Pattern based cache coherency architecture for embedded manycores

J. Marandola, S. Louise, L. Cudennec

Procedia Computer Science

Volume 80, 2016, Pages 1542–1553

ICCS 2016. The International Conference on Computational Science

Pattern Based Cache Coherency Architecture for Embedded Manycores

Jussara Marandola[1], Stephane Louise[2], and Loic Cudennec[2]

[1] Research work done when employed at CEA, LIST
[2] CEA, LIST, Saclay, France
stephane.louise@cea.fr,loic.cudennec@cea.fr

Abstract

Modern parallel programming frameworks like OpenMP often rely on shared memory concepts to harness the processing power of parallel systems. But for embedded devices, memory coherence protocols tend to account for a sizable portion of chip's power consumption. This is why any means to lower this impact is important.

Our idea for this issue is to use the fact that most of usual workloads display a regular behavior with regards to their memory accesses to prefetch the relevant memory lines in locale caches of execution cores on a manycore system.

Our contributions are, on one hand the specifications of a hardware IP for prefetching memory access patterns, and on another hand, a hybrid protocol which extends the classic MESI/baseline architecture to reduce the control and coherence related traffic by at least an order of magnitude. Evaluations are done on several benchmark programs and show the potential of this approach.

Keywords: Chip Multi-Processor, Cache Coherence Protocol, Memory Access Patterns

1 Introduction

Today's best way to improve processor's performance without increasing too much power consumption and heat dissipation issues would be to increase the number of processing cores on the chip. But this means good parallelization of applications. There are only a few programming models that can cope easily with manycores. Among them, one of the most popular is OpenMP.

Nonetheless, OpenMP relies on an execution model which states that memory is shared in a consistent state (memory coherence) between cores. For embedded computing it is an issue as memory coherence is a cause of timing hazards so real-time applications are difficult to implement with such feature, and also a cause of increased power consumption.

Our approach is based on a hardware/software co-design to optimize the cache coherence protocol. We worked with baseline architecture, above a Chip MultiProcessing (CMP) architecture to deploy a model of shared memory on the manycore. In terms of performance, the

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.481

main goals was to obtain a better speedup in applications such as video compression, and other video and signal processing applications, as often met in embedded systems. It is based on the fact that most of these applications display a very regular access to shared memory, so we can use pre-compiled memory access pattern tables to achieve this speedup.

Our optimized architecture focused on these memory access patterns, locality and cache, write policies and optimization cache coherency systems. The main contributions regarding our Co-Design Cache Coherent Architecture (*a.k.a* CoCCA) are:

- An Integrated Part (IP) to manage memory access patterns; the pattern table optimizes the number of messages sent during cache coherency transactions;
- A first example of pattern format specification and pattern table structure;
- A model of transaction messages and read/write operations;

The remainder of this paper is organized as follows: section 2 presents similar projects in a brief state of the art and our motivations; section 3 explains our architecture and how we optimized it to take the best advantage of regular memory access patterns that occurs in applications, the principles, and the associated protocol; Section 4 shows our experimental evaluations. Section 5 gives an overview of related works before the conclusion. Most of this work was done as part of a PhD thesis research work at CEA, LIST.

2 Context and State of the Art

2.1 Architecture of Cache Coherence

We are interested by manycores systems, or more specifically shared memory architecture with a number of cores typically between 10 and 100. Usually, processor's cores are connected by a Network on Chip (NoC) with a mesh topology as an often met implementation. In CMP architectures with shared memory, we analyze the data consistency stored in main memory and distributed in different levels of cache (distributed in many nodes for read/write operations). In order to illustrate the behavior of the baseline protocol, we consider a CMP architecture. Each core of the machine hosts a L1 instructions and data caches, a L2 cache, a directory-based cache, a memory interface and a network (NoC) interface. The directory-based cache hosts coherency information of a given set of data stored in the cache. The coherency information is a set of $(N+2)$ long bit-fields, sorted by memory addresses, where N is the number of cores in the system. Traditionally, the coherency information is composed by 2 bits (e.g. MESI state) representing the coherence state, plus an N bits-long presence vector.

The length of atomic units for the cache coherence protocol is called its granularity. A fine granularity would be settled at the cache-line level, whereas coarse granularities are set at memory-page level. The choice of a granularity impacts the properties of the cache and the performance of the memory hierarchy when a core access a set of data.

In some cases, it is possible to see a saturation of coherence messages that access the same core in the system. Indeed, for each data element managed by the coherency protocol, a dedicated node, named Home-Node (HN), is in charge of managing coherency information associated with it. In the literature, many cache coherence protocols, such as proximity-aware [3], MESI and MESIF [6] derivate from the baseline protocol. Particularly, the MESI protocol gave birth to a whole taxonomy of related protocols which will not be discussed further here.

Shared Memory CMP Architectures are expected to host up to several hundreds of cores. These cores are connected through a scalable network (Network on Chip, NoC). In this context,

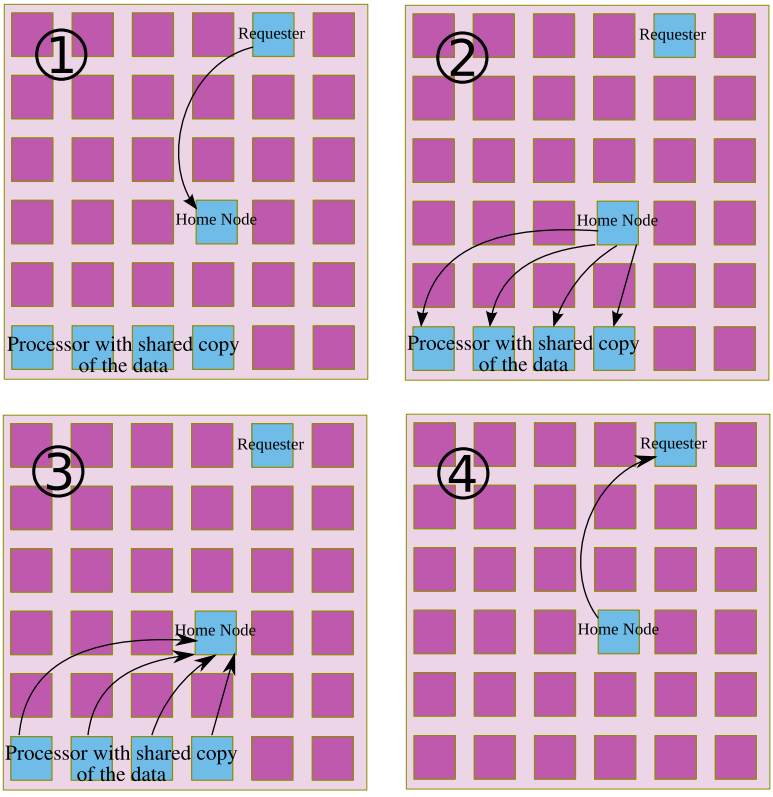


Figure 1: Baseline Protocol: a Simple Write Request

data coherence occurs when data are replicated on different cache memories of cores, due to concurrent read and write operations. Versions of data may differ between cores and with main memory. In order to maintain consistency, one popular approach resides in the use of a four-state, directory-based cache coherence protocol. This protocol, called baseline protocol, is a derivative protocol of the Lazy Release Consistency protocol[7], as found in some Distributed Shared Memory systems.

For the sake of clarity, we illustrate the baseline protocol through a very simple example involving one write request transaction. This transaction, as shown in Figure 1, triggers a sequence of messages transmitted between different cores. 1) The requester sends a message to the home node in charge of keeping track of the coherency information. 2) The home node checks the vector of presence and sends an exclusive access request to all the cores owning a copy of the data. 3) Then, all these cores invalidate their own copy and send an acknowledgment back to the home node. 4) Finally, the home-node grants the write permission to the requester and possibly transfers an up-to-date version of the data.

2.2 Home Node Management

As seen previously, each core of the system is the Home Node (HN) of a fraction of the cached data, and the coherence information of these data-sets is kept in the part of a memory of an extra storage named "*Coherence Directory*". The Home Node (HN) has the responsibility to

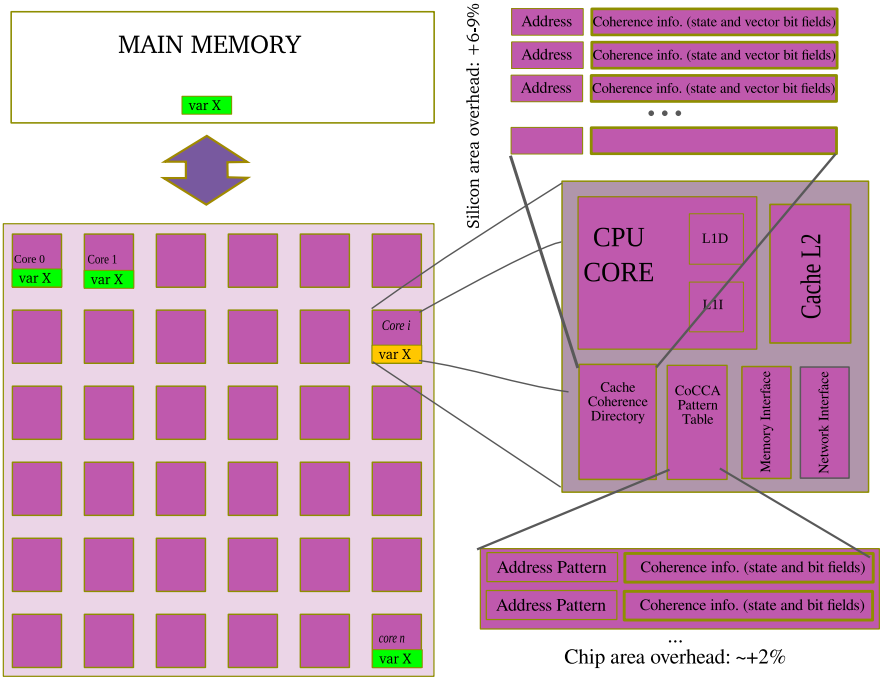


Figure 2: Principle of the hybrid architecture with both baseline and CoCCA protocols.

trace the consistency state of the data so that, when a processor needs a piece of data, it first checks with the HN for the coherency state. This task is handled by the coherence engine which sends/receives all messages related to shared memory access. The most usual way for allocating home nodes is a round robin way, by performing a modulo on a subset of the low-order bits of the address of the accessed data element.

One question is the granularity for state and HN attribution. As memory accesses are not distributed in homogeneous way for most of the programs, this leads to an uneven bandwidth consumption where some of the cores can become hot-spots. Therefore, to reduce hot-spot issues for cache management and to limit the number of messages of cache coherence protocol, we used a different granularity for the Baseline protocol and our Pattern-based CoCCA protocol. In our study, the Baseline protocol is fined-grained (64B or line-sized), whereas the ours is coarse-grained at a 2kB-page level.

3 CoCCA architecture and protocol design

The optimization of cache coherency traffic of CoCCA project is specialized for applications based on memory access patterns and one of our contributions concerns the management of memory access patterns. A new hardware-component and the new cache coherency protocol that implement the speculation of messages by memory access patterns.

The CoCCA protocol is a cache coherency hybrid protocol based on baseline protocol to which we add a speculative/prefetch protocol. Our first objectives were to provide the best cache politics to solve the problem of cache lookup and model of transaction messages in the system of cache for our cache coherency protocol. Current state-of-the-art architectures usually have

several levels of memory hierarchy. The main different between CoCCA and others architectures is that our Integrated Part (IP) stores the addresses in form of a table of patterns. The baseline protocol is used as a fallback.

Another contribution is the specification of the CoCCA protocol with its model of transaction messages that provides support for managing regular memory access patterns. The associated messages are called speculative messages. The CoCCA protocol is an hybrid protocol designed to interleave speculative messages and baseline messages through an IP that has the following purposes: store patterns and control transaction messages. The optimization of the CoCCA protocol is based on finding memory addresses of application matching a stored pattern. The requester sends the speculative message to the CoCCA Home Node (CHN) if it matches a stored pattern or otherwise, the requester sends the baseline message to the ordinary Baseline Home Node (BHN). Each CPU has the following IPs of cache hierarchy: L1 caches, a L2 cache (shared inclusive), a directory of cache coherence and the "*CoCCA Pattern Table*" (see Fig. 2).

3.1 CoCCA Pattern Table

Patterns are used to summarize the spatial locality associated to the access of data. The pattern table lookup process uses a signature of a pattern which is either its base address, or in a more general way a "trigger", i.e. a function that provides a specific signature of a pattern[1], as seen in figure 3). One can imagine a lot of different principles for patterns, but let us illustrate it with the simplest of them: 1D or 2D strided accesses, since it would cover a lot of data accesses encountered in embedded applications.

For 1D, CoCCA patterns would be defined as triplets:

$$Pattern = (base\ address, size, stride) \tag{1}$$

Where *base address* is the address of the first cache line of the pattern, *size*, the size of the pattern (number of elements), and *stride* expresses the distance between two consecutive accesses of the pattern. For 2D patterns we must add a second stride and a second length/size.

In Figure 3, we present some initial concepts about the Cocca Pattern Table, as any given access to memory is checked against a function called trigger itself associated with a pattern definition. The simplest trigger that can be defined is very simple: it is a true function only for the access at a given predefined address. Hence every time a failed access is done at the address associated with this trigger, the IP fetches the pattern(s) it is associated with. This is the simple trigger function we tested in this paper, but we could imagine more complex trigger functions for more specific and less error prone pre-fetch. This is still considered as future work.

The base architecture is a multi-core system, each core fitted with its memory hierarchy (L1, L2), Directory and Pattern Table, and all cores have access to a Network on Chip (NoC) that permits each core to communicate with one another and with main memory. The CoCCA protocol optimizes the coherency protocol for the recognized memory access patterns of a given application running on the system (Fig. 2).

The main interest of pattern tables, is that the ranges of addresses that are defined by patterns provide a way to enhance the baseline protocol (MESI modified protocol) by authorizing a speculative coherence traffic which is lighter (*i.e.* with less message throughput) than the baseline protocol alone. Hence, it accelerates shared memory accesses (see Figure 4).

[1] Of course the simplest trigger, and the one we implemented in our evaluation in this paper, is the use of the base address.

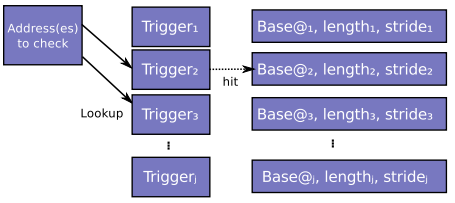


Figure 3: Cocca Pattern Table lookup principle: in a first implementation, triggers are the base addresses of patterns

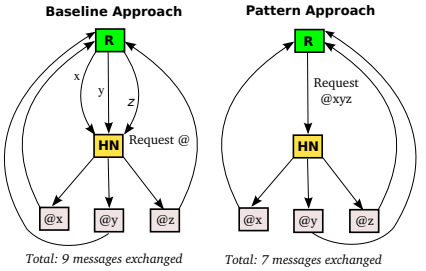


Figure 4: Baseline and Pattern Approach comparison for a sequence of read requests

In case of a cache miss, the flow of message transaction defined by the MESI modified protocol is sent by requested addresses (misses). When the pattern table and the speculative protocol (CoCCA specific) is added and a pattern is triggered, the flow of message transaction can be optimized in term of messages, because the pattern provides a means to use speculative messages which are in fewer number than in the baseline protocol. As a conclusion, when a pattern is discovered in our approach, the number of transaction messages is reduced by using speculation, leading to a better memory access time, less power consumption and an optimized cache coherency protocol.

In Figure 4, we compared two approaches of cache coherency protocols for a cache miss case. We present two scenarios: the baseline only approach, where the node requested sends the sequential addresses x, y, z, totalizing 9 messages; and the pattern approach where the node send speculatively the pattern with xyz, totalizing only 7 messages, and a early (speculative) prefetching of data in the cache. It is worth noting the interest of the CoCCA protocol grows linearly with the number of fetch in the pattern.

3.2 Hybrid Protocol

The pattern table permits the management of a hybrid protocol to improve the performance of transaction messages in the system memory. The hybrid protocol was specified for Cache Coherent Architecture to optimize the flow of messages, permitting both utilization of baseline messages and speculative messages. To avoid hotspots of messages in the systems by using different levels of granularity for each protocols.

Figure 5 shows the read transaction message model in the coherency hybrid-protocol that describes the key rules: requester, CoCCA home node, baseline home node. Each read access triggers the search of its address in the pattern table. If the pattern table lookup returns true, the base address is sent to the hybrid home node. Otherwise, the baseline message is sent to the baseline home node (of course, both table lookup can be done in parallel).

For the hybrid protocol, the rules can be divided in: core that request the data (requester) –the baseline home– and the hybrid home node. The baseline home node is the core appointed by fine granularity (line) while the CoCCA home node is define at page-level (2kB).

The first model of decision tree is based on the read transaction message, where the pattern table lookup is similar to a cache lookup as seen in figure 6. The request read presents 3 states : requester, baseline home node and hybrid home node. This formal model is a tree of make-decision for transaction messages during read requests.

This schematic decision tree for read accesses describes data lookup in cache. In the case of a pattern table hit, the speculative message is sent to the CoCCA HN (determined by page

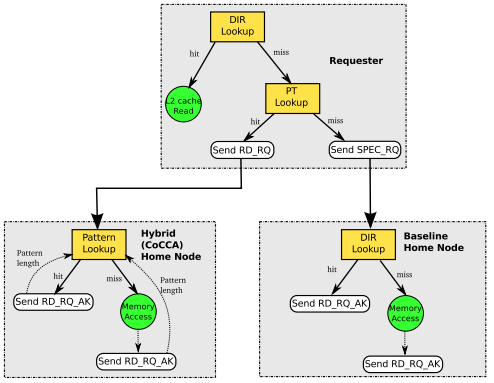


Figure 5: Model of Transaction Messages including the Home Nodes

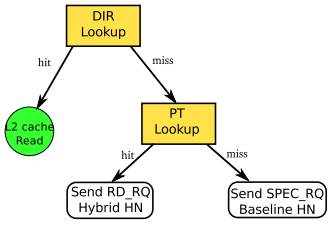
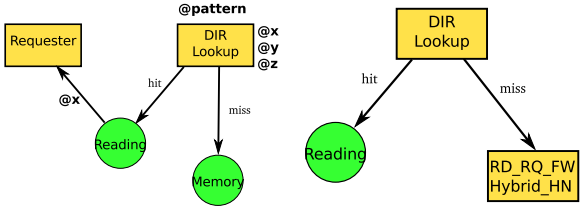


Figure 6: Read Transaction Message



(a) Hybrid Home Node (b) Baseline Home Node

Figure 7: The 2 available protocols: baseline and hybrid.

granularity). For write sequences, the principles are the same and can be extended the same way. Another case, as seen in Figure 7a, show how a read request can be handled at the CoCCA HN. Reciprocally, the baseline home node on Figure 7b can handle a request to the CoCCA HN in case of miss.

4 Experimental evaluation

We looked at several parallel applications mainly from StreamIt benchmark suite [13] (Filterbank) which are fine-grained, SDF^3 benchmark suite [12] (H.263 decoder, H.263 encoder, Rate converter) which is really coarse grain, and a few in-house applications (e.g. a Laplacian filter) from the ΣC programming tutorial which are somewhat in-between.

4.1 Protocol of experimentation

For each, we transformed the application (usually written in a stream language like StreamIt [1] or Sigma-C [2]), so we used the code generated at the task level (C-code) and used a simple bulk-synchronous scheduler to generate the execution. A bulk synchronous execution is very simple, but still efficient if the execution times of the different tasks are not too different.

We used Intel's Pin binary analyzer, and more specifically, an in-house modified version of the pinatrace *Pintool*, which traces all the memory accessing instructions (address of the instruction, address of the access and its lenght, whether it is a read or write instruction and

on which core the instruction is executed).

From the hundreds of millions of memory accesses each application generated, we extracted the accesses to shared memory, and generated the patterns by using a simple algorithm inspired by the principle of the Hough transformation on the memory accesses.

Table 1: Characteristics of the tested benchmark programs.

Program	Benchmark	Shared Mem (total)	Min FIFO size	Max FIFO size	core#
H263 decoder	SDF^3	2.6 MB	16kB	600kB	4
H263 encoder	SDF^3	36 MB	16kB	12MB	5
Sample rate	SDF^3	2 kB	56B	320B	6
FilterBank	StreamIt	6.1 kB	8 B	264 B	27
Gaussian blur (HDTV)	ΣC	15MB	157kB	157kB	49
Laplacian (256 × 256)	ΣC	448 kB	12 kB	12kB	9

In the following, we will discuss the case of the Laplacian transform from the introduction to the Sigma-C programming language because it has all the characteristics of a typical embedded parallel application: It displays a good level of parallelism (both data parallelism and pipeline parallelism) and a heterogeneous task-set whilst it remains simple enough to explain (at least in its simplest implementation).

The filtering is done in a unusual way here: access are always done in line (except for the transpositions) and each way of the 2 parallel way works on one line on two. This choice of implementation has a peculiar incidence on patterns, as we will see.

4.2 Extracted patterns and analysis

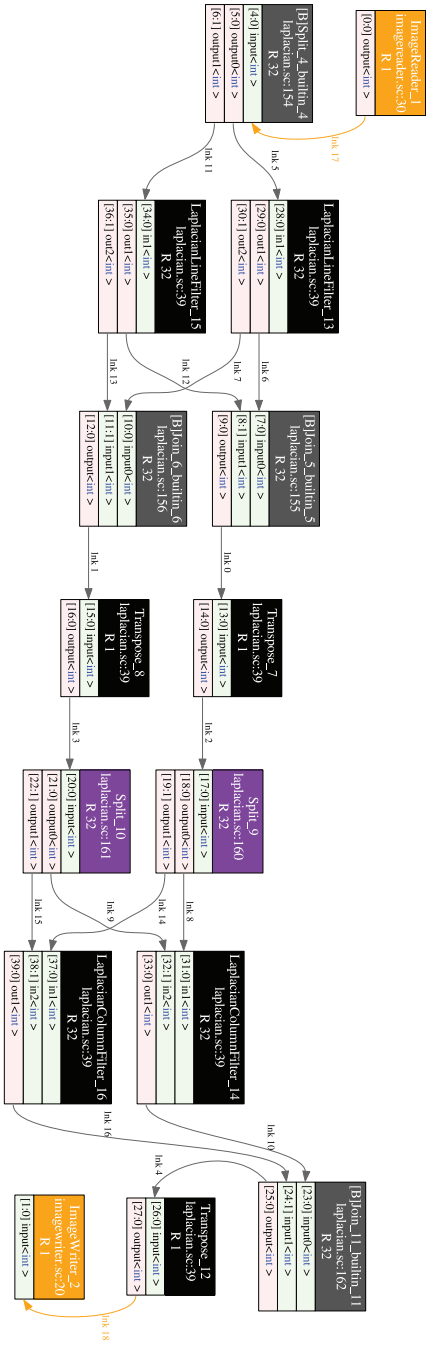
All these applications displayed very regular behaviors with regards to their accesses to shared memory: either the computation is done locally on a core and this can work without memory coherence, or the shared memory spaces allow the use of our memory patterns. Sometimes, it is required to remove false sharing from applications (*i.e.* 2 cores using the same line of caches because data elements used by one and data elements used by the other are close in address) to achieve this result, but it is very consistent among applications. We can assert with a good confidence that this would be the case for all SDF (Synchronous Dataflow) or CSDF (Cyclostatic Dataflow), because reading and writing tokens in FIFO display a very regular behavior. This behavior is very typical of computational embedded applications. This changes for highly dynamic dataflow, which are usually avoided because their behavior is hard to control at large scale (the execution determinism is often not guaranteed before run-time), and in the case when the working data-set does not fit in the data cache (*e.g.* typically h263 encoding and some parts of h263 decoding). In these case, we can not use our protocol out of the box, but we must rely on the fallback baseline protocol.

For simple applications like the (homogeneous) Gaussian blur (Fig.8b), the number of 1D patterns required to account for a perfect prefetch is low: less than 5 patterns (see also [8]).

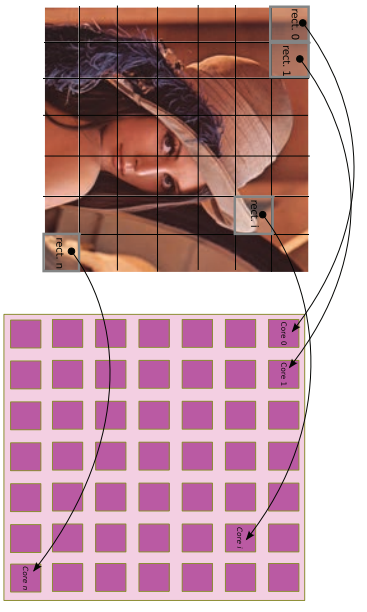
For applications with heterogeneous task-sets like the Laplacian (Fig.8a), and the same with similar programs like the FilterBank program from the StreamIt benchmark suite, the situation can be sometime a bit more complex if we limit our scope at 1D patterns.

We developed a simple tool to extract the access patterns:

Raw patterns	Cache-line patterns	Optimized pattern-set
0x7e43c0 0x4 256 R 0x664380 0x4 256 W	(0x7e43c0, 4, 0x40, R)	(0x7e43c0, 256, 0x100, R)
0x7e4bc0 0x4 256 R 0x664b80 0x4 256 W	(0x664380, 4, 0x40, W)	(0x664380, 256, 0x100, W)
0x7e53c0 0x4 256 R 0x665380 0x4 256 W	(0x7e4400, 4, 0x40, R)	(0x7e4400, 256, 0x100, R)
etc.	(0x6643c0, 4, 0x40, W)	(0x6643c0, 256, 0x100, W)
	...	...



(a) The CSDF graph as output by the Sigma-C compiler of the Laplacian application. The exchange buffers that implements the FIFO (shared memory) are also shown.



(b) Attribution of image parts to cores in Gaussian blur application.

Figure 8: Laplacian and Gaussian Blur application graphs.

After this step, all you have to do is to discard patterns corresponding to the same cache line, as they would have been already loaded, and check with the write patterns to avoid premature reloading of a pattern whose data values is unchanged.

For the Laplacian program, the access on the line is done every two lines for LineFilter tasks and (Transposed) ColumnFilter tasks, then, for an image of 256×256, we have 2 times (one for read and one for writes) 128 short (4 cache-lines) patterns. Such case is a good candidate for 2D stride definitions, but if required, we can use a trick to lower drastically the number of 1D patterns: instead of defining patterns for lines (which are short an numerous), we can define patterns for columns (*i.e.* the transposed patterns) and use the fact that cache-line loading can factorize a handfull of accesses, and therefore lower the effective number of require patterns. The trick can be easily generalized: as a set of pattern is identified, it is possible to try and lower the number of patterns by choosing another equivalent (in memory accesses) set of patterns with a lower cardinality. Therefore accessing to 128 lines of 8 conterminous cache-lines is equivalent to accessing 8 conterminous columns of 128 cache-lines. One drawback is that the access are not along a column, so the first line accessed may be plagued with cache misses. This can be easily overcome by using another pattern for the first line. This way, we can transform 1D complex pattern set with more than 256 elements to another 1D simple set with only 18 elements. The results are shown in Table 2.

Table 2: Pattern characteristics per schedule-firing (2 firing for 1 complete execution)

Program	Task	Read Patterns	Invalidation Patterns
Gaussian blur	Gaussian filter	5 (1D)	5 (1D)
Laplacian	ImageReader_1	0	18 (1D) or 1 (2D)
	LineFilter_13/15	9 (1D) or 1 (2D)	9 (1D) or 2 (2D)
	Transpose_7/8	1 (1D)	1 (1D)
	ColumnFilter_14/16	18 (1D) or 2 (1D)	9 (1D) or 1 (2D)
	Transpose_12	1 (1D)	1 (1D)
	ImageWriter_2	1 (1D)	0
SDF^3	(all)	1 (2D)	1 (2D)

Most of the time a few 1D patterns is enough to account for the majority of the shared-memory access we experimented, but a few 2D entries would help a lot in reducing the number of required entries. The conclusion we can draw with streaming embedded applications show that we can have a mix of 2D and 1D patterns for the CoCCA IP, with 8 to 16 entries of each kind. It would easily embrace most of the usual cases, and for the cases which does not, we would have been limited by the size of the cache anyway.

As a conclusion of this study, for usual streaming application, our prefetch pattern-based augmented coherence protocol a few dozen entries (e.g. 32) would account for the vast majority of the cases, even with simple 1D pattern definitions. When using 2D memory-access patterns, the number can even be reduced further a bit.

We have evaluated that on Xeon Nehalem cores the acceleration of kernels given by the protocol by comparing the execution without prefetch and with a prefetch. The induced prefetch done by our protocol can enhance the performance in significant ways: 70% for the Gaussian blur which is heavily memory bound (the Gaussian task run in a bit less than 4490.10^3 cycles on this core, with preloaded caches *i.e* no cache miss, for 965328 shared memory accesses including 37128 write accesses and 928200 read accesses; without prefetch we have 17283 read misses with memory and about 13000 cache sharing requests). Even for the Laplacian, which is much less memory bound, we have around 30% speedup (limited by the worst case tasks LineFilter and ColumnFilter) thank to the prefetch of our protocol. The other advantage is that the baseline protocol (acting as fallback) is not used at all in all these programs. We expect that it would

be translated as a way to lower the power consumption of the CMP with our IP.

One important point is that we can limit the number of entries in the Pattern Tables, therefore, it takes a limited amount of transistors to implement it and as it integrates with the standard MESI protocol, the overhead in both dice size and dedicated consumption should remain low.

5 Comparison with other works

In the literature, several projects propose to optimize data consistency protocols by supporting data access patterns. This has been explored in the fields of database systems, distributed shared memories or processor cache management.

In [5], Intel uses patterns as a sequence of addresses stored in physical memory. A dedicated processor instruction set is provided to apply patterns given a base memory address and an offset. A single call to these instructions can perform accesses to non-contiguous addresses in the cache. However this mechanism is limited to data stored in one cache. In the context of many-core computing, patterns may describe data that are owned by different cores of the chip.

In [4], IBM proposes to sort patterns by type: read-only, read-once, workflow and producer-consumer. Based on these types, patterns are stored in a specific hardware component. A dedicated processor instruction set is provided to detect and apply patterns. Here again, this mechanism is not fitted to the context of many-core computing, as it only applies patterns on a local cache.

The mechanisms of pattern lookup could be compared with conterminous group and locality [9]. Additionaly, we analyzed several cache politics such as Spatio-Temporal Memory Streaming (STeMS) [10], Spatial Memory Streaming (SMS) [11] and Temporal Streaming of Shared Memory (TMS) [14]. Recent Intel processor also nearly all include prefetchers. While the efficiency of these mechanisms is high, with good prediction ratio over 90%, they also have several major drawbacks when one wants to use them in the context of embedded software development. As dynamic prefetchers they need several memory accesses to find the pattern, and then they sometime fetch too much data, because they need to be very aggressive (with an associated high power consumption). Moreover for short pattern, they are poorly efficient, as they can lead to too much prefetch, therefore to cache pollution and lastly, the approach is difficult to scale up to hundreds of cores. Because in the embedded world cache are usually smaller, performance is closer to the minimum desired one and power consumption also matters a lot, we believe that a static approach is best for these kinds of applications.

6 Conclusion

In this paper, we proposed to manage memory access patterns at both software and hardware levels. A regular consistency protocol has been modified to handle speculative requests and a new hardware component has been designed to store and retrieve patterns.

This architecture offers some significant advantages for embedded manycores, as we have seen on the evaluation: a low number of pattern can account for the majority or even often all the shared memory accesses, thus reducing the number of messages on the NoC, and prefetching the caches of each core with data that are relevant to its computation, and therefore increasing performance in a significant way, with more than 20% for most of the programs to more than 70% speedup for memory bound programs. We can expect intuitively at the same time a reduction of the weight of memory coherence on power utilization of embedded CMPs for such

typical embedded applications. Our next step is to further evaluate the model, and if possible find a partnership with a hardware oriented laboratory to find out what we can expect for real on the power consumption side.

References

[1] Saman Amarasinghe, Michael I. Gordon, Michal Karczmarek, Jasper Lin, David Maze, Rodric M. Rabbah, and William Thies. Language and compiler design for streaming applications. *International Journal of Parallel Programming*, 33(2/3):261–278, Jun 2005.

[2] Pascal Aubry, Pierre-Edouard Beaucamps, Frédéric Blanc, Bruno Bodin, Sergiu Carpov, Loïc Cudennec, Vincent David, Philippe Dore, Paul Dubrulle, Benoit Dupont de Dinechin, Francois Galea, Thierry Goubier, Michel Harrand, Samuel Jones, Jean-Denis Lesage, Stephane Louise, Nicolas Morey Chaisemartin, Thanh Hai Nguyen, Xavier Raynaud, and Renaud Sirdey. Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. In Vassil N. Alexandrov, Michael Lees, Valeria V. Krzhizhanovskaya, Jack Dongarra, and Peter M. A. Sloot, editors, *ICCS*, volume 18 of *Procedia Computer Science*, pages 1624–1633. Elsevier, 2013.

[3] Jeffery A. Brown, Rakesh Kumar, and Dean Tullsen. Proximity-aware directory-based coherence for multi-core processor architectures. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 126–134. ACM, 2007.

[4] E. Debes, Y-K. Chen, M. J.Holliman, and M. M.Yeung. Using data access patterns. international business machines corporation, patent us 7,395,407 b2. Technical report, 2008.

[5] Zhuo Huang, Xudong Shi, Ye Xia, and Jih kwon Peir. Apparatus and method for performing data access in accordance with memory access patterns. Technical report, 2006.

[6] H.H.J. Hum and J.R. Goodman. Forward state for use in cache coherency in a multiprocessor system, July 26 2005. US Patent 6,922,756.

[7] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, November 1989.

[8] Jussara Marandola, Stephane Louise, Loic Cudennec, Jean-Thomas Acquaviva, and David A. Bader. Enhancing cache coherent architectures with access patterns for embedded manycore systems. In *System on Chip (SoC), 2012 International Symposium on*, pages 1 –7, oct. 2012.

[9] Xudong Shi, Zhen Yang, Jih kwon Peir, Lu Peng, Yen kuang Chen, Victor Lee, and Bob Liang. Coterminous locality and coterminous group data prefetching on chip-multiprocessors. In *In: Proc. of the 20th Intl Parallel and Distributed Processing Symp. Toronto: X-CD Technologies Inc*, 2006.

[10] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. *SIGARCH Comput. Archit. News*, 37(3):69–80, 2009.

[11] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. *SIGARCH Comput. Archit. News*, 34(2):252–263, 2006.

[12] Sander Stuijk, Marc Geilen, and Twan Basten. Sdf^3: Sdf for free. In *Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, pages 276–278, 2006.

[13] William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 365–376, 2010.

[14] Thomas F. Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Temporal streaming of shared memory. *SIGARCH Comput. Archit. News*, 33(2):222–233, 2005.