance reasons.

Our contribution has been analytically evaluated using synthetic codes coming from industry-grade image processing applications and a trace-based simulator that shows the benefits of the sliding mechanism, by drastically decreasing the number of data eviction.

Our paper is organized as follows: Section II presents a survey of works related to cooperative caching strategies, from which we motivate our contribution. In Section III we describe the Data Sliding mechanism, and the two cache replacement and neighbor selection policies. A first performance evaluation is discussed in section IV, wherein we do a global analysis of the results compared to both the baseline cooperative mechanism and the adaptive Elastic Caching strategy described later. The last section is for conclusion and future perspectives.

II. Related work

A. Cooperative Caching

Memory issues induced by concurrent accesses, large piece of data and dataflow processing are increasing in conjunction with the number of integrated cores on a single die. Data caching issues in large scale systems are also considered in several other areas such as mobile networks and web servers. In these contexts, several contributions [12], [13], [14], [15] regarding cache level organizations have been proposed in order to enhance cache miss rate and access latency.

In a small history of microprocessors, the first architectures were based on fully private caches. Shared caches were proposed to reduce the main memory access rate [16]. In order to benefit from both designs, hybrid caching was proposed to support heterogeneous workloads [17]. This leads to different utility-based and power-aware optimizations through several cache split strategies [18], [19]. Hybrid cache organization improves the local hit rate, taking benefit from the fully private cache approach, and keeps the overall miss rate as within the shared cache hierarchy. With the growing number of processing elements in manycore architectures, it appears to be quite difficult, or even impossible, to provide physical shared memories among a large number of cores (although 3D stacking may have the leverage to design such systems in the future). This is why distributed caching algorithms are deployed over manycore architectures.

One of the major approach emerging in this context is based on private cache aggregation [10], [13], [20]. Cooperative caching has been proposed to enhance access latency and reduce cache miss rate. Under heavy load conditions, available cooperative caches provide unused sets of blocks to neighbors that are short of space [12], [14]. This way, data are kept close to the requesting nodes.

In spite of the increasing number of nodes, data caching capacity is more limited because of different physical and technological constraints. It is even more crucial to efficiently choose which data to remove and which data to keep close to the processing element. Typical cache management strategies for optimizing data eviction are called replacement policies. These policies play a significant role in reducing cache miss rate and data access time.

For example, energy aware platforms like mobile networking and storage file systems consider that communications are the major source of power consumption. Thus, a good replacement policy allows reducing general traffic by conserving cached data as long as possible. In order to ensure this, some replacement policies were previously proposed [15], [21]–[23]. They are classified following the

parameters used to take the replacement decision. A few examples of parameters are: access cost, access pattern, spatial and temporal dependencies. For instance, traditional Least Frequently Used policy assumes that most frequently accessed data will be the most probably called in next accesses.

The next generation of on-chip systems has to support a broad spectrum of applications with different memory requirements, while the on-chip storage capacity is lower and the cost of off-chip misses becomes more and more significant. Cooperative caching seems to be a relevant approach and the induced cache partitioning issues lead to important optimization tracks. Thus, it is necessary to have the best tradeoff for optimizing private and shared data caching according to different workloads.

B. Adaptive Cache Partitioning

Many static and dynamic approaches have been presented to improve cache resources allocation in cooperative zones. In order to avoid strong cache contention, most current works propose a time-based sharing partitioning system. It particularly considers multiple applications utilities to unfairly allocate cache resources [20].

One of the main goals is to set the frontier between private cache and shared cache by limiting underused space. Elastic Cooperative Caching (ECC) [24] has been proposed as an adaptive memory hierarchy that consists in creating a hybrid cache which dynamically re-adjusts local and shared zones according to cached data reuse in each side. The ECC provides an autonomous way to control data spilling in cooperative area to avoid contention and power consumption at the level of cache coherency unit. In addition, cache elasticity allows to get both advantages of private cache in terms of access latency and those of shared cache approach with cache miss rate reduction.

Another adaptive cache partitioning proposal is the Adaptive Set-Granular Cooperative Caching [25]. This proposal is also based on data migration from high utility caches to underused ones. The Set-Granular Cooperative Caching proposes spilling techniques that allow measuring the stress level of each set in a set-associative cache. Depending on measured stress degree of sets the system decides the spilled ones and those that will receive them.

In a situation of a global short of storage capacity, where all cooperative caches are full, none of the presented adaptive mechanisms can afford the ability to spread out cooperative zone, while keeping data 1-Hop close to the requesting cores.

The Data Sliding approach handles efficiently stressed cooperative neighborhood with data migration through neighbor's cooperative zones. In response to storing requests, each saturated node should push only one time local private blocks to his least stressed neighbor, so that it could store spilled data from most stressed requesting neighbors. Finally, local data migration is stopped when reaching a cache free cooperative area.

Our proposal is based on two main policies: Priority-based replacement policy and Best Neighbor Selector. These mechanisms allow each node to decide efficiently the block to be replaced and the best host cache (Best Neighbor) to receive it.

III. Contribution: Data Cooperative Sliding Mechanism for highly stressed zones

A. Proposed Data Sliding protocol

The manycore processors provide the ability of running several parallel applications in the same time. One of the biggest challenges in such highly parallel architectures is to adapt the memory resource allocation to different application workloads. Previous studies of adaptive cache partitioning [24] have classified the applications in four categories regarding the amount of shared data and working set; saturating utility, low utility, shared high utility and private high utility. In our paper,we focus on private high utility applications (e.g: Swim) with a high data reuse amount. These applications are characterized by important private cache requirements.

Different combinations of these kind of applications provide effective issues of managing concurrent accesses to cache hierarchy, especially in cooperative neighboring cache schemes. Cores with short cache solicitations benefit from the extension of storage space through shared zones. This creates highly stressed spots in cooperative zones. To be able to provide an adaptive cooperative policy, it is important that storage space can be distributed efficiently between cooperative nodes. In fact, in the case of highly stressed neighborhood, adaptive cooperative caching mechanisms can not manage data storage considering the memory needs of all the cooperative nodes.

The elastic cooperative cache hierarchy provides balanced cache partitioning between local cache space and shared unused space. However, the cyclic adaptive aspect of this mechanism depends on the running application behavior as well as the neighbor's one. Therefore, when both applications are private high utility, the partitioning unit will try to dynamically adjust the size of each zone. As cyclic cache partitioning decision is based on concurrent private and shared access hits, more blocks will be replaced in both zones and then evicted off-chip.

In this paper, we propose a new cache spilling mechanism which allows stressed neighborhoods to keep frequently accessed data on-chip, near the owning tiles. The sliding mechanism allows the migration of private blocks between neighbors in a highly stressed context, as seen in figure 1.a. As far as we know, this is not allowed in existing cache cooperative systems. When a tile sends a private block to a neighbor cache, this one replaces the least recently used private block by the new received one. In order to

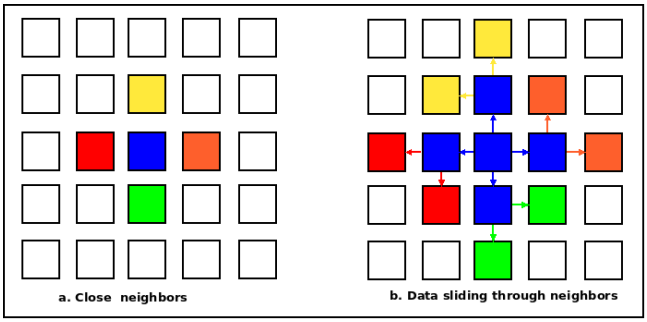

Figure 1. Data propagation through neighbors

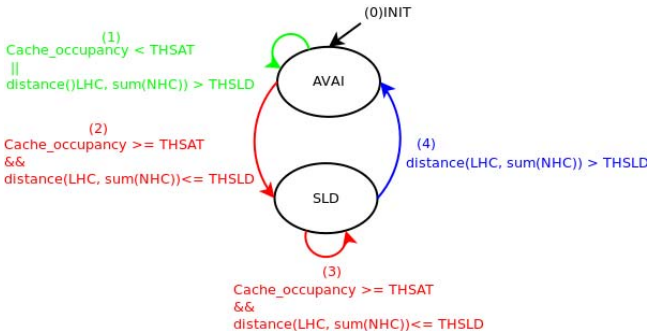

Figure 2. State machine of Data Sliding mechanism

avoid the eviction of the data, the host node sends the replaced private block to his nearest neighbor. Thereby, every neighbor receiving spilled data pushes local blocks to his cooperative neighboring zone in order to release cache space for incoming shared blocks. This process is recursively repeated until the sliding propagates to a non saturated area, or to what has been defined as the edge of the chip. In the latter case, private data is evicted out of the chip. However, this worst case may not occur in large manycore architectures, where temporary cool areas still exist.

To simplify our approach, cached blocks are 1-chance forwarded, which means that a block that has been moved to a neighbor can not be moved again (except to move back to the owner). This is not a limitation, and we can imagine switching to a N-chance forwarding system, as long as it remains more efficient than getting the data from external memory. Thus, each node keeps its private data one-hop close (figure 1.b), instead of migrating them to the opposite side of the chip or even evicting them off-chip.

B. Functional Description of Data Sliding Mechanism

According to our approach, the cache functional state is either in $AVAILABLE$ or $SLIDING$ mode. The $SLIDING$ mode is activated when the whole neighborhood is highly stressed. When a cache is saturated (ie: cache occupancy is higher than $THSAT$), data sliding is not automatically activated. Only highly competitive accesses to saturated caches allows data neighbor-to-neighbor sliding.

In order to get information about both core and neighborhood workloads, we define two main types of counters, that are managed on each core:

- The Local Hit Counter (LHC) is incremented according to local accesses,
- The Neighbor Hit Counter (NHC) is associated to each neighbor and is incremented according to each shared data accesses from the neighbor.

Comparing neighboring access to local access amounts lets the system evaluate the memory needs of both sides. A tile is defined as stressed when the distance between LHC and $\sum NHC$ is lower than a given threshold $THSLD$. Then, the system switches on $SLIDING$ mode. When the previous condition is no more satisfied it returns to $AVAILABLE$ mode and all counters are reset (see figure 2).

For performance reasons, a dedicated free block is chosen on each core in order to satisfy every sliding request without waiting for the end of the propagation. The least recently used private block is thereafter chosen to be slided onto another neighbor and the resulting free block is locked to satisfy the next sliding request.

The figure 3 presents different steps of cache replacement policy with sliding blocks through 3 neighbors. Three neighbors $N1$, $N2$ and $N3$, are represented along with their private caches (fig. 3.1) . A data $@a$ is stored on the $N1$ floating free block (fig. 3.2) . The least frequently used data $@b$ on $N1$ is thereafter elected to be slided on neighbor $N2$ (fig. 3.3). The resulting block is being the new floating free block (fig. 3.4) . Finally, the sliding process stops for neighbor $N3$ offers enough free space.

C. Data Replacement Policy

Once a core receives a sliding request from one of its neighbors, a block of its own has to be slided. This block is chosen in regard of the data replacement policy. Replacement policy presented in Elastic Cooperative Caching strategy [24] doesn't consider memory needs of cooperative neighbors. It only relies on current partitioning scheme and replaces data in the appropriate area. In highly solicited neighborhood, this strategy can lead to under-used cache space in an area whereas the other one is in lack of storage space. It is mostly because the partitioning scheme doesn't reflect memory needs of the whole cooperative neighbors. We therefore propose to get rid of the cache partitioning scheme, while introducing labels to the data that indicate if it is private or shared. This way, the proposed replacement policy can either choose to replace private or shared blocks according to data access frequency of both the core and its cooperative neighborhood.

According to counters values, the private or shared last recently used data is replaced as following:

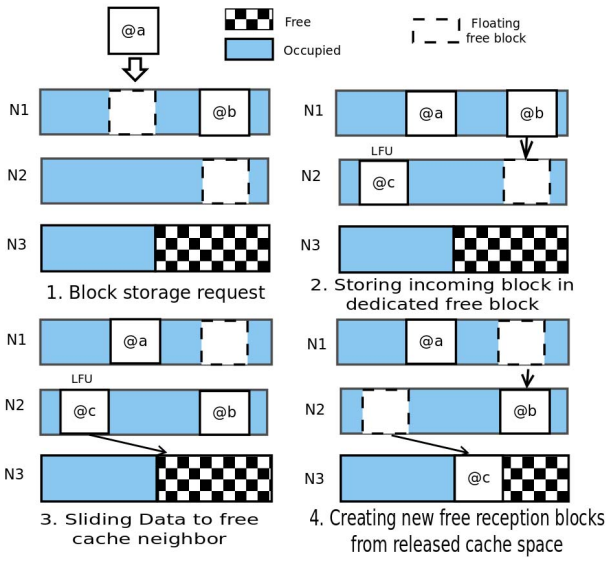

Figure 3. Data Replacement Strategy

1) $LHC > \sum NHC$**:** The local core needs are greater than the overall neighbor needs. Therefore, the shared LRU is replaced,
2) $LHC < \sum NHC$**:** The overall neighbor needs are greater than the local core. Therefore, the private LRU is replaced,
3) $LHC \simeq \sum NHC$**:** Both sides are running under high mutual solicitation. This condition is raised when the distance between LHC and $\sum NHC$ is smaller than a given threshold. In this case, we choose to replace the private LRU. Our sliding protocol is activated and the local data is migrated to a neighbor. If it appears that this neighbor is also stressed, it replaces its local block with the new one and sends the old block to its own neighboring. Thereby, hot spots are managed with less off-chip evictions.

Through this counter-based priority replacement policy, frequently accessed blocks are maintained as long as possible close to their owner nodes. In a highly stressed context, shared blocks have always the priority. Only private data are evicted to neighbors. Sliding data are propagated across the chip until reaching a free cooperative cache. Therefor, global cache workload is shared evenly between stressed and free zones through the chip.

D. Best Neighbor Selector

The second important mechanism in this data spilling strategy is to efficiently choose the destination neighbor. We propose a neighbor selection policy based on the NHC counters introduced in section III-C. Unlike a simple Round Robin strategy, used in previous works, that evenly balances data over the neighborhood, we propose to select the neighbor which has a smaller hit counter. The NHC information is therefore used to choose the neighbor with the less workload. In a highly stressed neighborhood, a large number of cache blocks may move to neighbors, taking advantage of the highly connected network-on-chip mesh.

The *best neighbor* definition depends on the associated NHC counter. This counter somehow mirrors the neighbor cache needs. Indeed, stressed neighbors send more requests to their cooperative zone, mechanically increasing their associated NHC. Blocks are sent to the least stressed nodes, which leads to a non-uniform distribution of requests. In this approach, available nodes are the most solicited in order to avoid blocks eviction. Thus, it enhances cooperation, avoiding hot spots by considering host node availability in blocks distribution.

Furthermore, the best neighbor selection is based on status information collected in a *passive* manner. The NHC counters are updated thanks to each neighbor request, not using an active polling of the neighbors. Active polling (i.e. heartbeat) would give more accurate real-time information, but would also increase the number of control messages handled by the network on chip, decreasing the application performances. This approach, close to the piggybacking practice, gives to each core a local view of the chip status, which is something very familiar to large-scale distributed systems such as peer-to-peer and wireless ad-hoc networks, where a global view would be practically impossible to build.

In manycore architectures, some of the cache coherence features can be handled and accelerated by dedicated hardware. Our cache mechanisms require a single Cooperative Caching Unit (CCU). This unit is in charge of 1) labeling every stored block in the cache and 2) enforcing both data replacement and neighbor selection policies. If we compare to the Elastic Caching system, our solution reduces the metadata traffic since both blocks eviction and allocation are only based on local counter comparisons, without any complex synchronization or distributed consensus between cores.

IV. Performance Evaluation

A. Test Procedure Description

In order to evaluate the benefits of the sliding mechanism, data replacement and neighbor selection policies, we have made some preliminary experiments that compare the behavior of the contribution with two cache coherency protocols: the Baseline protocol [26] and the Elastic Cooperative Caching protocol (as presented in section II).

The Baseline protocol is a regular protocol widely used in multi-core architectures. One of the most famous flavor is the 4-state home-based MESI protocol (which stands for modified, exclusive, shared and invalid). In a home-based protocol, each piece of data is under the responsibility of a dedicated core, the home-node, in charge of granting access and managing the data consistency state. Data addresses are

usually mapped to home-nodes using a round-robin or a hash function. Each time a core requests to read or write a data, it has to contact the associated home-node to get access to an up to date version of the data. This procedure is costly and most likely means that there are concurrent accesses on the data between cores, or that the data is not cached on the chip and has to be fetched from external memory. The Baseline protocol does not implement any cooperative strategy between caches. Both Elastic Cooperative Caching and Sliding protocols can be built upon the Baseline protocol, as a transparent extension that virtually enlarges the private local cache on each node.

The goal of these experiments is to show that there is a real benefit to 1) use cooperative strategies by comparing the Baseline protocol with the ECC protocol and 2) use a sliding mechanism to keep data on-chip by comparing the ECC protocol with our contribution.

All the experiments are based on an analytical simulation that calculates, for each physical link of the network on chip, the number of messages induced by the application accesses. This experimental protocol is based on an analytical approach and does not take into account the timing of events. We used several synthetic applications, as well as some image processing pieces of code taken from an industry-grade application. We ran these codes under the supervision of the PinTools [27] instrumentation software. Pintools allow to analyze the application behavior at the instruction level. In our experimental process we used a modified version of the pinatrace [28] plugin to generate a specific cache access trace with an additional core id field.

The format of the output file contains six information fields per access:

1) Instruction address
2) Access type: read or write
3) Mapped data address
4) Data size
5) Prefetched instruction indicator
6) Core id

Here is an example of a cache trace generated by the modified Pinatrace tool:

```
0xb5d79c80: W 0x8057284 4 0 3
0xb5d79c60: W 0x8057274 4 0 3
0xb5d78cb7: R 0x8057280 4 0 1
0xb5d78ce9: R 0x8057274 4 0 1
```

For each access, we simulate the behavior of the underlying cache coherency protocol and we deduce, for each physical link of the network on chip, the type and the number of messages that are transferred. We consider a 4x4 core array connected thanks to a mesh topology wherein every tile is connected to 4 neighbors. This NoC topology is used in some current manycore architecture such as Tilera Gx (6 overlapped meshes of 4-port routers). Requests are routed through a point to point communication mode. In this context, the distance between two cores is determined by the Manhattan distance. Different types of messages are distinguished depending on the request subject and destination. In order to compare the cache protocols by highlighting the cooperative ability, we focus on two groups of requests:

- **Messages to Home Node:** all messages whose destination is a Home Node (Read, Write, Invalidation, Downgrade)
- **Messages to Neighbor:** requests sent to neighbors to read and write shared data.

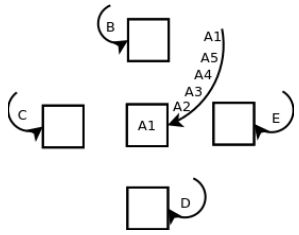

Figure 4. Access pattern for the global traffic evaluation

The cost of each request depends on the number of nodes that messages go through before reaching the destination. Hence, a 1-hop communication leads to a lower access cost in term of latency and power energy than requests to remote nodes across the chip. Requests to access data in a neighbor shared cache are always 1-hop far, whereas reaching the home-node and fetching the data from another node or from external memory usually take a greater number of hops. In order to evaluate the protocols, we used the following performance metrics:

- **Network load:** message distribution across the chip and average point to point message count,
- **Hop counts:** the number of hops between source (requester) and destination or Home Node.

In the remainder of this section, we present three main experiments evaluating the global traffic on the chip, the neighbor selection policy and the replacement policy.

B. First Performance Evaluation Results

1) Global traffic evaluation: In this first test, we analyze the traffic induced by an important workload and a highly stressed neighborhood. In such a context, nodes mutually request additional memory storage support from neighbor caches. To induce this behavior, we used a synthetic application that alternates accesses between the central node and its four neighbors, as shown in figure 4.

Figures 5 and 7 respectively show the number of messages triggered by the Baseline and the Elastic protocols. The X and Y axis represent a flat view of the chip, made of a 4x4 core array. The Z axis represent the number of messages that goes through these coordinates.

The upper yellow shapes indicate the maximum number of messages observed on the chip, and the downer blue shapes

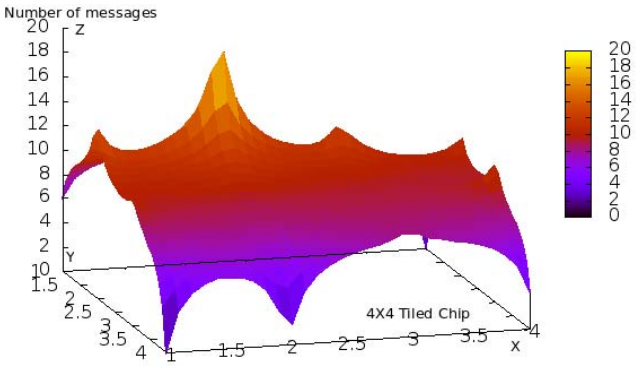

Figure 5. Traffic with Baseline Protocol

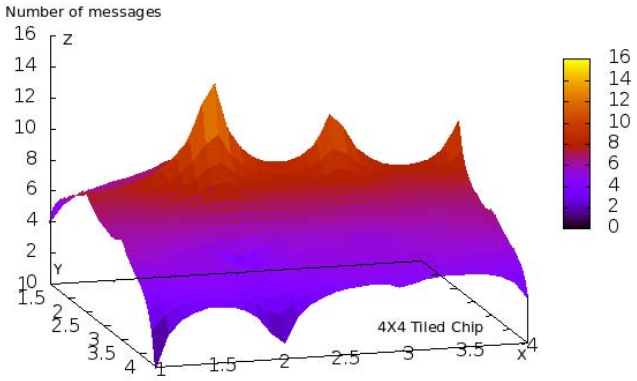

Figure 6. Traffic with Sliding Data Caching

indicate that there were no communications. In both Baseline and ECC figures there is a yellow peak corresponding to the highly stressed core and a surrounding medium red shape corresponding to the stressed neighborhood.

Using cooperative caching has two major advantages. First, it slightly reduces the overall number of messages by keeping more data on-chip. Second, traffic peak outskirts are cooler using the cooperative protocol than the baseline one. This is due to the massive reduction of home-node requests that cross the entire chip in the case of the baseline protocol. Only the immediate surroundings remain hot, due to the cooperative caching activity.

Afterwards, we compare the Elastic Cooperative protocol with our Sliding mechanism (see figure 6). The number of exchanged messages is approximately the same for both the Elastic Caching mechanism and the Sliding Data Caching. However, we notice that the traffic area is reduced with the proposed Sliding mechanism. Communications are basically concentrated in the near neighborhood.

Remote requests to Home Nodes reflect a high cache miss rate. Thanks to cooperative caching, traffic area is remarkably reduced. This is due to the reduction of the average communications with Home Nodes which means

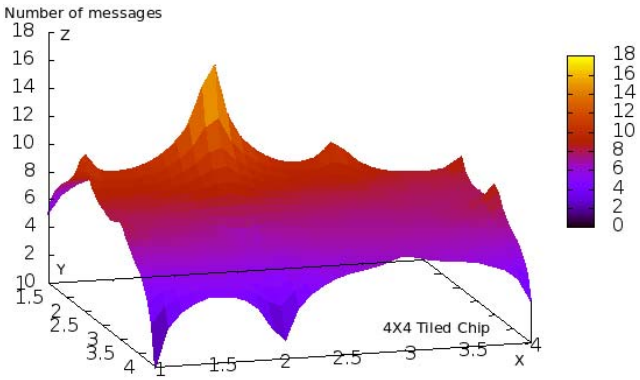

Figure 7. Traffic with Elastic Caching

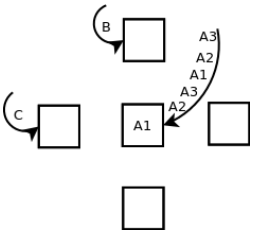

Figure 8. Access pattern for the best neighbor selection policy

that there are less cache misses.

The Sliding Caching mechanism allows every cooperative node to push local private data to its neighboring perimeter for storing shared blocks. On one side, less blocks have to be evicted off-chip which reduces cache miss rate and then requests to Home nodes. On the other side, each node maintains frequently accessed data at only 1-Hop close. This approach promotes neighbor-to-neighbor communications, which explain traffic concentration on the cooperative zone.

Remote requests to Home Nodes are multi-Hop, point-to-point communications, where closer requests (1-Hop communication) enhance access cost to data and reduce the global network load through the chip.

2) Best Neighbor strategy evaluation: In this experimentation, we evaluate the best neighbor selection policy for cooperative caching compared to the regular round-robin policy. As for the previous experimentation, we set a highly stressed central node. We also saturate the North and West neighbors while the South and East ones remain free (see figure 8 for this scenario). In both protocols, the central node workload triggers cooperative process with its neighboring.

In the Elastic approach, the data repartition is based on a Round-Robin algorithm. This allows to equally distribute data caching in a circular order on the neighboring. The best neighbor strategy defines the destination node according to the memory load of each neighbor, as reflected by the local neighbor hit counters (NHC).

Figure 9 and 10 respectively show the traffic standing

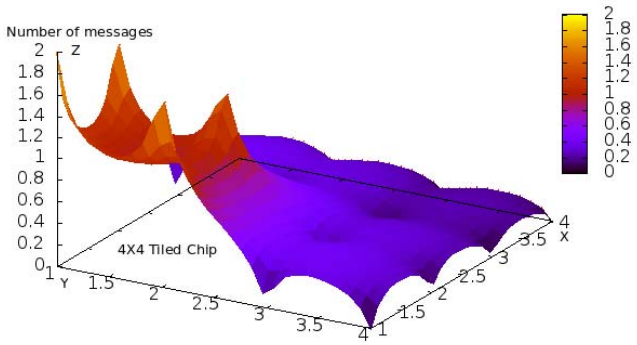

Figure 9. Traffic to neighbors using the Round Robin policy

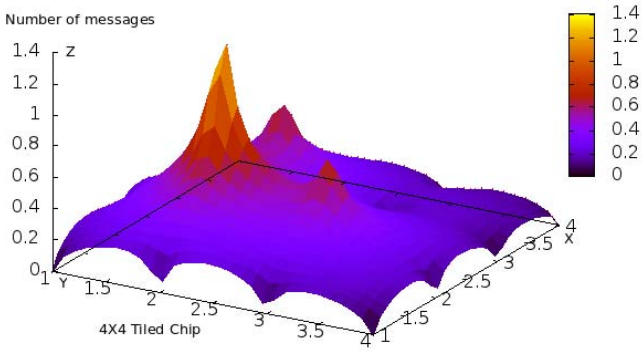

Figure 10. Traffic to neighbors using the Best Neighbor policy

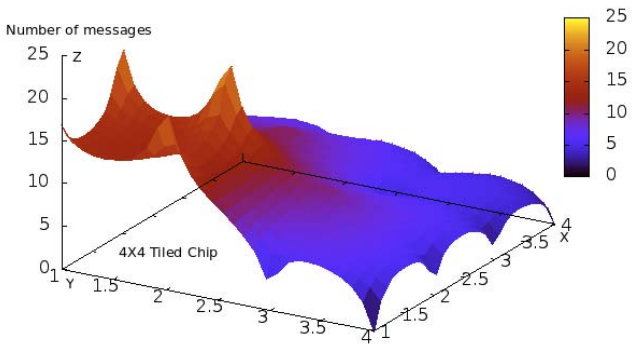

Figure 11. Traffic using the cyclic elastic partitioning policy

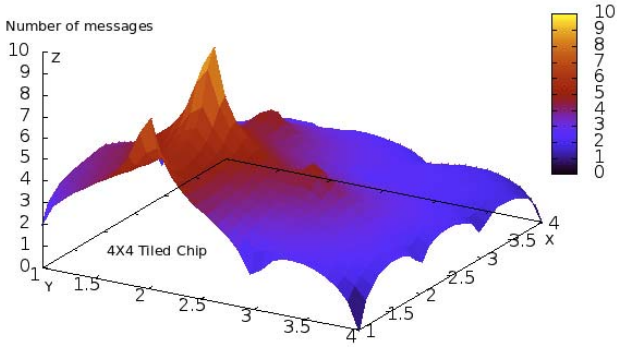

Figure 12. Traffic using the priority replacement policy

between neighbors in order to remotely access to data in a shared cache.

As the Best Neighbor policy considers the memory needs of the selected destination node, evicted blocks from the central stressed neighbor are stored in free cache neighbors. Whereas, the Round-Robin policy penalizes private data of North and West nodes, which increases the cache miss rate and consequently the Home-Node accesses. North and West nodes replace their private data with shared blocks from central node.

3) Priority Based Replacement Policy: The third performance test evaluates the cache partitioning policy based on the shared and private blocks priority. The experiment consists in saturating all neighboring caches, and multiplying redundant data accesses across the neighborhood. This scenario generates alternated accesses to shared and private areas of cooperative caches.

The Elastic Cooperative approach updates its cache partitioning every N cycles, which doesn't satisfy the application needs in presence of a very heavy data flow. In fact, concurrent accesses to private and shared zones generate a strong oscillation between both areas. Our proposed cache replacement policy is not time dependent, but rather *event-driven*. At every storing request, the local and neighbors counters are compared. The decision about the block to be replaced is based on the access frequency to both private and shared blocks.

Figures 11 and 12 respectively show the traffic induced by both regular cyclic elastic partitioning policy and the priority replacement policy. Using the priority replacement strategy, the maximum number of messages is divided by two, drastically reducing the hot spot.

The decrease of requests to Home Nodes is due to the low data eviction rate. So, the average miss rate is lower when using priority based data replacement.

As a conclusion we can observe that proposed techniques based on comparing locally managed counters allow to reduce data off-chip evictions based on the consideration of both private and shared data access rates. Obviously, priority is given to frequently accessed blocks. Data with less priority is migrated to neighboring caches in case of highly stressed context and evicted off-chip otherwise.

Both Best Neighbor policy for destination nodes and priority replacement strategy for evicted blocks grant a robust behavior in the framework of the adaptive sliding mechanism, in the context of highly stressed neighborhood.

V. Conclusion

The Data Sliding strategy described in this paper, is a new approach of Cooperative Caching intended to handle highly stressed neighborhood. Our proposal is based on two main mechanisms:

1) **Neighborhood Counters:** This mechanism allows each core to passively monitor its neighbor stress rate and is used for cache resource partitioning. Data to be replaced in the cache, as well as the remote host core are selected through a comparison between local and shared hit counters. The use of different counters is an efficient way to estimate the memory requirements of each cooperative node in order to adapt the sharing cache resource to different loads.
2) **Allowing 1-Hop private data migration:** Sliding Data strategy provides a balanced data repartitioning across the chip when cooperative area is highly solicited. It leads to fairly handle competitive cache accesses between local and shared blocks. Our contribution proposes to promote the received data, and push the local blocks to the neighbor's caches. We show that it is an efficient way to reduce the overall cache miss rate by keeping frequently accessed data on the chip and replacing off chip evictions with neighboring storage.

As for now, the data stored in a neighbor cache can not be accessed by this neighbor. This is not only designed by simplicity: letting a remote node access another private data *bypasses* the underlying cache coherence protocol and breaks the consistency model. However, some particular data access patterns involving those two neighbors may benefit from such a clearance. As a future work, we plan to study the interactions between the cooperative sliding protocol and the main cache consistency protocol in order to allow the local use of neighbor's data, while preserving a consistent state.

Experiments presented in this paper are based on an event-driven trace. They do not take into account any time metrics, such as access latency and data access rate. These metrics would give information about the absorption capacities of access requests. That is why we plan to use a micro-architecture simulation environment such as [29] for evaluating time performances of our proposed strategy. We also aim to use a scalable cache coherent NUMA architecture such as [30] that allows us to implement our proposal in a highly parallel platform.

Furthermore, the evaluations are based on a synthetic access memory trace that only reflects the behavior of some application sub-routines. In our future analysis, we plan to consider heterogeneous parallel applications that generate multiple hot spots on the chip. For this, we plan to use high utility applications, such as the distributed version of the simulated annealing metaheuristic algorithm [31] that recursively leads to multiple hot spots on the chip while exploring the space of solutions.

Acknowledgment

We would like to thank Jean-Thomas Acquaviva and Stephane Louise from CEA LIST for the help they provided with the cache validation simulator, and the modified pinatrace tool during the validation phase.

References

[1] "International technology roadmap for semiconductors, 2011 edition, system drivers." International Technology Working Group, 2011. http://www.itrs.net.

[2] A. Agarwal, L. Bao, J. Brown, B. Edwards, M. Mattina, C.-C. Miao, C. Ramey, and D. Wentzlaff, "Tile processor: Embedded multicore for networking and multimedia," in *Proceedings of the 19th Hot Chips Symposium*, HC 2007, (Stanford, California, USA), August 2007.

[3] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, and R. Van Der Wijngaart, "A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling," *Solid-State Circuits, IEEE Journal of*, vol. 46, pp. 173 –183, jan. 2011.

[4] "The kalray mppa 256 manycore processor." Kalray S.A. http://www.kalray.eu/.

[5] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pp. 983 –987, march 2012.

[6] "The tera-scale architecture project (tsar)." Bull and LIP6. https://www-soc.lip6.fr/trac/tsar.

[7] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar, "An 80-tile sub-100-w teraflops processor in 65-nm cmos," *Solid-State Circuits, IEEE Journal of*, vol. 43, pp. 29 –41, jan. 2008.

[8] adapteva Inc., "A 1024-core 70 gflop/w floating point manycore microprocessor," in *Proceedings of the 15th Annual Workshop on High Performance Embedded Computing*, HPEC 2011, (Lexington, Massachusetts, USA), September 2011.

[9] G. Chrysos, "Intel xeon phi coprocessor (codename knights corner)," in *Proceedings of the 24th Hot Chips Symposium*, HC 2012, (Stanford, California, USA), August 2012.

[10] J. Chang and G. Sohi, "Cooperative caching for chip multiprocessors," in *Computer Architecture, 2006. ISCA'06. 33rd International Symposium on*, pp. 264–276, IEEE, 2006.

[11] J. Chang and G. Sohi, "Cooperative cache partitioning for chip multiprocessors," in *Proceedings of the 21st annual international conference on Supercomputing*, pp. 242–252, ACM, 2007.

[12] Y. Ting and Y. Chang, "A novel cooperative caching scheme for wireless ad hoc networks: Groupcaching," in *Networking, Architecture, and Storage, 2007. NAS 2007. International Conference on*, pp. 62–68, IEEE, 2007.

[13] V. Holmedahl, B. Smith, and T. Yang, "Cooperative caching of dynamic content on a distributed web server," in *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pp. 243–250, IEEE, 1998.

[14] J. Cho, S. Oh, J. Kim, H. Lee, and J. Lee, "Neighbor caching in multi-hop wireless ad hoc networks," *Communications Letters, IEEE*, vol. 7, no. 11, pp. 525–527, 2003.

[15] L. Yin and G. Cao, "Supporting cooperative caching in ad hoc networks," *Mobile Computing, IEEE Transactions on*, vol. 5, no. 1, pp. 77–89, 2006.

[16] K. Jackson and K. Langston, "Ibm s/390 storage hierarchy—g5 and g6 performance considerations," *IBM Journal of Research and Development*, vol. 43, no. 5.6, pp. 847–854, 1999.

[17] L. Zhao, R. Iyer, M. Upton, and D. Newell, "Towards hybrid last level caches for chip-multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 2, pp. 56–63, 2008.

[18] M. Qureshi and Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 423–432, IEEE Computer Society, 2006.

[19] C. Liu, A. Sivasubramaniam, and M. Kandemir, "Organizing the last line of defense before hitting the memory wall for cmps," in *Software, IEE Proceedings-*, pp. 176–185, IEEE, 2004.

[20] H. Dybdahl and P. Stenstrom, "An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 2–12, IEEE, 2007.

[21] P. T. Joy and K. P. Jacob, "Cache replacement policies for cooperative caching in mobile ad hoc networks," *CoRR*, vol. abs/1208.3295, 2012.

[22] L. Shi, Z. Liu, and L. Xu, "Bwcc: A fs-cache based cooperative caching system for network storage system," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pp. 546–550, IEEE, 2012.

[23] V. Anagnostopoulou, S. Biswas, H. Saadeldeen, A. Savage, R. Bianchini, T. Yang, D. Franklin, and F. Chong, "Barely alive memory servers: Keeping data active in a low-power state," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 8, no. 4, p. 31, 2012.

[24] E. Herrero, J. González, and R. Canal, "Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 419–428, 2010.

[25] D. Rolan, B. Fraguela, and R. Doallo, "Adaptive set-granular cooperative caching," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pp. 1–12, IEEE, 2012.

[26] D. Culler, J. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st ed., 1998. The Morgan Kaufmann Series in Computer Architecture and Design.

[27] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Notices*, vol. 40, pp. 190–200, ACM, 2005.

[28] J. Marandola, S. Louise, L. Cudennec, J.-T. Acquaviva, and D. Bader, "Enhancing Cache Coherent Architectures with Access Patterns for Embedded Manycore Systems," in *International Symposium on System-on-Chip 2012 (SoC 2012)*, (Tampere, Finlande), Tampere University of Technology, Department of Computer Systems, IEEE, Oct. 2012.

[29] N. Ventroux, A. Guerre, T. Sassolas, L. Moutaoukil, G. Blanc, C. Bechara, and R. David, "Sesam: An mpsoc simulation environment for dynamic application processing," in *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, (Washington, DC, USA), pp. 1880–1886, IEEE Computer Society, 2010.

[30] A. Greiner, "Tsar: a scalable, shared memory, many-cores architecture with global cache coherence," in *9th International Forum on Embedded MPSoC and Multicore (MPSoC'09)*, 2009.

[31] F. Galea and R. Sirdey, "A parallel simulated annealing approach for the mapping of large process networks.," in *IPDPS Workshops*, pp. 1787–1792, IEEE Computer Society, 2012.

[32] P. T. I. (ws and J. Kulick, "Multiprocessing on a chip," 2008.

[33] G. Almaless and F. Wajsburt, "Does shared-memory, highly multi-threaded, single-application scale on many-cores?,"

[34] G. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *The Journal of Supercomputing*, vol. 28, no. 1, pp. 7–26, 2004.

[35] M. Qureshi, A. Jaleel, Y. Patt, S. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 381–391, ACM, 2007.