# Enhancing Cache Coherent Architectures with Access Patterns for Embedded Manycore Systems

Jussara Marandola, Stéphane Louise, Loïc Cudennec, Jean-Thomas Acquaviva, David Bader

# Enhancing Cache Coherent Architectures with access patterns for embedded manycore systems

Jussara Marandola*, Stephane Louise†, Loïc Cudennec†, Jean-Thomas Acquaviva† and David Bader*
* Georgia Institute of Technology
† CEA, LIST, CEA Saclay Nano-INNOV, Bat 862
PC 172, Gif-sur-Yvette, 91190 France
Email: jkofuji, bader @cc.gatech.edu
stephane.louise, loic.cudennec, jean-thomas.acquaviva @cea.fr

***Abstract*— One of the key challenges in advanced micro-architecture is to provide high performance hardware-components that work as application accelerators. In this paper, we present a Cache Coherent Architecture that optimizes memory accesses to patterns using both a hardware component and specialized instructions. The high performance hardware-component in our context is aimed at CMP (Chip Multi-Processing) and MPSoC (Multiprocessor System-on-Chip).**

**A large number of applications targeted at embedded systems are known to read and write data in memory following regular memory access patterns. In our approach, memory access patterns are fed to a specific hardware accelerator that can be used to optimize cache consistency mechanisms by prefetching data and reducing the number of transactions. In this paper, we propose to analyze this component and its associated protocol that enhance a cache coherent system to perform speculative requests when access patterns are detected. The main contributions are the description of the system architecture providing the high-level overview of a specialized hardware component and the associated transaction message model. We also provided a first evaluation of our proposal, using code instrumentation of a parallel application.**

## I. Introduction

Chip multi-processing (CMP) has become very popular lately, providing the power of massively parallel architectures on a single chip. One of the key challenges arising from these systems consists in designing the right programming model which would be as independent as possible of the underlying hardware. This is particularly critical in the field of data management between cache memories of many-core systems. In such architectures, each core may store a copy of a data element in its cache. Cache coherence is either directly managed by the programmer or falls under the control of a cache coherence unit (usually hardware based). This second solution makes all updates and data transfers transparent and also simplifies the development of applications. Unfortunately, it is known to have a cost in term of hardware design, refraining it from being massively adopted in embedded computing.

To make data coherence more attractive for massively-parallel embedded architectures, we think that cache coherency models and protocols should be tightly adapted to the needs of targeted applications. A large number of applications deployed on embedded devices focus on image, video, data stream and workflow processings. This class of applications tends to access data in a regular fashion, using a given set of memory access patterns. These patterns can be used to optimize the cache coherency protocol, by prefetching data and reducing the number of memory transactions.

Using memory access patterns has already been studied in the literature but, as far as we know, our way of mixing a software and a hardware approach is unique. In this paper we describe the system architecture of a hardware component proposed to store and manage patterns, and the associated protocol which takes advantage of them for optimizing memory consistency and access time. Our main contributions provide the state of the art of directory-based cache protocols, adding an optimization for regular memory access patterns. These contributions are part of the CoCCA project standing for Co-designed Coherent Cache Architecture. We also provide in this paper a first evaluation by code instrumentation of a typical parallel program.

The remainder of this paper is organized as follows: section II presents a brief state of the art and related works; section III 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 IV relates an analysis of a first parallel benchmark by tracking memory traces. Finally, section V concludes and gives some perspectives about this work.

## II. Memory Consistency and State of the Art

### A. Context: Cache Coherence for CMP Architectures

Shared Memory Chip Multi-Processor Architectures are expected to host up to hundreds of cores. These cores are connected through a scalable network (Network on Chip, NoC) usually based on a mesh topology. In this context, coherence issues occur 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 is to use of a four-state, directory-based cache coherence protocol. This protocol, called *baseline* protocol, is a derivative of the Lazy Release Consistency [1] protocol.

### B. Baseline Protocol: a Directory-based Cache Protocol

In order to illustrate the behavior of the baseline protocol, we consider a CMP machine. Each core of the machine hosts a

This work was done in part as the first author was in CEA, LIST

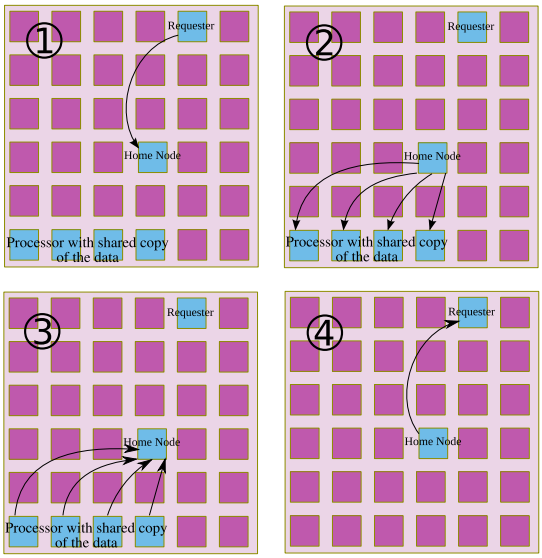


Fig. 1. Baseline Protocol: an example of a simple Write Request

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 (see also Figure 2 since the CoCCA approach only modifies the CMP architecture by adding the Pattern Table and and modifying the protocol). 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 representing the coherence state, plus a $N$ bits-long presence vector. The coherence state field represents four states, as defined by the MESI protocol:

- M (modified): a single valid copy exists across the whole system; the core owning this copy is called *Owner* of the data and has the right to write. The value of this copy has changed since the data was cached by the owning core.
- E (exclusive): a single valid copy exists across the whole system, the core owning this copy is named the *Owner* of the data and has the right to write. The data was not modified since it was cached by the owning core.
- S (shared): multiple copies of the data exist, all copy are in read-only mode. Any associated core is named *Sharer*.
- I (invalid): the copy is currently invalid, should not be used and so will be discarded.

The length of the presence vector is equal to the number of cores in the system: 0 at the $i^{th}$ bit means the data is not cached in core $i$, and 1 at the $j^{th}$ bit means it is cached in core $j$. For each data element managed by the coherency protocol, a dedicated node, named Home-Node (HN), is in charge of managing coherency information for this particular element. In the literature, many cache coherence protocols, such as proximity-aware [4], alternative home-node [5], MESI [6] and MESIF [7] derivate from the baseline protocol.

We can illustrate the baseline protocol on a write request transaction, as shown in Figure 1: it 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.

### C. Optimizing the Cache Coherence Protocol

The number of messages generated by the coherency protocol is one of the most important criterion used to evaluate the overall performance. In section II-B, we have seen that a simple write request generates a four-step transaction with up to 10 messages sent over the network.

In a more sophisticated case, we can imagine an application accessing a picture column by column. This type of access cannot be handled by the baseline protocol in only one transaction. This simple example shows a case where the baseline approach falls into a worst-case scenario.

Working on columns in a picture can be achieved with the help of data access patterns. Patterns can be used to speculate on the next accesses, prefetching data where they will be most likely used in a near future. Patterns can also be used to save bandwidth, by reducing the number of protocol messages: one transaction can provide access to a whole set of data.

### D. Related works

*1) Exploiting Data Access Patterns:* 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 [2], Intel uses patterns as a sequence of addresses stored in physical memory. A dedicated 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 [3], IBM proposes to sort patterns by type: read-only, read-once, workflow and producer-consumer. Corresponding patterns are stored in a hardware component. Dedicated processor instructions are 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.

*2) MPSoC and other platforms:* Our Cache Coherence Architecture was developed with MPSoC (Multiprocessor System-on-chip) in mind. This architecture is based on a multicore system with state of the art shared memory and paradigm of parallel computing. Platforms such as hybrid or heterogeneous systems (*e.g.* composed with CPU+GPU) could adopt our model of architecture.

For high performance computing, we aim at the kind of following hybrid systems: multicore processor with GPU (Graphics Processing Unit), NVIDIA Tesla (Cluster of GPU

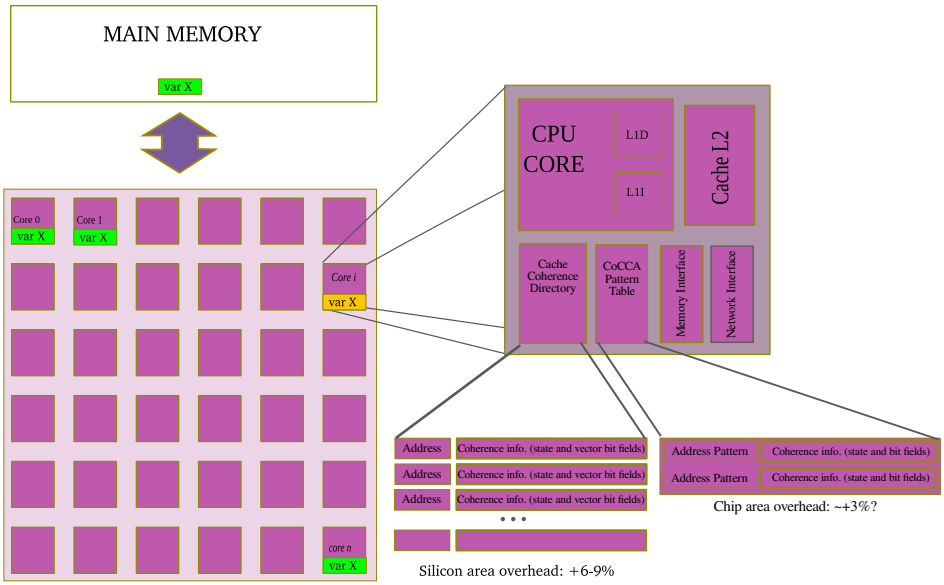


Fig. 2. CoCCA Architecture showing the addition of the Pattern Table to the cache hierarchy

processors), AMD FireStream (AMD processor + GPU), all processors presenting a new programming paradigm called HMPP - Heterogeneous Multicore Parallel Programming. This paradigm is dedicated to embedded and superscalar processors.

A related work is the TSAR project that describes a multi-core architecture with a scalable shared memory that supports cache coherency. The TSAR system [9] aims to achieve a shared memory architecture including thousands of RISC-32 bit processors.

## III. CoCCA architecture and protocol design

### A. Principle and motivations

The main contribution is the specification of the CoCCA protocol of transaction messages that provides support for managing regular memory access patterns. The associated messages are called speculative messages. The CoCCA protocol is a hybrid protocol designed to interleave speculative messages and baseline messages through a hardware-component 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 (or Hybrid Home Node, HHN) if it matches a stored pattern or otherwise, the requester sends the baseline message to the ordinary Baseline Home Node (BHN). This optimized method enhance the performance of cache coherency traffic, aiming for the following advantages:

- reduction of throughput of messages,
- lower time of memory accesses.

In the next sections, we will present the design principles of our architecture, a first specification of simple patterns, the bases of our protocol and the associated data structures.

### B. CoCCA Architecture principles

A typical implementation of the CoCCA Architecture would have several dozens of cores for a start. Each CPU core of the system may be involved in exchange of coherence messages, taking *four* different roles with regards to data, *i.e.* a core can play one role (or several) in transactions related to a given data, and assume different roles for different data.

- Requester, the core asking for a data.
- Home Node, the core which is in charge of tracking the coherence information of a given data in the system.
- Sharer, a core which has a copy of the data in its cache. This copy is in "*shared mode*", *i.e.* multiple copies of this data can exist at the same time for several cores.
- Owner, a core which has a copy of the data in its cache. This copy is in "*Exclusive*" or "*Modified*" mode, so one and only one instance of it can exist at this time across the whole system.

Each CPU has the following components of cache hierarchy: L1 caches, a L2 cache (shared inclusive), a directory of cache coherence and the "*CoCCA Pattern Table*", as seen on figure 2.

### C. 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 lot of different principles for patterns, but let us illustrate it with the simplest of them: the 2D strided access, since it would cover a lot of data accesses encountered in embedded applications.

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

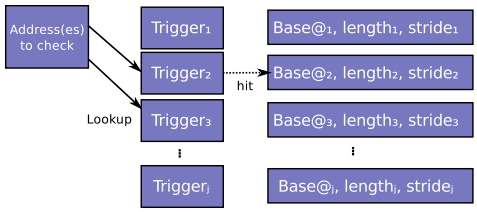


Fig. 3. Cocca Pattern Table lookup principle: in a first implementation, triggers are the base addresses of patterns

Such CoCCA patterns would be defined as triplets:

$$Pattern = (baseaddress, size, stride)$$

Where $baseaddress$ 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.

In figure 3, we present some initial concepts about the Cocca Pattern Table. 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 (figure 2). The CoCCA protocol optimizes the coherency protocol for the stored memory access patterns of a given application running on the system.

The pattern descriptor enables to describe the CoCCA pattern table entry:

$Desc = f_n(B_{addr}, s, \delta)$, with:

- $f_n()$ the function that build the pattern of length $n$ with the given characteristics,
- Desc the pattern descriptor that results from applying function $f_n()$ with the parameters $B_{addr}$, $s$ and $\delta$,
- $B_{addr}$ represents the offset address, regarding the first address of many addresses composed by pattern access on address lookup,
- $s$ is the size of the pattern, or number of elements,
- $\delta$ is the stride between two given accesses in the pattern

We can define an example of pattern access:

$$\{1,4,2\}\ following\ @1+1 = @2$$
$$and\quad \{1,4,2\}(@1) = (@2, @5, @8, @11)$$

So applying address @1 to the pattern $\{1,4,2\}$ is a series of 4 addresses starting at @2 (@1 plus 1 offset) and with an interval of 2 addresses not belonging to the pattern between two successive addresses. This defines the base addresses of our simple patterns. In general there can be more than one address for each element of the pattern: this is given by an extra $n$ parameter, which defines the length of each access in the pattern.

In our future work we want to simulate our principles using a transaction level model (TLM) like simSoC, but a first approach of cache coherence architecture was developed through an API that describes the hardware behavior. The implementation used the C language where Typedef structure modelize hardware storage and the API describes execution of a special instruction set to manage pattern tables.

A pattern table is similar to a hash table describing pattern ids (or triggers, as seen previously) and associated patterns.

*Definition of Pattern Structure*

```
typedef struct Pattern_ {
unsigned long capacity; /* sizeof(address) */
unsigned long size;     /* address number */
unsigned long * offset; /* pattern offset */
unsigned long * length; /* pattern length */
unsigned long * stride; /* pattern stride */
} Pattern_t;
```

Regarding the pattern table, we described it as composed by patterns. Our pattern structure is based on an associated stride; the key elements that compose the pattern are: offset, length and stride. To this, we added the capacity and a size. The capacity represents the memory space required to store each pattern and size is the space in memory.

In our first approach toward embedded systems, we thought that pattern table can be the result of the compilation process of an optimized application. Therefore, pattern tables can be fetched as part of a program, by using a specialized set of instructions. For our evaluation, a library is used to simulate the use of these special instructions:

- PatternNew(): function to create a pattern,
- PatternAddOffset(): function to add an offset entry,
- PatternAddLength(): function to add a length entry,
- PatternAddStride(): function to add a stride entry,
- PatternFree(): function to release the pattern after use.

### D. Protocol and Home Node management

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, interleaving baseline messages and speculative messages. We introduce the concept of granularity of messages to avoid hotspot of messages in the systems.

The specification of this hybrid protocol presents the following characteristics:

- Difference between baseline and speculative messages,
- Speculative messages that permit to read all addresses of pattern through their base address,
- Requests of speculative messages by page granularity,
- Round-Robin method to choose the Home Node (HN).

The CoCCA protocol augments the baseline protocol with a dedicated protocol for managing memory access patterns. Both protocols have an important actor in message management: the Home Node (HN). Each core of the system is the Home Node of a fraction of the cached data, and the coherence information of these data are kept within an extra storage named “Coherence Directory”.

When a processor accesses a data element, it needs to check the coherence state by asking to the corresponding HN. This task is handled by the coherence engine which manages all messages related to shared memory accesses. Therefore, the

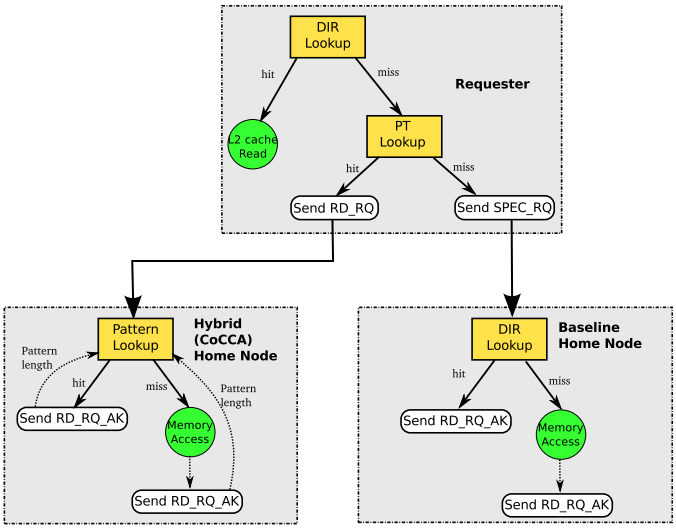


Fig. 4. Model of Transaction Messages including the Home Nodes

initial step is to determine which core in the system is the HN of the requested data. The basic, and classical, algorithm is an allocation of HNs in a round-robin way. Round-Robin is performed by a modulo operation on the low order bits of the address of the data element.

One key question is the granularity used for the round robin algorithm. For instance, it has be shown that memory accesses are not distributed in a homogeneous way, leading to an uneven bandwidth consumption (some cores becomes hot-spots because they are solicited often).

To reduce the hotspot issue of cache systems, the throughput of messages, and to limit the number of messages of cache coherence protocol, we chose a different granularity to determine the HN of the Baseline protocol and of the CoCCA (pattern specific) protocol: we use the line granularity and page granularity, respectively.

### *E. Transaction Message Model*

Figure 4 shows the read transaction message model in the coherency hybrid-protocol that describes the key rules: requester, hybrid (CoCCA) HN, baseline HN. 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 HN. Otherwise, the baseline message is sent to the baseline HN (note that table lookups can be done in parallel).

For the hybrid protocol, the rules can be divided in: core that request the data (requester), the baseline HN and the hybrid HN. The baseline HN is the core appointed by fine granularity (line) for the Round-Robin attribution, and the hybrid HN uses the coarse granularity (page) for the Round-Robin attribution. 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 the top box of figure 4. 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 Hybrid HN (determined by page granularity).

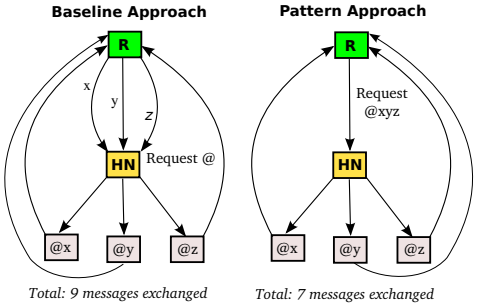


Fig. 5. Baseline and Pattern Approach comparison for a sequence of read requests

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 5).

In case of a cache miss, the flow of message transaction defined by the baseline protocol is sent by requested addresses. When the pattern table and the speculative (CoCCA) protocol 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 5, 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.

## IV. Code Instrumentation and First Evaluation

### *A. Choice of a first benchmark program*

Our goal is to provide a first evaluation of the performance of our hybrid coherence protocol over the baseline protocol alone (but it is worth noting that the CoCCA hybrid protocol relies on the baseline protocol for the messages outside of the prefetch mechanisms).

Therefore we need a benchmark program that would be representative of algorithms found in the embedded world, easy to parallelize, and that shows an interesting variety of behaviors with regards to cache coherence and prefetch. We decided to use a cascading convolution filter: it is very typical of image processing or preprocessing, make a good reuse of data, and is easy to parallelize. The cascading part of the convolution filter use the destination image of one filter as the source of the new filtering, the old source image becoming

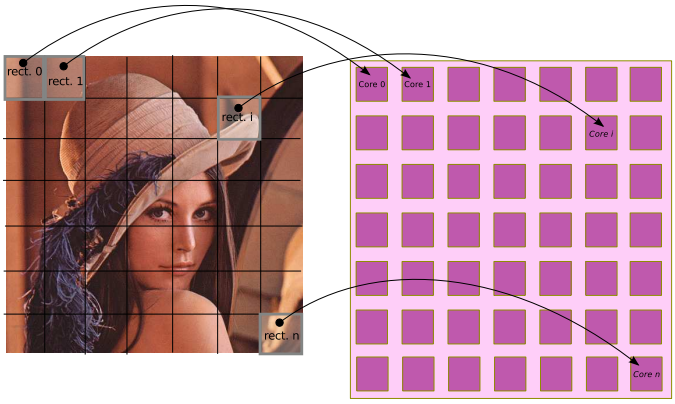


Fig. 6. Scheme of affectation of source image parts to each core in our benchmark program

the new destination image, triggering a lot of invalidation messages in the baseline protocol.

The choice done was to process the algorithm by dividing the source and images in nearly equal rectangles (little variations in rectangle sizes are due to the uneven division in integers) as seen in Figure 6. Source and destination images have a resolution of $640 \times 480$ and the underlying CMP architecture is chosen as a $7 \times 7$ processor matrix, each with 256KB of L2 cache (64B by line of L2 cache). Images are defined as a set of pixel and each pixel is composed of 3 floating point values (32 bits). Both the source and the destination parts of the image managed by a given core can fit in its L2 cache. This is not the best possible implementation of a cascading filter, but this application can show lots of different behaviors regarding caches and consistency.

### B. Instrumentation

In order to make a first evaluation of the hybrid protocol, we need to extract shared data read and write for each core, for this program. We decided to use to use the Pin/Pintools [8] software suite to that end. Pin is a framework that performs runtime binary instrumentation of programs and provides a wide variety of program analysis tools, called pintools. It uses JIT techniques to speed up instrumentation of the analyzed program. A lot of different pintools exists from the simplest (like "*inscount*") to very elaborate ones.

Let us give an example of the use of Pin, using the Simple Instruction Count (instruction instrumentation); this *inscount* pintools instruments a program to count the number of instructions executed. Here is how to run a program and display its output:

```
~/pin-2.10> ./pin -t $(PIN_OBJ_PATH)/inscount.so --
    ./cascading-convo-single-proc
~/pin-2.10/test> cat inscount.out
Count 450874769
```

This is the number of sequential instructions executed when running a mono threaded version of our program. There exists a multi-thread version of this pintool. When the multi-threaded convolution is used, we can obtain a number of instructions executed per core:

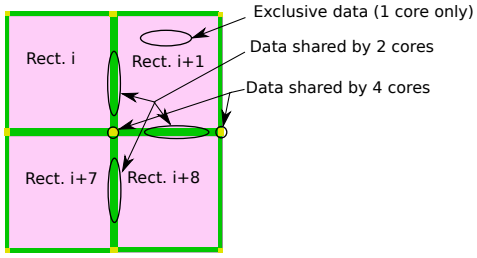


Fig. 7. Read data sharing in conterminous rectangles

```
Number of threads ever exist = 50
Count[0]= 238062
Count[1]= 9064522
Count[2]= 9087339
...
```

An interesting pintool is *pinatrace* which is a memory reference trace (instruction instrumentation): this tool generates a trace of all memory addresses referenced by a program. We modified it to provide also the core Id on which a given memory access is done. It generates the *pinatrace.out* file:

```
0x401c5e: R   0xa0aecc  4      12
0x401d5d: W   0x6832b4  4      12
0x40119c: R   0xa02c20  4      9
0x4011c8: R   0xa02c2c  4      9
...
```

The indications of this file are, in order: the Load/Store instruction address, the Read (R) or Write (W) status, the memory access address, the size of the access in bytes, and the core Id number of the CMP architecture (the execution cores are numbered from 1 to 49 in the output). With this modification of the *pinatrace* pintool, we filtered the accesses to the shared memory accesses. The trace file has nearly 48 millions accesses for a single execution.

### C. Approach to patterns

We can define three kinds of patterns on this benchmark:

- Source image prefetch and setting of old Shared values (S) to Exclusive values (E) when the source image becomes the destination (2 patterns per core),
- False concurrency of write accesses between two rectangles of the destination image. This happens because the frontiers is not alined with L2 cache lines. The associated patterns is 6 vertical lines with 0 bytes in common[2],
- Shared read data (because convolution kernels read pixels in conterminous rectangles, see figure 7). There are 6 vertical lines and 3 sets of two horizontal lines for these patterns.

As can be seen a few set of simple patterns are enough to cover all the coherence data for our benchmark program. Number of patterns is limited to 6 patterns for each core to handle all the coherency issues. This is a tiny number, showing that our approach is sustainable without having too much of impact on chip size (we can imagine to keep the pattern tables

[2]hence, the CoCCA pattern has the information that this is a false concurrency, and that the synchronization can be serialized.

and associated components for managing pattern table lookup and the enhanced protocol at half the size of the coherence directory, by storing only the most relevant patterns).

### D. First evaluation of the protocol

The hypotheses we rely on at this point, is that the pattern tables are a given result of the compilation process, either by code instrumentation, as above, or by static analysis. They are statistically attributed for a given part of an application, and reloaded as required. This is a valid hypothesis in the embedded world where static generation is often standard because the system is tuned to a limited set of applications that are highly optimized. For pure HPC systems, an automatic dynamic generation of patterns would be preferable, but this is still future work.

In the periodic execution of our program, once initialization is passed, we have the following trend of message for the pure MESI versus the hybrid protocols:

| Condition | MESI | CoCCA |
|---|---|---|
| Shared line invalidation | 34560 | 17283 |
| Exclusive line sharing (2 cores) | 12768 | 12768 |
| Exclusive line sharing (4 cores) | 1344 | 772 |
| Total throughput | 48672 | 30723 |

Hence, there is a reduction of over 37% of message throughput. This does not includes the advantages of data prefetch which reduce in a large way the memory access latency. On this example, prefetch stands for about 10% of the on-chip cache sharing and nearly all main memory accesses, minus the first ones corresponding to the first access of a given pattern.

On an Intel Xeon Nehalem a single task run in a bit less than $4490.10^3$ cycles on a core, with preloaded caches (no misses, 37128 write accesses and 928200 read accesses in shared memory). This is the expected speed when the CoCCA protocol is used, since, in this case, caches are efficiently prefetched. With the baseline protocol alone, all the write accesses trigger a memory access, 17283 read misses with memory also appear and about 13000 cache sharing requests. When using 80 cycles to access main memory and a mean of 20 cycles to access on chip L2 shared data, this gives a total overhead of $3.10^6$ cycles or 67% slower. For this application, using a speculative protocol like CoCCA is a huge performance boost.

## V. Conclusion

With the growing scale of chip multi-processors, data cache consistency becomes one of the key challenge to efficiently support parallel applications. This is also true for embedded systems: a large number of embedded applications read and write data, according to regular memory access patterns. These patterns can be used to optimize cache coherence protocols and therefore, to improve application performances when sharing data among cores.

In this paper, we proposed a system architecture of Cache Coherency Architecture that make such use of memory access patterns. A regular consistency protocol has been enhanced to handle speculative requests and a new hardware component has been designed to store and retrieve patterns. We described a new hardware-component with an auxiliary memory unit that composes the cache hierarchy to implement that.

We provided a first evaluation of the benefits of our enhanced Cache Coherency Architecture. Basically, we generated the memory access traces for a benchmark program and showed the easiness of handling patterns: only a few patterns per core are sufficient to handle all the coherency traffic with the speculative protocol. On our benchmark, the evaluation shows a performance boost of over 60% thank to the reduced access time to data (prefetched L2 caches). We showed also an optimized throughput of messages by over 35%.

As future work, we want to use an analytical model of cache coherency protocol that would permit to evaluate the effective cost of our protocol performance with regards to the standalone baseline protocol with more accuracy, and use a real simulator platform like SoClib for that. We want also to extend our protocol toward HPC friendly systems, with a dynamic (online, or at runtime) generation of pattern tables.

## References


[1] Li, K., Hudak, P.: Memory Coherence in Shared Virtual Memory Systems. ACM Transactions Computer Systems 7, 4, 321-359 (1989).

[2] Debes, E., Chen, Y-K., J.Holliman, M., M.Yeung, M.: Apparatus and Method for Performing Data Access in Accordance with Memory Access Patterns. Intel Corporation, Patent US 7,143,264 B2 (2006).

[3] Shen, X., Shafi, H.: Mechanisms and Methods for Using Data Access Patterns. International Business Machines Corporation, Patent US 7,395,407 B2 (2008).

[4] Jeffery, A., Kumar, R., Tullsen, D.: Proximity-aware Directory-based Coherence for Multicore Processor Archectures. ACM Symposium on Parallel Algorithms and Architectures SPAA'07, pp. 126-134, June 9-11, San Diego, California (2007).

[5] Zhuo H., Xudong S., Ye X., Jih-Kwon P.: Alternative Home Node: Balacing Distributed CMP Coherence Directory CMP-MSI: 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects, Beijing, China (2008).

[6] Chung, E., Hoe, J., Falsafi, B.: ProtoFlex: Co-simulation for Component-wise FPGA Emulator Development. 2nd Workshop on Architecture Research using FPGA Platforms - WARFP (2006).

[7] H.J.Hum, H., R.Goodman, J.: Forward State for Use in Cache Coherency in a Multiprocessor System. Intel Corporation, Patent US 6,922,756 B2 (2005).

[8] Moshe Bach et al.: Analyzing Parallel Programs with Pin. IEEE Computer Society 0018-9162 (2010).

[9] Tera-scale Multicore processor architecture (TSAR). Project Profile Medea+, European Project 2008-2013.

[10] Beamer, S.: Section 1: Introduction to Simics. CS152 Spring 2011, Berkeley University.

[11] K. Rupnow, J. Adriaens, W. Fu, and K. Compton. Accurately Evaluating Application Performance in Simulated Hybrid Multi-Tasking Systems. Accepted for publication at ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2010.

[12] Xudong S., Zhen Y., Jih-Kwon P., Lu P., Yen-Kuang C., Lee V., Liang B.: Coterminous locality and coterminous group data prefetching on chip-multiprocessors. In: 20th International Parallel and Distributed Processing Symposium - IPDPS'06,IEEE (2006)

[13] Stephen S., Thomas F., Anastasia A., Babak F.: Spatio-Temporal Memory Streaming. In: 36th International Symposium on Computer Architecture - ISCA'07, pp. 69-80. ACM (2009)

[14] Stephen S., Thomas F., Anastasia A., Babak F.: Spatial Memory Streaming. In: 33th International Symposium on Computer Architecture - ISCA'06, pp. 252-263. IEEE (2006)

[15] Thomas F., Stephen S., Nikolaos H., Jangwoo K., Anastasia A., Babak F.: Temporal Streaming of Shared Memory. In: 32th International Symposium on Computer Architecture - ISCA'06, pp. 222-233. IEEE (2005)