Using the Spring Physical Model to Extend a Cooperative Caching Protocol for Many-Core Processors

Safae Dahmani, Loïc Cudennec, Stéphane Louise
CEA LIST
Gif-sur-Yvette, France
firstname.name@cea.fr

Guy Gogniat
University of Bretagne Sud
Lorient, France
guy.gogniat@univ-ubs.fr

***Abstract*—As the number of embedded cores grows up, the off-chip memory wall becomes an overwhelming bottleneck. As a consequence, it is more and more prevalent to efficiently exploit on-chip data storage. In a previous work, we proposed a data sliding mechanism that allows to store data onto our closest neighborhood, even under heavy stress loads. However, each cache block is allowed to migrate only one time to a neighbor's cache (e.g. 1-Chance Forwarding). In this paper, we propose an extension of our mechanism in order to expand the cooperative caching area. Our work is based on an adaptive physical model, where each cache block is considered as a mass connected to a spring. This technique constrains data migration according to the spring constant and the difference of workloads between cores. This adaptive data sliding approach leads to a balanced spread of data on the chip and therefore improves on-chip storage. On-chip data access has been evaluated using an analytical approach. Results show that the extended data sliding increases the global cache hit rate on the chip, especially in the context of juxtaposed hot spots.**

***Keywords*-Many-cores, Cooperative Caching, Data Sliding, Mass-Spring Physical Model**

I. Introduction

Many-cores are processors made of hundreds to thousands cores on a single chip interconnected by a dedicated network. They offer high computing performance - provided the applications take benefit from massively parallel architectures - while drastically reducing the power consumption. They are expected to enter both green HPC and autonomous embedded systems. Today popular examples include Kalray MPPA (256 cores) [1], Tilera Gx (72 cores) [2], Adapteva Epiphany (64 cores) [3], [4] and Intel Xeon Phi (61 cores) [5]. One big challenge towards efficient programmability of many-cores is the ability to properly feed the computing threads with data. As in large parallel and distributed systems (e.g. computing clusters, grids and clouds), data management is of major importance when it comes to performances.

In this context, the memory wall [6] refers to the fact that the speed of a processor is limited by its data accesses, and no longer by the processing units. This is particularly true for manycores, as a large number of cores solicits memory, even at a low clock frequency. Several manycore architectures have been proposed to address the memory problem. They differ in two main aspects: the data cache organization and the network topology (mesh, torus, hypercube..).

While it is quite common to offer a small high speed L1 cache for both data and instructions attached to each core, the other cache levels are part of the inner chip specificity and the associated programmability choices. Each level makes the chip bigger in terms of silicon surface, power consumption and design complexity. For example, some L2 caches can be attached to each core (and aggregated within a virtual shared L3 cache, as in the Tilera Gx processor), physically shared within a cluster of cores (as for the Kalray MPPA), or even removed from the design (as for the Adapteva Epiphany). For the sake of simplicity, we consider in this paper a chip with a flat organization of cores and memory: each core has a private place to store data and is able to communicate with N neighbors, as well as the external memory.

Some class of applications (e.g. adaptive signal processing, computational dataflow,..) can generate hard-to-predict hot-spots on the chip. These hot-spots may be surrounded by a cooler neighborhood that can help locally by sharing some cache slots, in a cooperative behavior. Here, we assume that accessing a data stored by a neighbor is more efficient than fetching it from outside the chip (a dedicated low-latency NoC, versus some external data controller such as a DMA). Cooperative caching (CC) has been used in several contexts, and one implementation has been applied to many-cores in the Elastic Cooperative Caching system (ECC) [7]. This system allocates a shared zone in each private cache that can be used to help the neighborhood.

In a previous work [8], we proposed an extension of the ECC system, called the data sliding mechanism. It allows a core to store a data onto a neighbor, even if this neighbor is stressed by a hot-spot. In order to help, this neighbor will in turn move one of its own data onto a different neighbor. The sliding mechanism stops whenever it reaches a cooler area, or the edge of the chip. However, in order to keep the data close to the owner, the migration can not exceed one hop (1-chance forwarding). This contribution has shown to significantly reduce data eviction.

In this paper, we propose to evaluate the possibility to

expand the cooperative caching area (N-chance forwarding). We think that in large many-core architectures, it remains more efficient to access on-chip data, even stored in a few hops, than fetching from the external memory. The NoC should provide small latencies and an aggregated bandwidth, according to its topology. Data are allowed to migrate among the entire chip following some rules. These rules are inspired by the spring physical model, in which the data is an object with a mass (the priority), moving around the cores following their difference of potentials (the cache use rates), and attached to the owner thanks to a spring (in order to stay close to the core).

Starting from the theoretical formula, we propose a lightweight distributed algorithm that calculates the next move if a data needs to be slided. This algorithm has been implemented within an analytical cache simulator that calculates statistics on the network activity, based on a genuine memory access trace. These results show a significant reduction of the cache miss rate in configurations involving close hot-spots. As a counterpart, it also shows an increased traffic activity within the hot-spot neighborhood.

II. Related work

The challenge of efficiently using the on-chip cache space is becoming particularly critical in high performance systems. Cache requirements of new emerging high-end applications (Streaming, Imaging, Simulation) remarkably degrade its performance. Among the large literature on data caching, we can highlight these two problems:

- Cache pollution incurred by keeping in cache less frequently used data [9], [10] and
- Cache thrashing which is caused by cache access conflicts especially in the LLC (Last Level Cache) [11], [12].

Many-core systems are more affected by vainly off-chip data ejection. Indeed, cache thrashing induced by LLC contention, large workloads and dataflow processing becomes more complicated to be addressed as the number of cores increases. A number of cache management approaches were proposed for multi-core processors and distributed systems in the large. One popular approach is the Cooperative Caching (CC) mechanism that has been proposed for chip multiprocessor systems [13], [14], [15], several other areas such as mobile networks [16], [17], [18] or big data systems [19], [20]. The CC mechanism consists in aggregating a group of nodes in order to form a (virtual) unified super-cache space. In the context of heavy workloads, available cores provide underused cache space to help other stressed cores.

In a previous work [8], [21], we introduce the data sliding mechanism as a 1-chance forwarding protocol, allowing each core to store its own data in direct neighbors caches.

One of the shortcomings of the closed cooperative caching is that data cannot travel farther in the chip. The direct neighborhood is the only one that can receive evicted blocks in the presence of memory hot spots. Our main contribution, called the mass-spring cache model, is an extended cache cooperative approach that minimizes off-chip data ejection by expanding the sliding area.

In order to mitigate the cache performance degradation in highly stressed context, prior works discussed the possibility to extend data migration radius to N-Hop. Some proposals try to dynamically adjust the aggregate shared cache according to each core activity. This technique allows data spilling to the requesting core closer neighborhood. Several works aimed to optimize data spilling techniques, especially for choosing the right cache resource distribution and replacement policies. A number of works propose some power-aware spilling techniques for different systems, based on selective data migration approach [22], [23], [24]. Another set of proposals try to improve data spilling and the destination selection policies for the N-chance forwarding mechanism. Dominguez-Sal et al. proposed the high reused data aware strategy [25] that consists on spilling blocks to the destination core where it is going to be reused soon. For all the proposed techniques, the number of allowed hops per block is statically defined. The limitations of such an approach is reached when the stressed zone is important. If no cache space in the defined spilling area is available, cache blocks are evicted off-chip. Another reason why our physical model outperforms the prior ones is that migrated blocks are brought back near the owner once its neighborhood workload is down. Besides, the adaptive aspect of our approach takes into consideration the current load of each eventual host core.

Finally, the Evicted-Address filter proposed by Seshadri et al. [26] is a new technique that can predict the reuse behavior of missed cache blocks in order to avoid both pollution and thrashing. This could be used in conjunction with the mass-spring model to achieve a better sliding scheme.

III. Background: Data Cooperative Sliding Mechanism

One of the biggest challenges in highly parallel systems is to manage on-chip memory resources exposed to heterogeneous application workloads. In order to manage efficiently on-chip data storage, many static and dynamic techniques have been proposed, most of them aim to handle either cache partitioning or data migration issues.

The Elastic Cooperative Caching [7] has been presented as an adaptive memory hierarchy. This memory adjusts both local and shared areas according to the amount of data reuse that is available on each core.

Another work called the Adaptive Set-Granular Cooperative caching [13] proposed techniques that measure the stress level of each set in a set-associative cache. According to this, the protocol makes a decision on the set that is going to be spilled.

In case of low storage capacity, the presented adaptive mechanisms cannot afford data migration to spread out of the direct cooperative area. But instead, it keeps each block near its owner core. To overcome this, the data sliding mechanism [8] has been proposed. The main motivation of this prior work was to enhance both cache partitioning and data sliding techniques in order to effectively manage highly stressed neighborhoods.

The data sliding mechanism consists in 1-Hop Cooperative Caching. A saturated core is allowed to push the local blocks to its direct neighbors. In order to offer a cache space to host new blocks, saturated neighbors spill their local data to their neighborhood. This process is repeated until the propagation reaches a non-saturated area.

We propose in the current work, a data sliding mechanism with a dynamic sliding radius. This contribution is an intuitive extension of our original work. Thanks to different cache management policies, we expect to reduce the data off-chip ejection and to balance the memory workloads across the chip.

The heart of the proposition is not the possibility to move the data from neighbor to neighbor, but the way such a decision is made. We rely on a physical mass-spring model, which is, as far as we know, a unique contribution in the field of large multicore and many-core architectures.

IV. Mass-Spring Model for Cooperative Caching

One important aspect of N-chance forwarding is the migration policy: moving a data too far from its owner may become less efficient than getting it from the outside of the chip. Another consideration is that each move should select the closest, and most available neighbor in terms of cache saturation level. We think that these migration constraints are relevant in the context of the physical mass-spring model.

In our contribution, we propose to apply the mass-spring model to the data cache management. It assumes that each cache block is a mass attached to its owner node with a spring. As in the physical spring model, each block is constrained by the *spring constant* (K) which defines its degree of freedom. Thus, the sliding radius is defined depending on this constant. The block migration path is chosen according to the memory load of cores. This memory load acts as a *potential* on each core. When migrating a data, the stressed core compares his potential with its neighbors, and chooses the one with the lowest potential. Migrated blocks will then move between cores, avoiding high potentials. Figure 1 shows a $(3*4)$-core processor in which cores are represented by red bar plots. The height of the plots indicates the potential, that is equivalent to the access rate of the local cache. Selecting the lowest potential gives the neighbor that is much likely willing to help.

The elasticity of the spring model allows owner cores to pull back their remote blocks, whenever the workload is lower or the chip limits are reached. The physical spring

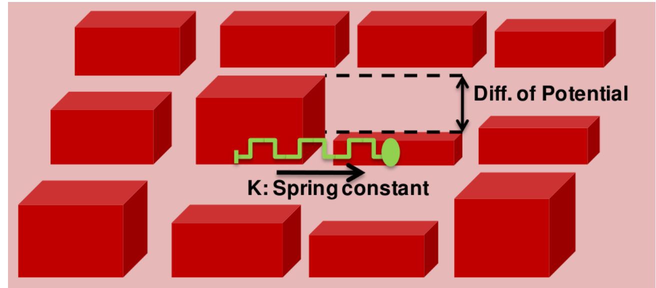


Figure 1: Applying the spring-mass physical model to a 12-core processor.

characteristics allow to adjust data migration parameters in an adaptive way. It takes into account the traveling distance of the block and the memory load of previously solicited cores. More details about implementation will be discussed later.

A. The physical model theory

1) Approach: Physical phenomena we can encounter everyday are often really good at decreasing local stress points by spreading the stress on the vicinity. They are good at finding an equilibrium that is good compromise on the energy point of view. But if we want a physical analogy, we also want to find a simple one that does not involves too much computation, that can be decided locally with little knowledge from far away points and that is not likely to lead to chaotic behaviors. Although the later point can be hard to prove with the local expression only, at least it would be easy to constrain the model to converge quickly to an equilibrium, at least locally.

If we think about the stress on a cache of a given core as an initial energy, then the most simple physical model would be inspired by the spring physics, since equations are simple and linear. To force a quick flow toward equilibrium and avoid resonant state for local configurations, the idea would be to use a fluid friction, because it is also linear, and choose critical coefficients in order to reach the local equilibrium as fast as possible.

2) Mathematical model: Noting Δh the stress on the cache, and doing the analogy with a height at which the spring would hang, the mass is pulled away from the center by gravity and because of the slope reaction. The effect can be projected on the x axis, so the reaction that pulls away the mass is proportional to Δh. The fluid constant would be c and the spring constant k, so:

$$m\frac{d^2x}{dt^2} = \alpha mg\Delta h - c\frac{dx}{dt} - kx$$

We can simplify equation (1) so that:

$$\frac{d^2x}{dt^2} + c'\frac{dx}{dt} + k'x = \alpha'\Delta h \quad (1)$$

Which is a nice linear equation of the second order. It becomes critical *iff* $c'^2 - 4k' = 0$. If we note $a = c'/2$, then:

$$(1) \Rightarrow \frac{d^2x}{dt^2} + 2a\frac{dx}{dt} + a^2x = b\Delta h$$

By using a discretization in time for the model where D_i defines the distance from its origin at time $t = i$ (discrete, $i \in \mathbb{N}$), we find:

$$D_i = 2AD_{i-1} - A^2D_{i-2} + B\Delta h \quad (2)$$

As can be seen, the value at next step only depends on the local value with a history length of 2 in the past, and math are simple products and additions. So this basic model has the required properties of simplicity and locality.

B. Implementation

The implementation of the mass-spring model has been made in a distributed context, keeping in mind that the algorithm should not introduce too much overhead to the global computation. This is why it only relies on a few local information already computed by the regular sliding mechanism, and the simple formula (2).

The main principle is to compute a cumulative distance, tracking each block movement. In the case of a saturated neighborhood, this distance is used to make a decision about the destination core. The corresponding algorithm, executed on each core when it comes to migrate a data, is described as follows:

1) For each of the N neighbors at step i, we compute the *difference of potential* Δh, as well as the *distance* D_i, using formula (2).

$$D_i = 2 * A * D_{i-1} - A^2 * D_{i-2} + B * \Delta h$$

Where D_{i-1} and D_{i-2} are the two previous calculated distances taken from the history of data migration. With constant A and B defined by

$$A = \frac{1}{K}$$

Where K is the spring constant. And

$$B = \frac{g}{K}$$

Where g is the weight constant.

2) We compare the N distances D_i together: the greatest one gives the natural direction the mass would have taken in the sandbox. We therefore move the block to the suitable core. If no neighbor is available for help, what can occur when all the neighbors are more stressed than the requester, we choose the nearest destination to the data owner core regarding the number of hops in the network.

The neighbor load information are locally stored, using counters called Neighbors Hit Counters (NHC) and described in the data sliding mechanism. These counters are incremented and decremented according to each neighbor request to a data stored in the shared part of the local cache. NHC are therefore continuously updated using a method close to piggybacking, and are available for use when needed, without taking time to ask the neighbors.

V. EXPERIMENTS

In this section, we first compare the mass-spring model to the regular data sliding mechanism that uses a 1-chance forwarding policy.

The second experimental phase compares the mass-spring protocol model with the two following naive models:

- *Random-Based protocol*: Data is distributed arbitrary through the chip. The destination core is chosen in a random way for each data migration. In this protocol, we do not neither consider the destination memory load nor the data migration distance.
- *Load Balancing protocol*: The second protocol consists in distributing data in a balanced way on the chip. The destination of the slided data corresponds to the current least loaded core. This technique aims to balance data distribution through the cores by keeping continuously a chip workload global overview that would not be relevant with large chips.

The aim of the experiments is to show that the mass-spring model improves the data availability and proximity in presence of several juxtaposed hot-spots. As a counterpart, the neighbor-to-neighbor communications are increased due to several neighbors accesses. We assume that off-chip memory accesses to satisfy a cache miss is much more costly in terms of latency and power consumption [27], than on-chip neighbor-to-neighbor requests.

A. ***Consistency protocol***

The mass-spring cache protocol is quite independent from the top level data consistency protocol: it introduces a distributed shared cache among the neighborhood. From the cache coherence point of view, this distributed cache is used the same way as a regular private local cache. Therefore, we use the same data consistency protocol and we measure the cache hit rate, as well as the protocol traffic, with both 1-chance forwarding data sliding and N-chance forwarding mass-spring model.

In the following experiments, we refer to the *baseline* data consistency protocol. This protocol is widely used in multicore processors. One of its popular implementation is the home-based, four-state MESI protocol, standing for *Modified*, *Exclusive*, *Shared* and *Invalid*. In a home-based protocol, each shared data (or piece of data) is paired with a core (the home-node), that is responsible of keeping trace of the data and its current state. The home-node can be paired using a round-robin distribution over memory addresses. Each time a core accesses a shared data, it has to contact its associated home-node to request clearance. This request to

home-node either means that there is a strong concurrency between cores onto this particular data, or that the data is not stored on the chip and has to be fetched from external memory. Therefore, we distinguish small messages that are exchanged between neighbors to retrieve the data, from messages to the home-node that will trigger a costly data transfer from external memory.

B. *Experimental setup*

In order to evaluate our contribution, we used an analytical simulator that gives a detailed view of the generated on-chip traffic. Our in-house simulator collects statistics on exchanged messages between all cores. It also allows to calculate the cache hit rate. The simulator reads a memory trace that contains a sequence of read and write accesses. The trace file is generated by running some synthetic or real applications under the Pinatrace instrumentation tool shipped with the Pin project [28]. We used a modified version of Pinatrace to generate the instructions set with an additional field giving the core id.

For each entry in the memory trace, the simulator calculates the number and the type of messages that are transferred on the network-on-chip. However, this calculation is not time-triggered, which means that except for the implicit *precedence* relation that is given by the memory access sequence, we do not take into account the possible contentions that can occur on a data or on a protocol role (such as the home-node). Therefore, we are not able to appreciate the real performance of the contribution onto the application. Nonetheless, this approach gives solid clues on the global behavior of the cache protocol, and what we can expect in terms of data transfers within the chip and with the external memory.

As for the network topology, we consider a 2-dimension mesh that interconnects $8 * 8$ cores. The network handles transactions through a point-to-point communication routing mode. We can find this configuration in recent platforms such as the Epiphany-IV 64-core microprocessor from Adapteva [3]. We assume each pair of cores is separated by a *Manhattan* distance that is used to calculate the number of hops.

To compare the 1-HOP data sliding with the mass-spring contribution, we use on-chip traffic metrics (Number of exchanged messages). In the baseline protocol, every cache miss generates a request to the home-node. In the following experimental scenarios, a cache miss systematically triggers a data transfer from the external memory. We also assume that requests to neighbors mean that the block is stored onto the chip.

The cost of each request depends on the number of hops to reach its destination core. According to the *Manhattan* distance, and because we consider large many-core architectures, an access to the neighborhood is statically less costly than an access to the home-node (if we consider a round-robin or a random distribution of home-nodes).

Another used metric is the cache hit rate. We define a *cache hit* as a successful access to the requested data, either it is in the local cache, or in a neighbor cache. Otherwise, we fall back into a *cache miss* that requires to contact the associated *home-node* and implies an external memory access cost.

Finally, we use for comparing the mass-spring model with the two naive mechanisms described above the manhattan distance between the requesting core and the accessed data. We then cumulate, for each core, all the distances in order to compare the proximity of data in different scenarios.

C. *Scenarios*

The used experimental scenarios are based on synthetic applications, that illustrate the behavior of the protocols in presence of juxtaposed hot-spots. All scenarios involve three roles that are mapped onto the chip. The first role frequently accesses a large set of data: 30 different memory addresses. The second role frequently accesses a smaller set of data: 4 different memory addresses that are not included in the first role set of addresses. The third role is the home-node, paired using a random distribution. Figures 2a and 2b illustrate the 3 scenarios mapped onto the $8*8$ chip. The first role appears in dark orange, the second role in soft orange and the home-nodes in blue. The basic idea is to generate hot-spots thanks to some close cores that play the first role. These hot-spots are surrounded by cores playing the second role. Some of these saturated cores can also act as home-nodes.

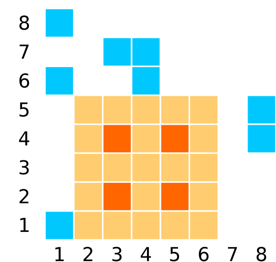
The cache size is set to a value between 4 and 30, which means that there is enough room for all addresses of the second role, but not for the first role. As the sequence of accesses is interpreted by the simulator, all caches in the orange area become full of data. While the second roles are keeping data in their local cache, first role hot-spots ask the neighborhood for help. In turn, this direct neighborhood ask for help according to the data sliding mechanism.

The results of the experiments are displayed as a processor heatmap. For each of the scenarios, a heatmap shows an overview of the protocol traffic and its associated metrics.

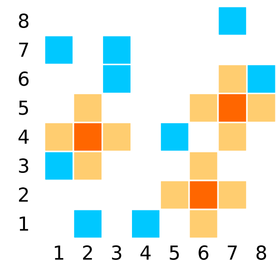
The X and Y axises represent a flat view of the processor's mesh array. The grey scale of the heatmap represent the value of the associated metric (the *number of messages*, the *cache hit rate* or the *cumulative distance*). For both data-sliding and mass-spring protocols, we use the same grey scale in order to facilitate the comparison.

D. *Cache hit rate and on-chip traffic trade-off*

As expected, the experiment based on the first scenario shows that the number of cache hits has been enhanced using the mass-spring migration model (115 hits), compared to the 1-hop migration mechanism (88 hits).

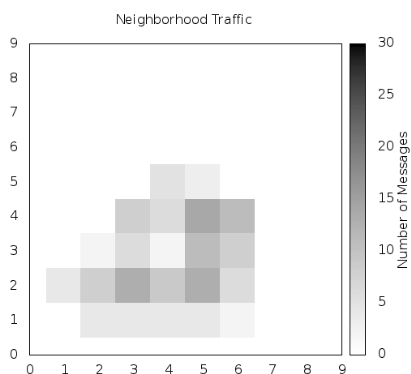


(a) B: four hot-spots forming a square, and the sparse home-nodes.

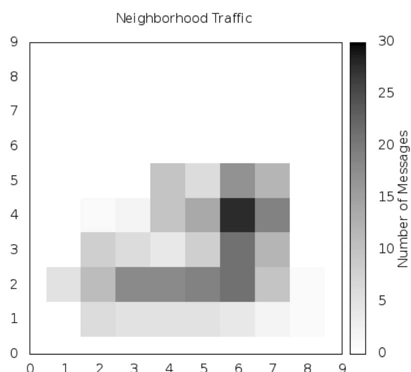


(b) C: Three random hot-spots, and the sparse home-nodes.

Figure 2: The used scenarios mapped onto a $8*8$ chip: dark orange is for hot-spots, soft orange is for stressed neighbors and blue is for home-nodes.



(a) Original data sliding protocol



(b) Mass-Spring protocol

Figure 3: Neighbors message traffic

In figure 3, we then compare the cooperative area traffic between the 1-chance (fig. 3a) and the N-chance (fig. 3b) forwarding strategies. Requests to neighbors traffic show that the 1-Chance sliding mechanism generates less traffic to neighbors (158 messages). Whereas, as the migrated data are most likely to be accessed, frequent accesses to remote data yield to an important neighbor-to-neighbor traffic (264 messages) in case of using the mass-spring sliding model.

Otherwise, the same figure shows that the cooperative neighborhood area is quite larger while using the new sliding mechanism. As the 1-HOP sliding is limited to direct neighborhood, data cannot migrate farther in the chip, thus it is quickly ejected. Whereas, the proposed protocol allows data to look for farther free cache space on the chip which leads to wider cooperative area. However, the spring's stiffness of the model allows to control the data spread over the chip in order to keep it as near as possible from their owner cores.

On one hand, our mechanism proposes a trade-off between caching data farther on the chip and the substantial penalty of ejecting it out of the chip. On the other hand, the mass-spring model promotes cheaper and efficient access to data at the expense of local traffic.

It seems obvious that the final choice for using such a N-chance forwarding policy over a regular cache protocol has to be made according to the intricacies of the chip network topology. Some topologies, even the single 2-dimension mesh we consider in these experiments, may be inappropriate: too much communications between neighbors may significantly decrease the application performances. Nonetheless, the mass-spring model should largely benefit from dense mesh networks, if not 3-D stacked, that offer aggregated bandwidth and low latencies. Such network include for example the 6-mesh NoC featured with the Tilera Gx processor.

E. Distance-aware Migration

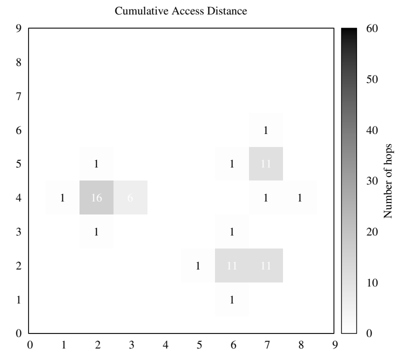
For the second experimental phase, we used the scenario of 3 hot spots distributed as described in the figure (figure 2b). We initially saturate both of the first and the second roles. Afterwards, we stress the first one twice more than the second within frequently accesses to different sets of data. All the highly stressed first role cores have the same number of accesses (24 accesses). This permits to fairly compare the cores behaviour.

Frequent accesses to remote migrated data generate additional costs. Thus, we analyze through these experiments the access cost using the *Cumulative Access Distance* metric. This metric is defined as the number of hops corresponding

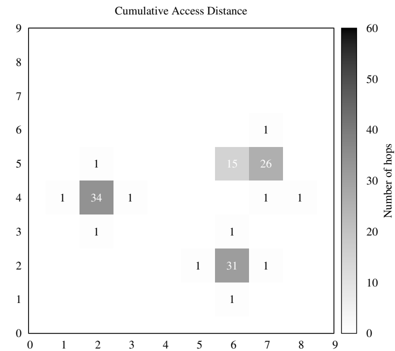
to the global remote accesses performed by each core. The figures below show results using the mass-spring model, compared to the described naive approaches (figure 4).

We deduce from these figures that the *Cumulative Access Distance* per core is remarkably reduced when using the proposed migration model. On one hand the *Random-Based protocol* and the *Load Balancing protocol* solicit randomly storage support from all the on-chip cores, it gives more sliding freedom to data. On another hand, the mass-spring protocol considers distance and destination load in each data migration decision which limits the data sliding zone.

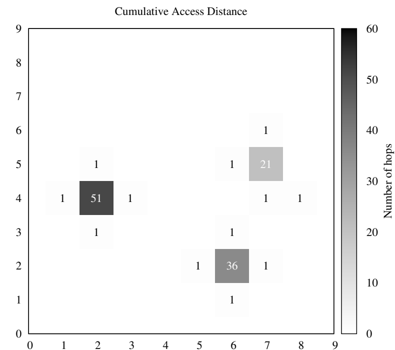
Such naive approaches are efficient when the global on-chip workload is slightly distributed. However, when solicited areas are far from the requesting cores, distance cost is increasingly important. The proposed data sliding strategy with the spring physical model allows to move data back once the neighborhood load is reduced. This leads to enhancing access cost and promoting data storage locality.



(a) Mass-Spring protocol



(b) Random-Based protocol



(c) Load Balancing protocol

Figure 4: Data Access Distance

VI. CONCLUSION AND PERSPECTIVES

As the number of cores is expected to increase in many-core architectures, the data management becomes prevalent and shares some similarities with large distributed infrastructures. Cooperative caching can be used to virtually extend the private caches, reducing data evictions and cache misses. In this paper, we have presented an extension of the sliding mechanism that allows a data to migrate from neighbor to neighbor. The decisions are made following the spring physical model, that offers relevant constraints to move data to the less saturated core and to stay close to the owner. It is also possible to set physical parameters to describe data priority and the cooperative area size. This model is implemented using local core information and a simple formula that could even be set into the hardware.

Analytical results show that in presence of multiple juxtaposed hot-spots, this approach decreases the number of cache misses compared to the regular cooperative data-sliding protocol. This enhancement has to be put into perspective with the growing traffic that can be observed in the neighborhood. Further experiments should determine if this local traffic is worth to generate in comparison to the costly external accesses.

This work can be expanded with a multi-spring model. In this model, a data is attached thanks to a spring to each core wherein a thread is accessing in shared mode. This should dynamically constrain data migration to equally serve all accesses. Another perspective concerns the setting of the spring protocol parameters: the data mass and the spring constant can be statically or dynamically adapted, depending on the application performance requirements or the current chip status. Some off-line decisions, based on operational research, should be taken by the compiler in order to tune the spring protocol for each shared data and each computing step in the application.

ACKNOWLEDGMENT

We would like to thank Jean-Thomas Acquaviva for the help he provided with the cache validation simulator, and the modified pinatrace tool during the validation phase.

REFERENCES

[1] “The kalray mppa 256 manycore processor.” Kalray S.A. http://www.kalray.eu/.

[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] "Epiphany-iv 64-core 28nm microprocessor." http://www.adapteva.com/products/silicon-devices/e64g401/, 2012.

[4] 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.

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

[6] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st conference on Computing frontiers*, CF '04, (New York, NY, USA), pp. 162–, ACM, 2004.

[7] 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.

[8] S. Dahmani, L. Cudennec, and G. Gogniat, "Introducing a data sliding mechanism for cooperative caching in manycore architectures," *Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pp. 335–344, 2013.

[9] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," 2012.

[10] X. Ding, K. Wang, and X. Zhang, "Srm-buffer: an os buffer management technique to prevent last level cache from thrashing in multicores," in *Proceedings of the sixth conference on Computer systems*, pp. 243–256, ACM, 2011.

[11] J. Meng and K. Skadron, "Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling," in *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, pp. 282–288, IEEE, 2009.

[12] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 60–71, ACM, 2010.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] S. Dahmani, L. Cudennec, and G. Gogniat, "Adaptive cooperative caching for many-cores systems," *Proceedings of the 9th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems*, 2013.

[22] E. Herrero, J. González, and R. Canal, "Power-efficient spilling techniques for chip multiprocessors," in *Euro-Par 2010-Parallel Processing*, pp. 256–267, Springer, 2010.

[23] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Cooperative caching: Using remote client memory to improve file system performance," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, p. 19, USENIX Association, 1994.

[24] I.-W. Ting and Y.-K. Chang, "Improved group-based cooperative caching scheme for mobile ad hoc networks," *J. Parallel Distrib. Comput.*, vol. 73, pp. 595–607, May 2013.

[25] D. Dominguez-Sal, J. Aguilar-Saborit, M. Surdeanu, and J.-L. Larriba-Pey, "Using evolutive summary counters for efficient cooperative caching in search engines.," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 4, pp. 776–784, 2012.

[26] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: a unified mechanism to address both cache pollution and thrashing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, (New York, NY, USA), pp. 355–366, ACM, 2012.

[27] N. P. Jouppi and S. J. Wilton, "Tradeoffs in two-level on-chip caching," in *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*, pp. 34–45, IEEE, 1994.

[28] 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.