UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Universitat Politècnica de València

Department of Systems Data Processing and Computers

Parallel Architectures Group

Analysis of opportunities for cache coherence in heterogeneous embedded systems

A dissertation submitted in partial fulfillment of the requirements for the degree of

Master in Computer Engineering

Author

Albert Esteve Garcia

Advisors

Dr. Antonio Robles Martínez

Dr. Maria Engracia Gómez Requena

September, 2012

Acknowledgments

I would like to first and foremost thank my girlfriend Carla, for her patience and support during this time. Also to my mother Lola and my brother Victor for always being there when needed, for encouraging and supporting me, particularly my brother, who hooked me to biking, which has been my safety valve whenever I needed one.

I should also thank my advisors, Dr. Antonio Robles and Dr. Maria Engracia Gomez, for guiding and correcting me. They had an infinite patience with me, being always accessible.

Dr. Jose Flich also helped and guided me whenever I required it from him. I'm thankfull for his concern and consideration on my progress. His sense of humor and wisdom are inspirational.

It would have been impossible to do this dissertation on time without the help of Maria Soler, colleague and friend, who has always made efforts and given advice since her arrival to the group.

Last but not least, I thank my colleagues in the Parallel Architectures Group, who have made the time I spent with them something nice and relaxed. I hope to share many other cakes and meals together in the Osaka. Especially Ricardo Marin, which was unfortunate to sit beside me, but he have standed paciently, resolving all the technical problems I've had, no matter how simple they were.

Analysis of opportunities for cache coherence in heterogeneous embedded systems

Albert Esteve Garcia

Department of Systems Data Processing and Computers
Universitat Politècnica de València
2012

ABSTRACT

Embedded devices are becoming more and more present everywhere. Moreover mobile devices are becoming also more computationally powerful. These embedded architectures present new challenges since they execute several applications that must preserve security, allow sharing information in a coherent way, to be scalable and provide the required levels of performance, while at the same time they must be power-efficient.

In this context, the vIrtical project focuses on extending the virtualization concept to the embedded domain. Virtualization, widely used in the general-purpose computing domain, allows an effective and clean way to isolate applications from hardware, so being suitable to cope with the challenges faced by heterogeneous multi-core embedded systems. However, virtualization on embedded systems is still in its infancy.

To achieve the development of the virtualization concept, software/hardware extensions should be delivered at different layers of the design stack. As a part of this, in this work we have analyzed memory sharing patterns from industrial embedded applications in order to exploit them to make the coherence protocols more scalable and power-efficient. Nowadays the coherence protocols do not cope with the needs of embedded systems imposed by both their architecture and the supported industrial embedded applications. They introduce overheads in terms of coherence traffic and storage resources required increasing both execution time and power consumption. In addition, current protocols do not scale due to resource overheads and indirection when accessing data.

The study is made having in mind the latest proposals on coherence optimizations, and related previous studies based on block classification and coherence deactivation.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

This chapter first briefly explains the context which this dissertation is settled in and their motivations in Section 1.1. Then the coherence problem is identified in Section 1.2. Finally, the chapter concludes detailing the structure of this dissertation in Section 1.3

1.1 Context and Motivation

During the last decade embedded devices have invaded our everyday life, to the degree that it is becoming hard to imagine living without them. Embedded systems are currently present at home (set-top boxes, smartphones, TV set), at work (smartphones, tablets), even when we travel (in-car and in-flight entertainment). Nowadays this technology allows everyone to be *connected* nearly everywhere, by several possible means (voice, text, and video).

With the recent advances in wireless networks and the exponential growth in the usage of multimedia applications, multi-core platforms point to be the solution of feature-rich phones to deliver the performance comparable to today's computer system with the hugely increased complexity of also requiring real-time and protected execution environments to maintain their embedded functionality.

The availability of powerful heterogeneous multi-core processors has led to an unforeseen escalation of software complexity as device manufacturers are integrating multimedia services, data networking, photography, telephony and personal information management into their embedded products.

The software stack running on contemporary smart-phones is already 5±7Mloc (million lines of code), and growing. Top-of-the-line cars contain literally gigabytes of software. Increasingly, embedded systems run applications originally developed for the PC world (web browsers, games).

Fig. 1.1: Examples of different embedded systems present nowadays

Needless to say, the ever-increasing complexity of such devices becomes a limiting factor: being able to timely deliver top feature-rich devices poses a renewed challenge over system-on-chip (SoC) designers. To be competitive, new communication, consumer, and computer product designs must exhibit rapid increases in functionality, reliability, and bandwidth at sustainable cost and power consumption. There is intense pressure on chip designers to develop increasingly complex hardware in decreasing amounts of time.

The vIrtical project aims to tackle these challenges by extending the virtualization concept of the general-purpose domain to the embedded domain. Virtualization techniques have become popular among high-performance servers for several reasons, among which include reduction of expenses in terms of cost and simplification of the administration that provides the server consolidation (running a large number of virtual servers on a small number of real server). In addition, virtualization is revealed as a fundamental mechanism for the efficient use of the increasing number of cores that are integrated into a CMP.

However, the application of virtualization its still on his infancy on embedded systems. The challenge is the virtualization of embedded systems that requires particular approaches meeting tight resource budget and considering their particularities.

Such systems are currently configured on a SoC with multiple cores, which also integrate other elements such as GPUs, hardware accelerators, I/O devices, etc. In which the heterogeneity of its elements is one of its main features, as can be seen in Figure 1.2.

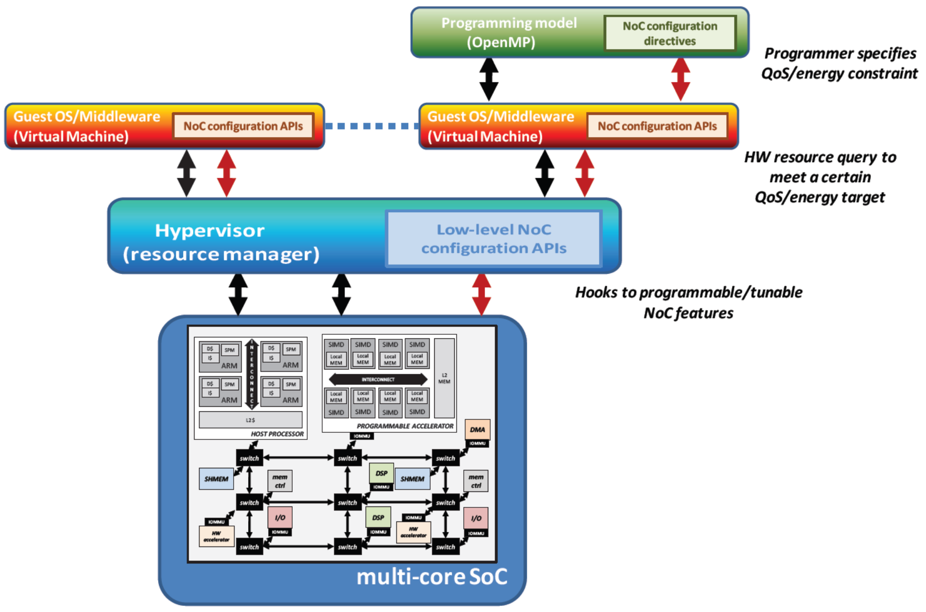

Programming model (OpenMP)
NoC configuration directives
Programmer specifies QoS/energy constraint
Guest OS/Middleware (Virtual Machine)
NoC configuration APIs
Guest OS/Middleware (Virtual Machine)
NoC configuration APIs
HW resource query to meet a certain QoS/energy target
Hypervisor (resource manager)
Low-level NoC configuration APIs
Hooks to programmable/tunable NoC features
HOST PROCESSOR
PROGRAMMABLE ACCELERATOR
multi-core SoC

Fig. 1.2: Embedded Heterogeneous System

As part of the vIrtical project, and taking into account the many extensions that should be delivered to the design stack to support virtualization, this dissertation focuses on making a study of the suitability of the many techniques present on the general-purpose domain to deal with the coherence problem in the embedded domain.

In this context, the cache coherency protocol, which manages communication between the cores of the CMP, is one of the key difficulties due to its scalability. In this sense, the design should try to minimize the overhead of the coherence protocol both in terms of storage resources and required traffic through the interconnection network. At the same time, virtualization servers pose another problem for the efficiency and scalability (current protocols do not scale due to resource overheads and indirection when accessing data) of coherence protocols, so that the design of the latter should be flexible enough to adapt effectively to a configuration-variable environment system as the derivative of virtualization.

The study is made with two main approaches in mind. One of the key approaches for this study is try to avoid the tracking of non-coherent memory blocks, which means that the block is either not shared or never written and, therefore, cannot suffer from inconsistencies. These kind of techniques reduce the network overload due to coherence messages, execution time and energy consumption; making them a good candidate for the needs of the aforementioned embedded systems. Another good candidate consists on deactivate the coherence when not needed as showed in Section 2.2.3.

1.2 The cache coherence problem

Chip MultiProcessors' (CMPs) cores logically share address space. Also, cores usually include at least one level of cache, which is private to each one, in order to deal with the increasing gap between processor and memory speeds (the memory wall problem). Memory caches are smaller and, therefore, faster. In this sense, in a memory hierarchy compound of various (more than one) levels, the closer to the core the memory is, the smaller and faster it is.

On the scenario in Figure 1.3 we have a shared memory M connected to a system consisting in many Processors P_n each one with its private cache C_n. If many cores (1 and 2) read the same memory block on their respective private caches in order to take advantage of locallity of data there is no problem as long as they keep it read-only. However, if a core modifies the shared memory block (3) without other cores being aware of it, this may result in different private caches containing inconsistent values for the same memory block (4). This is the cache coherence problem.

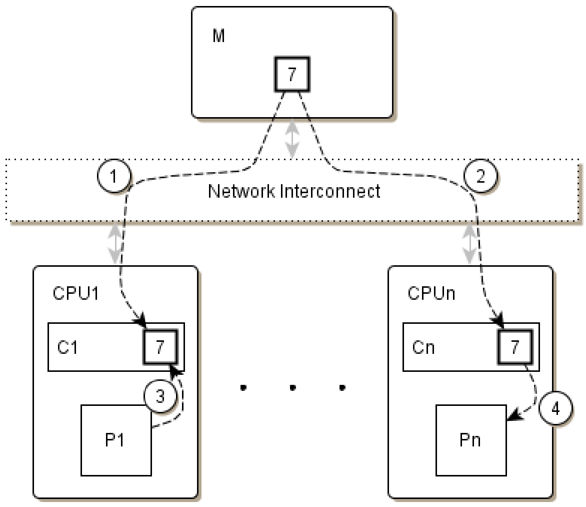


Fig. 1.3: The cache coherence problem

Approaches for solving the cache problem fall into two major classes: hardware-based and software-based approaches. The first one guarantee coherence by adding specific hardware capable of doing so. In contrast, the second one restrain caching of shared data to when is safe to do it, either done by the programmer, the compiler or the operating system.

Hardware-based cache coherence solutions can also be classified according to two dimensions [Ste90]:

1. *The type of interconnect*: when processors are connected through a shared medium (i.e. a bus), protocols may use broadcasting to enforce coherence. These protocols

are called *snoopy protocols*, and they apply to small-size bus-based systems because of their scalability shortcomings. In case no shared medium is present (the total order of messages cannot be guaranted), snoopy protocols are usually replaced by *directory-based protocols*.

2. *The type of cache-coherence policy*: in this category we find mainly two schemes: a *write-invalidation* and a *write-update* policy. In the former whenever a processor writes into a shared memory block B it invalidates all the copies of the block B present in other caches. On the other hand, in the latter one the caches are updated with the new value of B.

   Invalidation-based schemes are more common due to the fact that they are easier to implement and also more efficient for larger cache lines size than update-based schemes. Notice, however, that update-based schemes are more efficient when accessing heavily contended lines, since subsequent cache accesses result in a hit due to the update policy.

1.3 Thesis Outline

The organization of the reminder of this thesis is as follows:

- Chapter 2 presents a better insight to current solutions to the coherence problem. It is mandatory to understand the approaches being trend at the present time in order to see the suitability of those to the proposed system.
- Chapter 3 describes the target system of the study and the coherence needs of each element on it.
- In Chapter 4 the analysis of the applications is presented. Firstly the simulation

tools used to perform the analysis are presented. Secondly, the methodology followed to accomplish the acquisition of data is detailed, followed by the description of the application used to get the data. Then, the analysis itself is presented.

- Finally, Chapter 5 gives a discussion of the results obtained, the suitability of the many techniques to the coherence problem into the target system proposed and the future directions to be explored.

CHAPTER 2

The Coherence Problem and Solutions

This chapter gives some background to the coherence problem, detailing the cache coherence models (or policies) present in the literature.

The chapter also describes several of the many techniques used nowadays, which can be found in the literature, to deal with the coherence problem. This kind of study must be made in order to properly analyze the suitability of the techniques to improve the coherence protocol of vertical target system.

First of all, we must emphasize with the classification seen in Section 1.2:

- *Write-Invalidate*: in which it is assumed that the only valid copy of the data is the one that has changed (written) it for the last time. In a write-through policy another copy of the data is also found in the main memory.
- *Write-Update*: in which every modification of the data is communicated to every sharer of it. A valid copy of the block can be found on every sharer no matter who was the last writer.

Also it must note that this Section supposes a simple system with only one level of cache. When we refer to ”main memory” it can be understood as the immediately lower level in the cache hierarchy.

2.1 Cache Coherence Models

In all the hardware-based systems with several copies of the same block, a proper coherence protocol must exist in order to perform the required actions atomically.

In a generic cache coherence protocol every block present in the cache hierarchy has a state associated with it (along with tag and data), which indicates its arrangement. The existing states and the way the block is classified among them is described in the cache coherence model. It can be usually seen as a finite state machine specifying how the state of a block changes. Despite the fact that only cached blocks have a state, the blocks only in main memory can be seen as having a special Invalid state.

While in a uniprocessor system, for write-through write-no-allocate caches, only two states are needed (Valid or Invalid); in a multiprocessor system as the target system of this thesis, we can find a set of n states[1], each one manipulated by the finite state machine implemented on the cache coherence controllers on each node. The finite state machine is the same for every block and every cache, but the actual state of a block differs for different caches.

The most common states for cache coherence models are:

- **M (Modified)**: a cache block in this state holds the only valid copy of data. The core has read and write permissions over the block. The copy of the block found on main memory is stale. If another core requests the block, the cache with the block in the modified state must provide it.
- **O (Owned)**: a cache block in this state must provide the data if another core requests it. In this case the block can coexist with another blocks in the Shared

[1]Depending on the alternative chosen, namely MSI, MOSI, MESI, MOESI, etc. The names are accord with the states of the finite state machine associated

state. The core holding the block in the Owned state has just read permission over it.

- **E (Exclusive)**: a cache block in the exclusive state holds a valid copy of the data with read and write permissions over it. In this case the state does not imply ownership, so the core does not need to supply the block in the case another core requests it. Exclusive state can be seen as an intermediate state between shared and modified.

- **S (Shared)**: a cache block in this state has a valid copy of the data with read permission over it. Other cores can also hold the block on the shared state and one of them may have it in the owned state. If no owner block is present the main memory must provide it in the case that another core requests it.

- **I (Invalid)**: a cache block in this state does not hold a valid copy of the block.

As observed in table 2.1, only the M and E states imply exclusiveness on the block and, therefore, write permission. On the other hand, M and O states imply ownership, which means that cores must supply it in case of another core requesting it. Finally, all states imply validity of block, which means that they have at least read permission over it, except for the Invalid state.

	Property		
State	Exclusiveness	Ownership	Validity
M	$\surd$	$\surd$	$\surd$
O	X	$\surd$	$\surd$
E	$\surd$	X	$\surd$
S	X	X	$\surd$
I	X	X	X

Table 2.1: Properties of blocks regarding to their cache state

2.1.1 MSI

The MSI protocol supposes the minimum set of states to ensure the cache coherence protocol to work properly for invalidation-based write-back private caches.

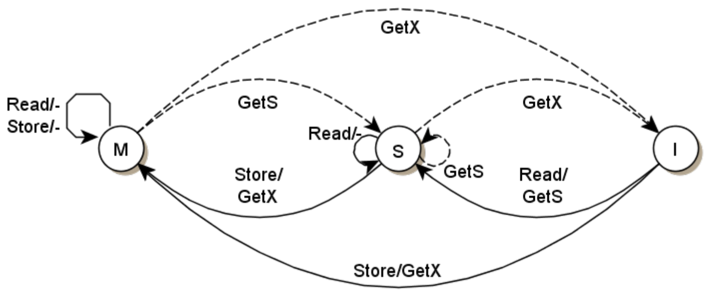


Fig. 2.1: Finite State Machine for a MSI protocol

In the Figure 2.1 the finite state machine for the MSI protocol can be observed. In the scheme dashed lines represent request from other cores and bold lines represent read and stores issued by the home core. Furthermore, the label for the transitions is in the form R/A, in which R represents the request that originated the transition and A represent the action made by the cache controller (coherence traffic generated).

In the MSI protocol a block can be obtained either in modified state, if it requires read/write permission, or in shared state, if the block is present in other private caches and does not require write permission.

2.1.2 MESI

Adding the E state optimizes the MSI protocol for non-shared data. It allows the block to be cached as exclussive if cached for the first time, avoiding generating a transition for subsequent writes. The transition diagram can be observed in Figure 2.2.

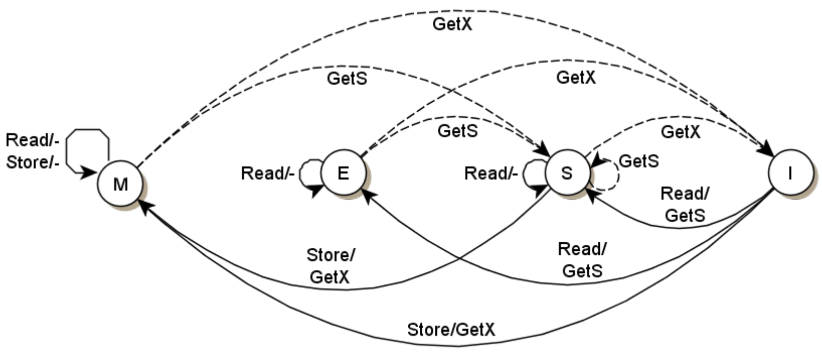


Fig. 2.2: Finite State Machine for a MESI protocol

2.1.3 MOSI

In the case of the MOSI protocol the new state (owned) supposes an optimization for the MSI protocol focused on allowing other caches to obtain the block from the private cache containing it on the owned state (only one at a time), due to the fact that it is less time consuming. Also the owned state does not enforce to maintain the block updated on main memory. In Figure 2.3 can be seen that the block in the owned state must change either to modified if it requires write permission, invalidating other sharers holding the block; or to shared if it updates the block on main memory.

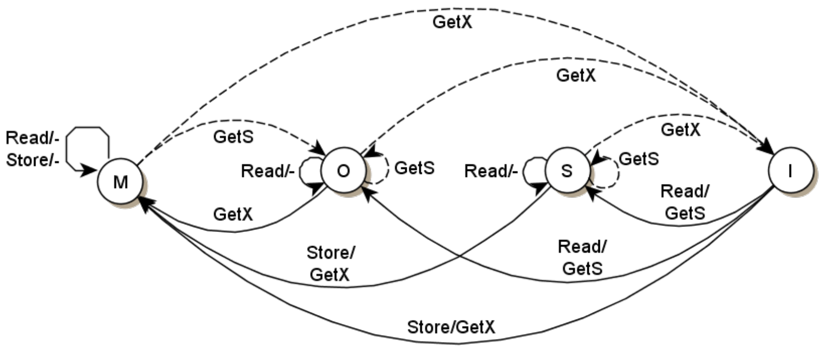


Fig. 2.3: Finite State Machine for a MOSI protocol

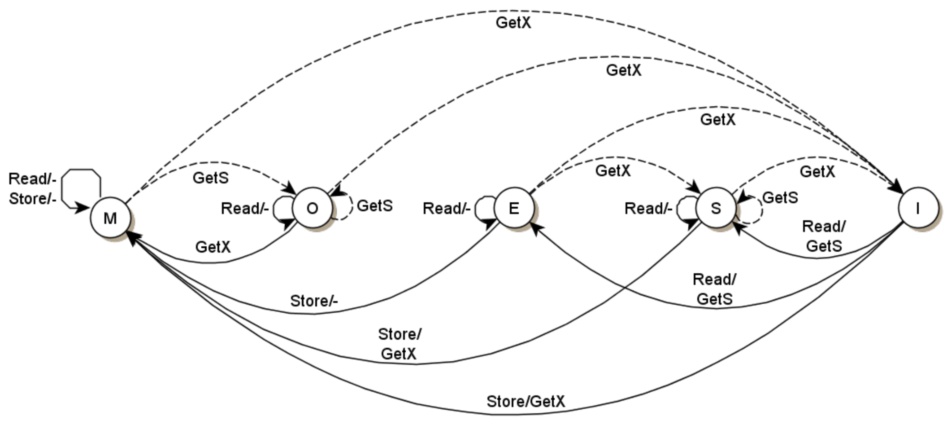


Fig. 2.4: Finite State Machine for a MOESI protocol

2.1.4 MOESI

Figure 2.4 shows the state transition diagram for the MOESI protocol, which is the last coherence model considered in this dissertation. It includes both optimizations for the MSI protocol considered in the previous sections.

2.2 Hardware-based Cache Coherence Protocols

Hardware-based coherence protocols allow developing software without taking into account the cache coherence problem. In general two main coarse grain architectural solutions can be found in the literature for hardware-based caching: *snoopy-based* and *directory-based* techniques. This Section appends some approach types to the classification. Especially one which appears to be a good candidate for the target system of the thesis: *cache deactivation* technique. Nevertheless, cache deactivation can be seen as an optimization of either previous approaches.

2.2.1 Snoop Based Protocols

As mentioned in Section 1.2, snoopy protocol are based in broadcast the request in order to enforce coherence. In this case, cache controllers ”snoop” on the bus and monitor all other cache transactions. Once detected a transaction, the cache controller must take the most appropriate action, which may include generating bus transactions to access memory. In order to ensure this kind of techniques it must be assured two key properties for the bus:

1. All cache transactions must be visible for all the rest of cache controllers.
2. The network must maintain the total order of transactions.

In these systems the lower cache hierarchy levels dissipate the larger amount of power. In a typical write-invalidate protocol, all bus-side cache controllers ”snoop” the bus upon a request, increasing the access to lower cache levels.

Snoops in this snoop-based protocol typically fail to find the block originated by the request, wasting the energy consumed to do so. Using the analytical model used in [KG97] snoop miss tag accesses can be estimated and suppose about 33% of all consumed by L2 caches for a 4-way SMP.

In [MMFC00], a family of energy-efficient structures called JETTY, which are capable of filtering snoop traffic and reduce energy consumption in all lower-level caches, are proposed. One set of JETTY resides between the processor and the memory-bus interface on every node of the network. When a request is received, JETTY first addresses this request either by guaranteeing that no copies exist or responding that copies may exist, therefore requiring a subsequent snoop directly to the cache hierarchy. In this case, JETTY relies on the following requirements to be successfull:

1. The majority of snoop-induced L2 accesses should result in a miss (which is, fortunately, the most common case).

2. It should be possible to identify most of these misses using a small structure.

3. We should never report a would-be miss while the data is locally cached.

Using JETTY, nodes maintain two structures that respectively represent a subset of blocks that are not cached (exclusive JETTY) and a superset of blocks that are cached (inclusive JETTY).

Moshovos et. al. demonstrate that a very small JETTY succeeds in filtering 74% of all snoop-induced tag accesses that would miss, in average. These results in an average energy reduction of 29% measured as a fraction of the energy required by all L2 accesses (both tag and data arrays).

There are many similar, more recent optimizations for snoopy protocols based in JETTY. Andreas Moshovos propose RegionScout [Mos05], a family of simple filter mechanisms based in JETTY that dynamically detect non-shared regions and provide nodes with the capability of determine in advance that a request will miss in all remote nodes. This capability allows reducing coherence traffic due to not probe any other node, and therefore, reducing the energy dissipated, bandwidth requirements and the latency of the corresponding memory requests. However, RegionScout filters utilize imprecise information about the regions that are cached in each node (using the Cached Region Cash (CRH) structure), which leads to a loss of coverage produced by not being capable of detect all requests that would miss in all other nodes.

Another more recent approach to snoop filtering is the Subspace Snooping [KAKH10] from D. Kim et. al., which uses an OS-based mechanism to maintain subspaces at page granularity. The set of sharers for a page is recorder on the OS page table entry and translation look-aside buffers (TLBs). Thus, in a coherence request, messages are delivered only to the nodes on the subspace. The main contribution of this approach is that it does not add significant hardware complexity and is, therefore, adaptable to many existing coherence techniques.

A similar technique intended to embedded systems is the one devised by X. Zhou et. al. [YZP09]. In embedded systems, where compiler, system software and hardware are fine tuned in their functionality and interaction with specific application requirements, this proposed technique takes advantage of these characteristics to achieve significant performance and power improvements through precisely identifying the shared memory regions for each task, and then providing this information to the operating system and cache snoop controller for runtime utilization. The information about shared memory regions must be provided by the compiler or software developer.

There are a number of proposals based also in snoop filtering but using coarse-grain memory regions tracking. In Coarse Grain Coherence Tracking (CGCT)[CLS05] each processor in the multiprocessor structure maintains a special structure (namely Region Coherence Array (RCA)) for monitoring coherence at a granularity encompassing a power-of-two conventional cache lines. On snoop requests, each processor's RCA is snooped along with the cache line state and the coarse-grain state is piggybacked onto the conventional snoop response and stored on the local RCA to avoid subsequent broadcasts for line on the coarse-grain memory region. In addition to the non-shared state, the RCA also tracks the shared read-only data to further optimization.

Also, in [ZSM07], RegionTracker (RT) is presented. RT introduces region-level functionality without compromising performance or area. With this technique communication still uses fine-grain blocks, but RT tracks if a block is cached and where. RT replaces the tag array from any conventional coherence protocol with a structure that facilitates region-level lookups and management. In the end, it results in a cache design that does not require additional area, nor higher associativity and it does not hurt performance, latency or complexity.

Another coarse-grain snoop filtering technique is the one found in [PG08]. It assumes that a page is divided in a number of consecutive regions and maintains coarse-grain sharing information for these regions in a set-associative structure called Snoop Filter Table (SFT). The usual coherence actions and the SFT updates are only performed if a SFT entry address match is found. In this way the sharing information is collected proactively and up to 90% of unnecesary snoop requests are filtered.

2.2.2 Directory Based Protocols

In highly parallel systems, interconnection structures that allow greater scalability are used. which makes the snoopy-based protocols unusable. This kind of architecture, typically a NUMA organization for a tiled architecture (CC-NUMA when it is Cache Coherent), needs a more scalable coherence protocol. Scalability concept is commonly based in directory usage. Directory is a data structure that tracks for each block a record of the block state and the actual sharers of that block. That record is known as directory entry. When a node misses a block on its private cache it first communicates with the home directory for that block in order to find the availability of it. Any modification on the block state (e.g. a write request) must be notified to the home directory. Every transaction may take further communication in order to ensure consistency for the block. Directory is also responsible of invalidating cache blocks if needed.

Directory-based mechanisms are independent of the interconnection used. Cache coherence protocol used can be either invalidation-based or update-based or hybrid, and the cache model can also have any number of states.

One of the main problems exhibited by directory-based cache coherence protocols is regarded to the storage resources required by the directories, which are increasingly greater as the number of nodes increases. In CMPs, where area and power constraints are a critical design issue, the use of directory-based cache coherence protocols compels us to reduce at most the silicon area required by directories. A lot of works have addressed this problem from different approaches. Most of them are focused on reducing either the number of entries or the entry size of the directory while maintaining system performance. Others even propose novel directory organizations with lower area requirements.

As known, Duplicated tag directories, where each block sharer is allocated to a different entry, guarantee enough space to track all the possible cached blocks at the expense of a high associativity (equal to the product of the cache associativity and the number of caches). Despite the small area required, the quadratic growth of the energy consumption because of their high associativity, precludes their practical applications to large CMPs. On the other hand, Sparse directories reduce associativity (much lower than the aggregate associativity of the caches they track) at the cost of extending each directory entry with some kind of sharing information and increasing to some extent the number of sets. However, they no longer guarantee that whatever combination of memory blocks can be simultaneously tracked. As a result, allocating a new directory entry may require evicting a current entry and invalidating all cached copies of the corresponding block. Performance penalization derived from forced invalidations can be mitigated at expense of increasing the directory size (at least, a coverage ratio between the number of cache entries and the number of directory entries equal to one is suggested). In order to invalidate cached copies, the sharing information is used. Such information may be either precise or imprecise. In the former case, the directory uses a full-mapped bit vector at each entry to track the block sharers. However, this approach is not scalable because the aggregated area required by the directories experiments a quadratic increase as the number of cores increases. On the contrary, in the latter case, a compressed representation of the sharing information is used, allowing more compact directory entry formats and reducing storage area requirements. In this sense, there exist many proposals aimed to shorten the sharing information, such as, for example, the use of a limited number of pointers, segment directories, chained pointers, and the use of coarse vectors among others.

On the other hand, some proposals choose to reduce the number of directory entries by combining several of them into a single one.

As observed, we can find many different approaches to improve the directory-based protocols in the literature, but we selected three different recent approaches.

On the one hand, in [ZSQM09] directory design is revisited combining two main ideas:

1. The set membership test[2] do not have to be precise, an estimate test is sufficient for correctness (despite a more accurate test can be desirable for performance).
2. Bloom filters[3] are space-efficient structures to perform set membership tests.

The proposal is a Tagless Coherence Directory (TL), a scalable directory structure that removes the need of a conventional directory structure and replace it by a grid of Bloom filters, with one column for each CMP core, and one row for each cache set. Each Bloom filter tracks the blocks of one cache set of one core, and accessing to it retrieves a sharing vector that represents a superset of all the sharers of the block. This solution saves area and power overheads compared to conventional sparse directories with no performance loss and little bandwidth increase by replacing energy intensive associative

[2]Maintain coherence consists of perform a set membership test on each cache to determine which ones have copies of a block.

[3]A Bloom filter, conceived by Burton Howard Bloom in 1970, is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. False positive retrieval results are possible, but false negatives are not.

lookup needed by shadow tags with Bloom filter tests. Therefore, Tagless provides good area scalability but it not energy-scalable. Due to the use of imprecise information, if a sharing pattern not represented occurs, the directory commonly resorts to broadcast invalidations at the expense of higher bandwidth utilization.

On the other hand, SPACE (Sharing PAttern-based CoherencE) [ZSD10] proposes an optimization for directory caches based in recognizing the sharing patterns in an application and taking advantage of it whenever many memory locations in an application are accessed by the same set of processor nodes. This is the case where a few sharing patterns on an application occurs frequently. SPACE holds the sharing patterns for each cache line in a separate directory table, thereby having multiple cache lines pointing to the same entry in the directory table. It reduces up to 60% of area overhead in conventional directory at 16 processors.

Finally, Cuckoo [FLKBF11] is a proposal for an area-efficient scalable distributed directory with nearly constant power and area utilization per core, regardless the size of the CMP. It roughly provides the equivalent of a fully-associative directory at the cost of a more complex insertion procedure. In order to do so Cuckoo uses a small-associativity hash table structure whose address bits are passed through different hash functions. Hits on the directory just require one access, but replacements needs many hash functions in order to obtain various candidates, attaining the illusion of a more associative cache at the expense of a greater energy consumption and latency.

2.2.3 Coherence Deactivation

In recent multicore systems, the ever-increasing number of cores increases the on-chip cache size in order to supply cores with data. This comes at the cost of growing access latency. There is also increasing access latency for far away nodes due to wire delay.

However a significant percentage of the memory blocks are only accessed by one core and, therefore, do not require coherence maintenance. Through dynamically classifying memory blocks we can take advantage of this and deactivate coherence and treat memory blocks as an uniprocessor system would do.

Following this trend we find several approaches, as in [PSNB10], where two new cocherence protocols are proposed: SWEL (protocol states are Shared, Written, Exclusivity Level) and RSWEL. The idea is having all private and read-only data in L1, while all shared and written blocks must reside at L2 level. The decision is made at run-time without software assistance. We can observe that directory-based protocols are very inefficient when handling producer-consumer pattern in large multi-core systems because of the indirection introduced by the directory. SWEL, through eliminating the classic invalidation- update-based patterns, eliminates the need of doing such costly mechanisms, improving throughput, and lowering protocol transient states and storage overhead. The penalty is that every mis-classification of the blocks leads to recovery via broadcast, a simple and infrequent action to ensure correctness of the protocol.

In [MS09], J. Meng et. al. develops a study of thrashing contention for inclusive caches. They discovered most private data is non-uniformly distributed among the last level of cache (LLC) sets, with wasteful unnecessary LLC conflicts. To reduce those conflicts among with the data thrashing a new cache organization namely NISC (non-inclusive, semi-coherent) is proposed. It allows private data to exist only in L1 caches and private data evicted from LLC need not to invalidate their copies in L1 caches, thus excluding private blocks from cache coherence.

In addition, in [CRG^{+}11] B. Cuesta et. al. propose a mechanism that classifies memory blocks into private and shared and, subsequently deactivates coherence for blocks classified as private. They rely on the OS in order to classify blocks, making it at page granularity. When a page is classified (Private as for new pages) all the block within the page are classified the same way. Coherence is not maintained for Private block, but the OS is capable of detect when a blocks becomes shared, thus triggering a coherence recovery mechanism. It only requires minor modification of the OS and the memory controllers. It is demonstrated that 57% (in average) of data blocks can be omitted from coherence tracking for directory-based protocols, and invalidations are decreased about 70%.

2.2.4 In-Network Coherence Protocols

As number of cores in multiprocessor architectures grows, we need a scalable coherence protocol to cope with the needs of these systems. However, despite that the classic choice to cope with the scalability constrain would be a directory-based protocol, it comes with some well-known limitations related both to the overhead in communications, to determine which cores have a block cached and invalidate them; and to the storage overhead, especially in area-constrained multi-core chips as in the embedded systems.

These techniques propose an implementation of the cache coherence protocol within the network. As the one described in [EPS06], which embeds directories within each router node, thus moving the protocol into the network, and leading requests traversing the router towards nearby data copies. In the aforementioned approach, it modifies the router architecture adding one extra stage to the conventional router pipeline called *Virtual Tree Cache*. Also sequential consistency is ensured. They evaluated that up to 44.5% save of memory latency overhead on a 16-processor system.

A different approach is [CMR+06], which proposes to adapt the interconnection wires in terms of latency, bandwidth and energy characteristics and map the coherence requests to the approppiate wire depending on their needs. Cheng et. al. shows different proposals to address and exploit this optimization with a write-invalidate directory-based protocol, with a write-invalidate bus-based protocol and with protocol-independent techniques. A 11.2% of performance improvenent and a 22.5% of reduction in interconnection energy consumptions are achieved.

On the other hand, [APJ09b] proposes ordering snoop requests in an unordered network. All the network routers contain pre-assigned *snoop-orders* that are tagged onto requests when they arrive from the attached cores. These snoop-orders are disjoint numbers, and the lower the snoop-order, the earlier the ordering of a request. INSO achieves global ordering of requests by ensuring that cores process requests in the order dictated by the snoop-order. To ensure global order of requests for the same region, INSO assign an snoop-order greater to the new region request than the snoop-orders of requests currently in the network for the same region. To establish this, the region update message is sent, which goes to the entire system and collects the lowest snoop-order present in the routers currently. It is made in a way that ensures that protocol ordering is never violated due to filtering.

Finally, [APJ09a] tackles the ordering of requests in snoopy protocols in unordered networks, but adding a filter that prevents broadcasts produced by the coherence protocol to be sent to nodes not sharing the block, thus saving power and bandwidth. To do so, small in-network coherence filters are placed inside the routers to collect sharing patterns and use them to determine the sharers for a block. This improvement allows snoopy-protocols to be scalable.

2.2.5 Token Based Protocols

This protocols are based in associate tokens with each block instead of state bits. There is a fixed number of tokens per block and the cores can exchange these tokens. A core with one or more tokens can read the block while a core with all token can either read or write the block.

Token Coherence protocol consists of two parts: correctness substrate and a performance protocol. While the former is responsible for ensuring safety and liveness, the later specifies what a cache controller does on a cache miss. Both snooping and directory can be interpreted as a Token Coherence protocol.

Under this classification we have [MHW03], which proposes TokenB (Token-Coherence-using-Broadcast), a specific Token Coherence performance protocol[4] to exploit a low-latency unordered interconnect while avoiding indirection. TokenB acts as a traditional MOSI snooping protocol, allowing cache-to-cache misses to achieve low-latency requests, until a transient request fail due to races. When this happens, protocol reissues until processor times out in which case a persistent request would be sent in order to prevent starvation.

In addition, in [MBH+05] Michael. R. Marty et. al. develop a M-CMP (Multiple-CMP) that is flat for correctness but hierarchical for performance, namely TokenCMP. Until this approach, the common solution was to use a hierarchical protocol separating inter-CMP requests from intra-CMP requests. However this leads to some difficult-to-verify race conditions. Furthermore they usually used directory-based protocols and, therefore, extra latency for sharing misses. Nevertheless Token-based protocols are good suited for these M-CMPs, and with TokenCMP a number of optimizations are proposed in order to improve performance under high-contention scenarios.

Finally, in [RBM08] PATCH (Predictive/Adaptive Token Counting Hibrid) is proposed. PATCH is a coherence protocol that extends the directory-based protocols in order to be capable of track the number of token to enforce coherence permissions. In have the same aforementioned properties than other token-based protocols: support direct requests in

[4]A performance protocol optimize for the common case and rely on the underlying correctness substrate to resolve races and prevent starvation.

unordered topologies and prevent starvation. However the main contributions of this protocol are: it introduces *token tenure*, which prevents starvation in a broadcast-free manner; and it depriorizes direct requests to improve the performance with regard to directory-protocols without impact on performance. This leads to have a protocol that retains the scalability of a directory-based protocol while matching the performance of broadcast-based protocols as TokenB.

2.3 Alternate Specific Approaches to Coherence

2.3.1 Sharing within Virtual Hierarchies

On a different approach to coherence maintenance among many-core CMPs we find Virtual Hierarchies [MH07]. It is based on the assumption that maintaining global coherence among all nodes in a CMP MPCore will be very unlikely. Due to the prohibitive cost of maintain global coherence it propose maintain it just among a limited subset of nodes.

These should be optimized for workload consolidation as well as traditional single-workload use. The main objectives of the memory system to support these approach should be: maximize shared memory accesses serviced within a VM, minimize interference among separate VMs, facilitate dynamic reassignment of cores, caches, and memory to VMs, and support sharing among VMs.

Virtual Hierarchies propose using virtual hierarchies to overlay a coherence and caching hierarchy onto a physical system. To do so, two flavors of hierarchical coherence are proposed: the first is a two level directory protocol, and the second is a first level local directory protocol with a backing broadcast mechanism for global coherence requests. Global broadcasts will likely be rare in this scenario but will experience lower latency and power by utilizing a multicast router.

2.3.2 Coherence on Heterogeneous Multiprocessor systems

On large-scale heterogeneous multiprocessing systems shared memory schemes with a directory-based protocol can be found for intercluster coherence issues, while a bus-based protocol can be used for intracluster coherence issues. However bus-based protocols cannot address heterogeneous coherence problem on intracluster communications.

In this specific scenario, in [SLB04] and [SBL04] Taewon Suh et. al. have developed an integration technique for different coherence protocols within the same multiprocessor system. They bear in mind the real time constraints of an embedded system. Real time operating systems (RTOS) could share semaphores, lock mechanisms, and mailboxes among others, making coherence support in heterogeneous systems mandatory.

This approach focuses on integrating invalidation-based protocols with the assumption that cache-to-cache transfers will occur only with nodes supporting MOESI protocol. To do so, they restrict the usage of protocol states that the many protocols does not share, using read-to-write conversion (integrating MEI with other protocols requires the removal of the Shared state), shared-signal assertion and deassertion (integrating MSI and MESI requires the removal of the Exclusive state), and Snoop-hit buffer (in MOESI protocols, when a snoop-hit occurs on the bus a back-to-back burst of external memory is required). Through the usage of this technique up to 51 performance improvement have been achieved.

CHAPTER 3

Coherence on the Target System

In this Chapter is described the target system and all the elements of it as defined in the target system of the vIrtical Project, by reason of this dissertation being within its framework.

It will also be described the coherence needs for each element. It is needed to understand the specific needs of the system along with the sharing patterns analysis in order to select a suitable coherence protocol approach.

3.1 Description of the Target System

The target system considered in the project is a heterogeneous multiprocessing system with a NoC connecting the main processing units, one or more GPU-like general-purpose programmable multi-core accelerators (GPPA) to enhance performance, several DSPs and functional HW Processing Units (HWPU), in addition to storage and I/O Resources.

The ARM architecture is defined as the target processing host, more precisely ARM's big.LITTLE architecture. It consists of one dual-core Cortex-A15 MPCore and one dual-core Cortex-A7 MPCore connected using the ARM CoreLink CCI-400 as seen in Figure 3.1.

Other characteristics of the target system are:

- 4 L1 caches (one per core) each with 128 sets, 4 ways and a line size of 64 bytes.

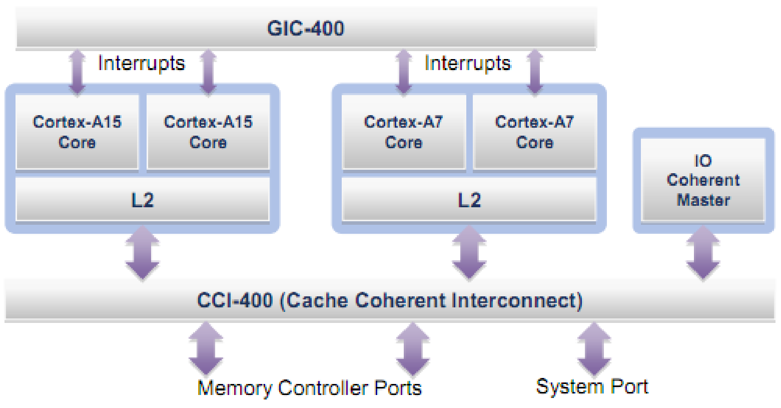


Fig. 3.1: Cortex-A15 CCI Cortex-A7 System

- 1 L2 cache (shared by all the cores) with 512 sets, 16 ways and a line size of 64 bytes.
- Caches are inclusive (L1 caches' content is included in L2).
- 2 AMBA4 ACE slave ports for connecting fully coherent masters
- 3 AMBA4 ACE-Lite slave ports for connecting non-coherent masters that can nevertheless snoop the ACE masters.
- 3 AMBA4 ACE-Lite master ports for connecting to memory and System NoC.

It can also be added an extra level of shared cache which constitutes the L3 level on the memory hierarchy. Therefore, the ARM based coherence protocol should be extended to also manage L3. We will assume a noninclusive policy for the L3 cache level in order to maximize the overall system capacity. Therefore, L3 will behave as a victim cache. Outgoing memory requests issued by either processor clusters or system accelerators should be routed toward the corresponding home memory L3 device. L3 module's NIs will include a directory cache/snoop filter in charge of forwarding the corresponding snoop request either to the potential sharers, the L3 module or the main memory. L3 module's NIs will support a MOESI based snooping coherency protocol

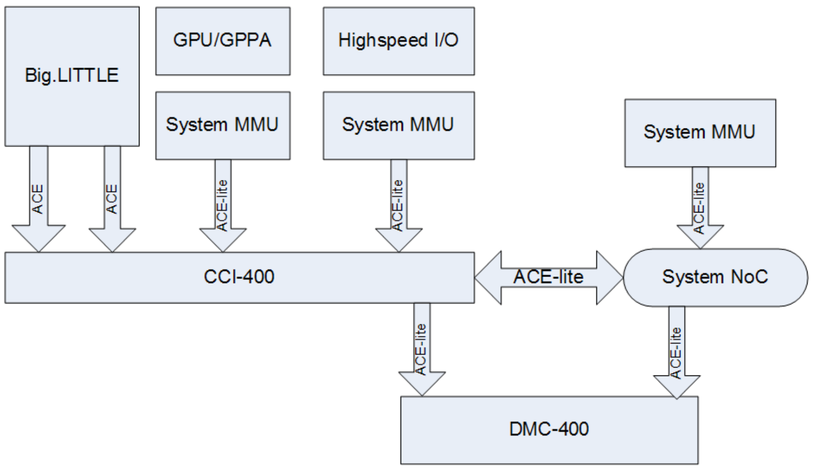


Fig. 3.2: Main elements of the target system

based on Hammer[1] (implemented in the AMD *Opteron*^{TM}). There is no coherency management for the instruction caches, including no automatic data/instruction cache synchronization.

3.2 Coherence Solutions in ARM Architecture

This Section is focused on the present solutions for coherence on the ARM Cortex-A family processors.

Embedded applications have driven the evolution of ARM's processors due to its high performance needs. RISC (Reduced Instruction-Set Computing) processors has evolved over the last years to address some of these embedded parallel applications' demands, including variable cycle execution, conditional execution or 16-bit Thumb Instruction Set[2] among others.

[1]Hammer avoids keeping coherence information at the cost of broadcasting requests to all cores. Although it is very efficient in terms of area requirements, it generates a prohibitive amount of network traffic, which translates into excessive power consumption.

[2]Thumb Instruction Set is an instruction set developed to improve compiled code-density by losing some functionality. It has more recent revisions, as the ThumbEE (Execution Environment) to output smaller compiler code without impacting performance.

In the same way, memory organization has also evolved, including an intelligent control block used to maintain coherence in an optimized way, including monitoring the system for a migratory line.

The coherency protocol within ARM processors has the following objectives:

- Correctness when sharing data across caches.
- Flexible protocol allowing components with different characteristics to interact.
- Maximize on-chip re-use of data.
- Simple definition, allowing ease of understanding and correct interpretation of the specification.
- Allows a trade-off to be made between high performance and low power.
- Reduces the interference with a master's cache from snooping traffic.

3.2.1 L1 Memory Organization

As seen in the Figure 4.1, the ARM Cortex-A15 have separate instruction and data caches. Both Physically-Indexed and Physically-Tagged (PIPT) and both implementing Last Recently Used (LRU) replacement policy. On a cache miss, critical word-first filling is performed also in both caches.

Cache coherence is supported explicitly on L1 data cache. L1 data cache has many combinations for different memory regions which determine processor behavior, impacting each one on its performance, including: Write-Back Read-Wite-Allocate, Write-Back No-Allocate, Write-Through No-Allocate, Non-Cacheable, Strongly-Ordered and Device. In this classification for memory types Strongly-Ordered and Device types are stricter and does not allow neither cache, merge or read accesses. Device memory can be nevertheless buffered.

All memory requests for pages that are marked as Inner Shareable (see Section 3.2.3) in the page tables and are Write-Back cacheable, regardless of allocation policy, are coherent in all caches that comprises Inner Domain. This comprises L1 data cache, L2 cache and all other Cortex-A L1 private data caches at a minimum.

It is unpredictable whether memory requests made to pages marked as Inner Non-Shareable are coherent within a Cortex-A family processor. No code must assume that this pages are incoherent among the caches.

Finally, the L1 data cache implements a MESI coherence protocol (Section 2.1.2).

3.2.2 L2 Memory Organization

L2 Cache is PIPT and uses a Random replacement Policy. It has an integrated Snoop Control Unit (SCU) connecting up to four cores within a Cortex-A MPCore device. L2 also interfaces with AMBA 4 (ACE) interconnect and an Accelerator Coherency Port (ACP) implemented as an AXI3 slave interface. It only incorporates a single dirty bit per cache line. Any write to a cache line results in the line being written back to main memory after its eviction.

Consequently, all coherence actions are taken through the SCU. The SCU uses hybrid MESI (Section 2.1.2) and MOESI (Section 2.1.4) protocols to maintain coherence between the L2 cache and the many L1 data caches present within the Cortex-A MPCore. It contains a snoop tag array with a duplicate of each L1 data cache directory. This snoop tag array aims on reducing traffic on the bus between L1 caches and the L2 memory cache. Any line in the Modified/Exclusive state belongs to L1 memory system. Any access hitting against a line classified as M/E must be serviced by the corresponding L1 and facilitated to the L2 memory. On the other hand, if the cache line is either Shared or Invalid then L2 cache can supply the data.

The SCU also contains buffers capable of handle direct cache-to-cache transfers between cores and therefore avoid reading or writing any data on the ACE buses. Lines can migrate between L1 caches without changing L2 state of cache line.

Snoop tag arrays are queried as a result of ACP shareable transaction, considered also as coherent. When a read occurs in a shareable line residing in one of the L1 data caches as M/E state, the line is transferred from L1 cache to the L2 memory cache and back on the ACP.

MOESI state machine for cross-cluster coherency in the ACE bus uses a specific terminology for cache line state maps across, being as shown in Table 3.1.

MOESI	**ACE**
Modified	Unique, Dirty
Owned	Shared, Dirty
Exclusive	Unique, Clean
Shared	Shared, Clean
Invalid	Invalid

Table 3.1: Terminology translation between ACE and MOESI state machines

3.2.3 Sharing Domains

The target system should be aware of the domain shareability levels present within the ARM architecture characteristics. In this case it should be allowed the presence of, at least, two levels of coherent shareability marked in the Page Table. Inner-Shareability (contains various -more than one- masters) applies to the processor subsystem and Outer-Shareable (contains all masters in the Inner domain and may also contain additional masters) extends other levels in the hierarchy. Other additional levels include

Non-Shareable (contains a single master) and System-shareable domains (contains all masters in the system and other domains).

Shareability domains are used to determine the other masters that should be considered for both memory coherency and for barriers. It is defined as a set of masters (usually cores). For coherent transaction, shareability domain is used to determine which other master might have a copy of the addressed location in their local cache and, therefore, determine which other masters should be snooped to complete the transaction. For barrier transactions, the domain is used to determine which other masters the barrier is establishing an ordering relationship with. It can be used to determine how far a barrier transaction is required to propagate and the blocking properties that are needed to establish the required ordering.

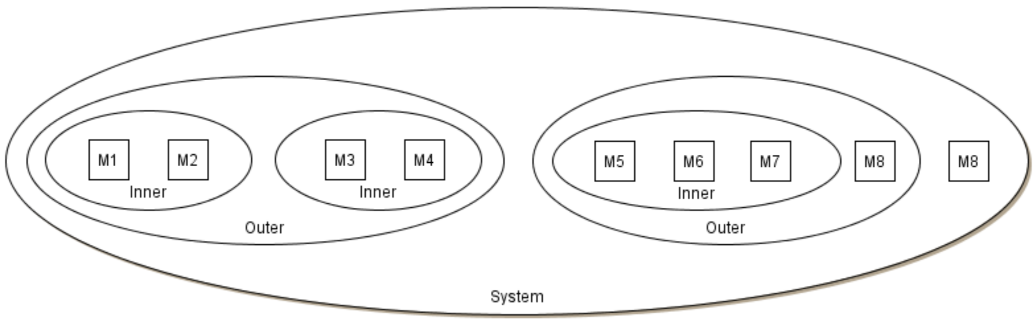


Fig. 3.3: Example of domain layout

In the Figure 3.3 can be seen a set of definitions including masters from M1 through M8. Domains are defined as non-overlapping. The System domain is used whenever a transaction must be visible to other masters in the system, including those that do not have hardware coherent caches.

On the other hand, in Figure 3.4 is showed an example of the domain layout for the final target system. In this case, the Cortex-A15 has four Masters within. Caches on the Masters inside the Cortex-A15 and inside the accelerator group are peer caches. Caches

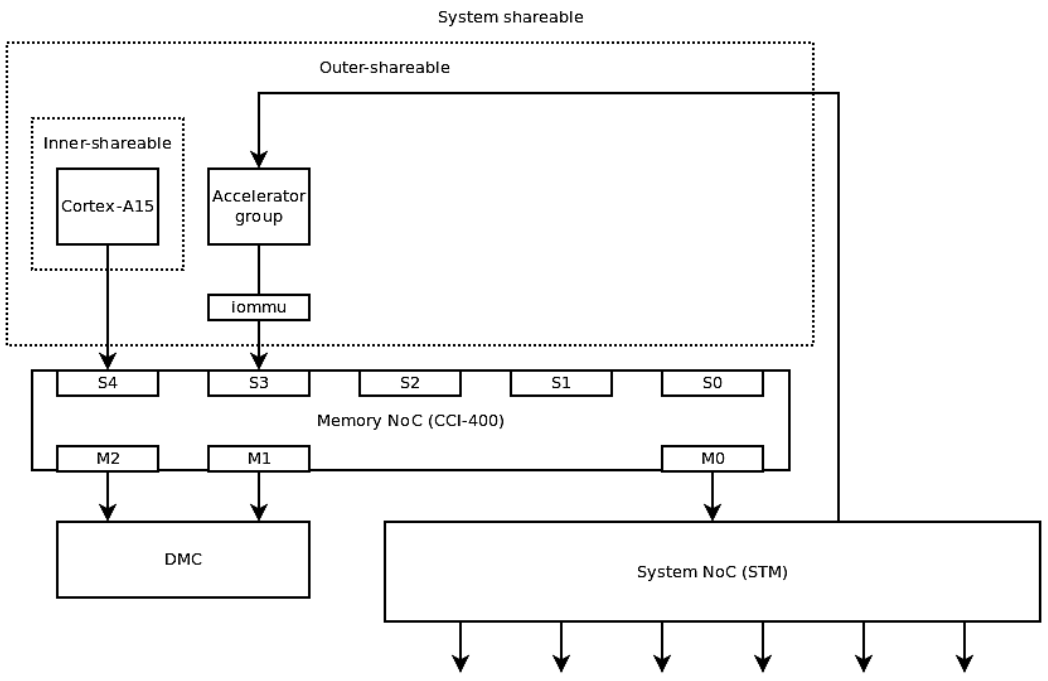


Fig. 3.4: Example of System Topology with domain layout

from Cortex-A15 are in the Inner Shareable Domain. Both Cortex-A15 and accelerator group are in the Outer Shareable Domain. Caches on the DMC are downstream caches.

Barriers

In a system, barrier transactions are used to guarantee the ordering and observation of transactions. Two types of barriers are supported: a memory barrier and a synchronization barrier. The former is issued by a master to guarantee that if another master in the domain observe any transaction after the barrier, it must be capable to observe every transaction also prior to the barrier; the later is issued by a master to determine when all transactions issued prior to the barrier are observable by every master within the appropriate domain.

System domain Synchronization barriers also require that all transactions issued before the barrier must have reached the end-point slaves they are destined to, before the completion of the barrier. A barrier transaction has an address phase and a response, but without any data transfer. Barriers enforce ordering because a master must not issue any transaction until it has received a response for the barrier on both read data and write response channels.

Distributed Virtual Memory

ACE includes support for Distributed Virtual Memory and has transactions that allow the management of a virtual memory system.

An Input/Output Memory Management Unit (IOMMU) is used to perform address translations from one address space to another. This can be:

- Translation from a virtual address space to a physical address space.
- Translation from a virtual address space to an intermediate physical address space.
- Translation from an intermediate physical address space to a physical address space

All components in the system must use a single physical address space and the use of IOMMU components allows different masters to operate in their own independent virtual address or intermediate physical address spaces.

3.2.4 Coherence Special Transactions

Coherent transactions are used to access shareable address locations, which may be held in the coherent caches of other components.

In ACE masters there are two configuration options (input signals for hardware, settings for model) to control external signaling of coherency management: BROADCASTINNER and BROADCASTOUTER. There is also BROADCASTACHEMAINT, which can be used to enable broadcasting of cache maintenance operations.

ReadClean, ReadNotSharedDirty and ReadShared

In the case that a master needs to perform a load from a location within a shareable memory area, there are three transaction types that can be used: ReadClean, ReadNotSharedDirty and ReadShared. These transactions allow the current holders of the line to retain their copy.

ReadClean transaction indicates that the master requesting the line require it to be a clean line, therefore, it cannot accept responsibility for a dirty line which needs later write back to memory. This type of transaction is typically used by a master that does not have the ability to accept a dirty cache line or has a write-through cache.

The ReadNotSharedDirty transaction indicates that the master requesting the read can accept a line in any state except SharedDirty. So, the line can be obtained either as clean (unique or shared) or as unique and dirty. In all cases it is acceptable for a cache that is being snooped to pass a line as dirty, even if it won't be accepted by the requesting master. In this situation, interconnect is responsible for writing back the dirty line to main memory. In fact when a cache receives a ReadClean, ReadNotSharedDirty and Read Shared transactions it must supply the data if it is holding a copy of it in order to complete the transaction. In any case, interconnect is responsible for passing the data back to master that initiated the transaction. If a cache is holding the data in a unique state, then it must change it into a shared state after the operation.

ReadShared transaction indicates that the master component requesting the read can accept a cache line in any state.

ReadUnique, CleanUnique and MakeUnique

When a master needs to perform a store to a location that resides in a shareable area of memory there are three transaction types that can be used: ReadUnique, CleanUnique and MakeUnique. All three of these transaction types ensure that there are no other copies of the location when the store occurs.

If the master is performing a partial line store, where it only stores some of the bytes in the entire line, and the master does not already have a copy of the line then it can use the ReadUnique transaction which both obtains a copy of the data and it also ensures that no other copies exist.

If the master is performing a partial line store and it already has a copy of the line then it can use a CleanUniquetransaction to remove other copies of the line. The CleanUnique transaction has the effect of removing all other copies of a line, but if it finds a cache that holds the line in a dirty state then it will ensure that the dirty line is written to main memory.

If the master is performing a full line store then it does not need to have a pre-store form of the line and it can simply remove all other copies. This is done using the MakeUnique transaction which effectively invalidates all other copies of the cache line.

ReadOnce, WriteUnique and WriteLineUnique

ReadOnce, WriteUnique and WriteLineUnique transactions are typically used by a master for accessing areas of memory that are shareable, but the issuing master is not going to keep a cached copy of the address, either because it does not want to allocate that line or because it does not have a cache at all. ReadOnce is used by a master to obtain a snapshot of the data which is not going to be cached for later use.

The ReadOnce transaction has the advantage that if a cache providing the data held the line in a unique state then it does not need to move to a shared state after the ReadOnce has occurred.

WriteUnique and WriteLineUnique transactions remove all copies of a cache line before performing a write. The WriteUnique transaction can be used for full and partial line writes and ensures dirty data is written to memory before performing the write transaction. The WriteLineUnique transaction must only be used for a full line write, where all bytes within the line will be written by the transaction. Unlike other transactions to shareable memory, ReadOnce and WriteUnique transactions issued by a master are not required to be a full cache line size. WriteLineUnique transactions, however, are required to be a full cache line size.

3.3 GPPA and Coherence

Embedded systems are resource constrained: battery capacity increases only slowly over time, hence embedded devices have tight energy budgets. An architectural countermeasure to achieve energy-efficient (MOPS/mm/W) targets is heterogeneous designs. Heterogeneity in embedded systems is typically achieved through the adoption of accelerator-based SoC designs, where a SMP multicore host processor is coupled to accelerators of different kinds: GPU-like general-purpose programmable many-cores (GPPA in the following) and several types of Hardware Processing Units (HWPU), implementing in hardware key computational kernels from the target application domain. Accelerator-based SoCs are already wide-spread, as witnessed by commercial products such as Qualcomm's Snapdragon, Nvidia Tegra, Apple Ax, TI OMAP. All the cited products feature a on-chip GPU-like accelerator, which the host processor can leverage to offload data-intensive computational kernels, achieving significant speedups and better energy efficiency even in common general-purpose applications.

While the embedded GPUs integrated in the mentioned SoCs are optimized for data-parallel (SIMD) computation, the focus of vIrtical project is on general-purpose many-core accelerators, which support a more flexible execution model (both data and task parallelism).

3.3.1 General-Purpose Programmable Accelerator

The targeted many-core programmable accelerator leverages tightly coupled clusters as a building block. A cluster consists of a configurable number (typically up to 16) of processors with private instruction caches. Processors communicate through a fast multi-banked, multi-ported Tightly-Coupled Data Memory (TCDM). The number of memory ports in the TCDM is equal to the number of banks to allow concurrent accesses to different banks. Conflict-free TCDM access have extremely low latency (ideally 1 cycle). TCDM are explicitly managed from software, and can be seen as a non-coherent memory area that accelerators can use for higher efficiency. Each cluster can also read/write into a cacheable memory segment. Transactions involving this address range are satisfied from a L2 cache, shared between multiple clusters. This L2 memory is kept coherent with the L2 cache in the host processor subsystem through services provided by the global NoC. Coherent caches could be present at the L1, co-existing with TCDMs. In this case the local NoC is responsible for managing coherency traffic as well. When copies of data residing in cacheable memory regions are allowed into TCDMs explicit coherence operations have to be taken (e.g. invalidate cache data if most recent copy is kept on TCDM).

The on-cluster communication fabric is a low-latency, high bandwidth logarithmic interconnect, and is built as a parametric, fully combinational Mesh-of-Trees (MoT) interconnection network. The interconnect provides fine-grained address interleaving on the memory banks to reduce banking conflicts in case of multiple accesses to logically contiguous data structures. If no bank conflicts arise, data routing is done in parallel for each core. In case of conflicting requests, round-robin scheduling coordinates accesses to memory banks in a fair manner. Banking conflicts result in higher latency, depending on the number of conflicting requests. Multiple concurrent reads on a same address are typically satisfied through read broadcast, which completes in one cycle.

Inter-core synchronization is supported by means of standard read/write operations at a memory bank providing test-and-set semantics (hardware semaphores). These memory locations are not cacheable.

In this template, scaling to larger system sizes is enabled by replicating clusters and interconnecting them with a scalable medium like a NoC. Each logarithmic interconnect routes data based on address decoding: a first-stage checks if the requested address falls within the local L1 address range or has to be directed off-cluster. If this is the case, the corresponding transaction is injected in the NoC through per-cluster Network Interfaces (NI).

All TCDMs in the accelerator subsystem are globally visible to every processor within a unique address space. The hierarchical interconnection system is thus in charge of routing I/O requests to the target memory module. Clearly, transactions hitting in the local TCDM have lower latency and higher bandwidth. Accessing remote memories (i.e. remote clusters, L2 or L3 memory) is subject to NUMA effects.

3.3.2 Hardware Processing Unit

Besides GPU-like programmable many-core accelerators, we also want to explore the adoption of Hardware Processing Units (HWPU), namely functional units executing in hardware key computational kernels from the targeted application domain. HWPUs could be integrated in the GPPA SoC and - similar to processors - should access data directly from the L1 TCDM through the logarithmic interconnect.I

The main advantage of this approach is that it enables zero-copy semantics for processor-accelerator communication. More specifically, different from the typical offload approach to accelerator exploitation, data need not be moved in and out of the accelerator private memory space. To support this communication model it is necessary to design ad-hoc interfaces.

3.4 Coherence for Other Devices in the Target System

There can be room for other devices with especial coherent needs that will be described in this Section.

I/O Devices can be found, and they may refer to Ethernet or USB controllers among others. Anyhow, in case that any of these devices stores a main memory block in some register or private storage unit; or the block pertains to a shared memory region, it must be aware of any change in the data and always retrieve the most updated version of it.

To do so, even for non-coherent devices (which won't need to maintain coherence, as they won't cache any data) the cache coherence protocol must provide the most recent version of data in the system. Thus, it is preferable to attach them to the ACE-Lite port (or other bus with a compatible interface) to best power efficiency and performance along with the required correctness. ACE-Lite supports transactions described in Section 3.2.4, and can be issued by this type of devices to retrieve data, due to the fact that they won't keep a cached copy of it. In the case of a read (ReadOnce transaction), the data should be provided by the node holding the most recent valid copy of it, without modifying the state of block in any sense. As a counterpart, if the device requires to write (either using WriteUnique or WriteLineUnique transactions for a word and a line writes respectively) into a shared memory region, the data should be previously invalidated and updated if proceeds.

Some of these devices may contain a Direct Memory Access (DMA) controller. DMA is a feature of modern computers that allows certain hardware subsystems within the computer to access system memory independently of the central processing unit (CPU). A DMA controller can generate addresses and initiate read or write accesses to memory as requested by the CPU. It has many registers that can be accessed to read or write by the CPU. Also, a DMA controller can read or write a burst of contiguous bytes of any length (hundreds or thousands of words in a row).

Furthermore, the DMA controller can allow access to main memory from a peripheral device. This can lead to cache coherency problems: if a block is cached and modified without the DMA being aware of it, external devices can access to stale copies of data.

This issue can be addressed if the cache-coherent systems implement a method in hardware whereby external writes are signaled to the cache controller which then performs a cache invalidation for DMA writes or cache flush for DMA reads. In the other hand, as an alternate solution, the system can rely on the OS that must ensure the cache line is flushed before outgoing DMA transfer is started. The latter approach introduces some overhead to the DMA operation, as most hardware requires a loop to invalidate each cache line individually.

As an example of a device relying on the DMA to deal with the coherency related problems, we can fin a Digital Signal Processor (DSP). A DSP is a specialized microprocessor with an architecture optimized for the fast operational needs of digital signal processing.

Digital signal processing algorithms typically require a large number of mathematical operations to be performed quickly and repeatedly on a set of data. Signals (perhaps from audio or video sensors) are constantly converted from analog to digital, manipulated digitally, and then converted back to analog form. Many DSP applications have constraints on latency; that is, for the system to work, the DSP operation must be completed within some fixed time, and deferred (or batch) processing is not viable.

Chapter 4

Analysis of Applications

The use of computers has been extended to most areas of our everyday life. These embedded devices present new challenges of security, scalability and power efficiency. We focus on these challenges addressing the design of the coherence protocol to be adopted to the need of these heterogeneous multicore platforms.

The cache coherence problem arises when copies of the data stored at several private caches associated to different cores, are reachable at the same time by two or more cores and modifiable by some of them. The cache coherence protocol must guarantee coherence of the data through the entire system which means deciding how and when a single core is granted permission to modify data and ensuring that subsequent readings of the written data by other cores will attain updated copies of the modified data (multiple readers). To do so, different cache coherence mechanisms can be applied. Commonly, these mechanisms introduce certain overhead in terms of either coherence traffic issued and storage resources required, which can significantly penalize performance, increasing the execution time of the running applications as well as power consumption. The current trend to increase the number of cores into CMP and MPSoC systems further aggravates this problem, as the cache coherence protocol does not scale due to its resource overheads and its indirection when accessing data (access latency is increased).

On the other hand, a different approach to tackle the coherence problem has recently been proposed [CRG$^+$11, HFFA09, KAKH10], consisting on removing coherence maintenance for those data objects that do not need it, either because they are not shared (private to one core) or because they are shared but never written by any core. This approach requires the use of effective mechanisms to identify data blocks that do not need coherence maintenance, which in turn may introduce certain overhead. However, the success and suitability of the selected coherence mechanism will strongly depend on both the architectural context of the system it is being applied to and the sharing patterns of the applications running on the system.

Therefore, a detailed analysis of the sharing patterns of the applications to be supported is needed, in order to identify opportunities of applying one or another cache coherence mechanism together with different coherence optimization techniques.

4.1 Analysis Methodology

This section describes the target system used for the analysis and the methodology followed in order to capture the information required to properly perform it.

4.1.1 Simulation Tools

ARM FastModels simulator provides out of the box programmer's view models of the ARM processors. Thus, it is both functionally accurate and easy to use since ARM processors models are already implemented as an Instruction Set Simulator. We use this simulator to model the target system and to run the targeted applications on top of it.

The system processor of choice is a quad-core Cortex-A15 MPCore (Figure 4.1), despite the fact that the target system is suposed to implement a big.LITTLE, as mentioned on Chapter 3. The trade-off between complexity and accuracy makes this option more suitable. Conclusions obtained with the quad-core Cortex-A15 MPCore can be extrapolated to its big.LITTLE counterpart with acceptable precision. Furthermore, big.LITTLE has two different working modes, either only Cortex-A15 or Cortex-A7 processors are awake or they are both working at the same time. When all processors are working at the same time they will not run a parallel application in both of them because they work at different frequencies. Since we are mainly concerned on parallel applications, the use of big.LITTLE is not mandatory.

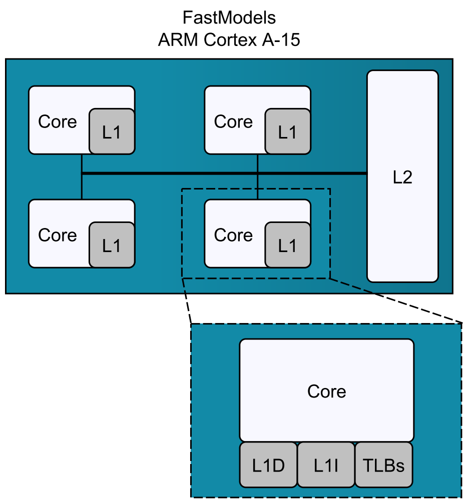


Fig. 4.1: Memory Hierarchy for the quad-core Cortex-A15 MPCore

The model of ARM Cortex-A15 provided with FastModels is capable of running basic applications, but it does not cover all the requirements of an operating system, which is needed to evaluate and benchmark parallel applications. We thus use a more complex model also provided with FastModels (namely RTSM-VE Cortex-A15) that allows the simulation of both operating systems and applications. In this RTSM-VE model, as seen on the Figure 4.2, the cores are connected directly to a Versatile Express platform through a 64-bits AXI bus. This platform includes the Motherboard Express μATX, which has been especially designed to support future generations of ARM processors, and the CoreTile Express daughterboard with the on-board DDR2 SDRAM. It is executed on top of it a Linux system based on TinyBSD.

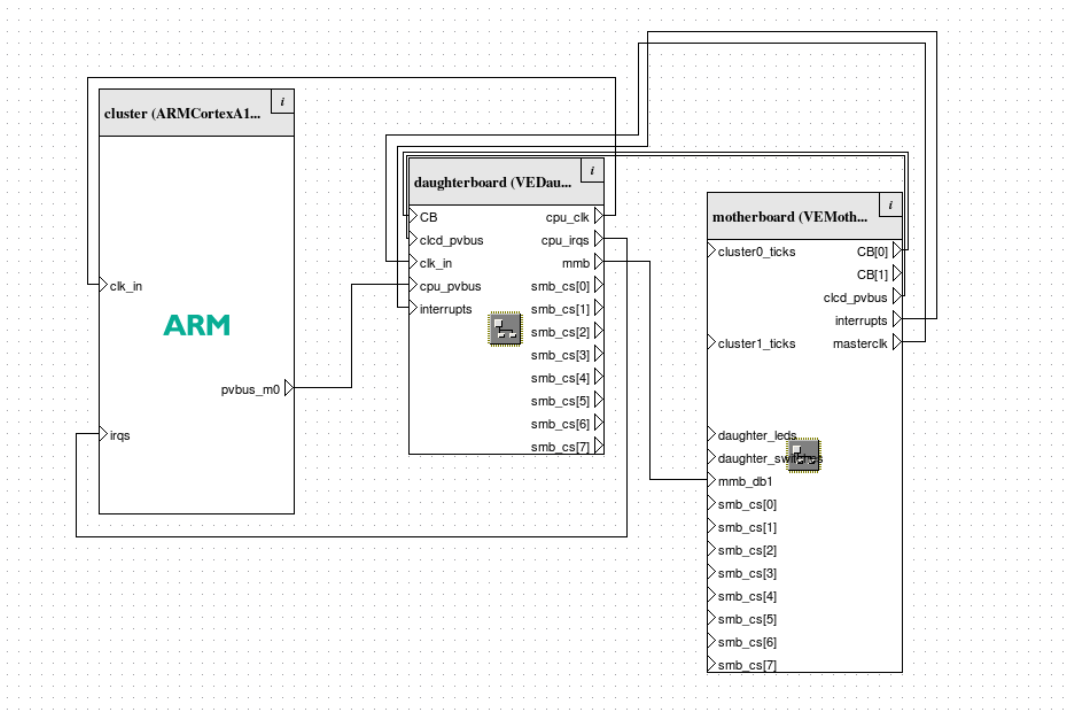


Fig. 4.2: ARM FastModels RTSM-VE model

4.1.2 Trace Acquisition Methodology

In order to perform the sharing pattern analysis we need to capture all memory accesses from the cores, being our interval of interest the parallel section of the applications, namely the section executed by several cores at the same time. To identify this section we explored the application code trying to reach the starting point and end point of threads. To delimit this section and make it recognizable by FastModels we introduce a special *nop* instruction on the application available on the ARM Instruction Set.

FastModels supports the use of a Model Trace Interface (MTI) plug-in that permits us to consistently track the execution of the model. Through implementing an MTI plug-in for tracing memory accesses produced by the cores and adding it to the simulation we are able to trace exactly what we need in the form that we require.

MTI plug-in provides many different sources to trace, but the more verbose the trace obtained is and the more sources are involved, the more it slows down the simulation. Since it takes billions of instructions to boot a Linux system on FastModels, we need to deactivate the output and minimize the number of sources of the tracing until the starting point of the segment of interest is detected. We have achieved an acceptable compromise solution by capturing only the instructions fetched by the cores until we reach the aforementioned special *nop*, and subsequently tracing loads, stores, and fetches until we get to the ending special *nop*.

The ARM Simulator provides programmer's view models with some limitations. On system simulators there is a trade-off between speed and accuracy. FastModels in particular opts for the execution speed thus lacking some features needed for our analysis, such as:

- Instruction timing: a processor issues a set of instructions (a.k.a a quantum) at the same point of the simulation time, and then waits some amount of time before executing the next quantum, being impossible to determine the right time each individual instruction is executed.
- Bus traffic: bus traffic has several optimizations that make it inaccurate.
- It does not support out-of-order execution and write-buffers as architecturally defined: execution on FastModels is only an approximation to execution of architecture and it must be thus considered.

Finally, we describe the trace file format. We need to obtain traces with the information required for coherency modeling, including the core id (to characterize the number of sharers of the block), the address (to identify the block being accessed), the type of access (to classify the block as data or instructions and to discriminate writes from reads) and the data (for further analysis). Traces are as follows:

```
core,address,type,data
0,001EA10C,l,B6F6713C
0,B6F6713C,f,E92D001F
0,BEF1DE10,bs,BEF1EEF4,BEF1EF10,BEF1F984,001A0C9C,00000003
```

Where type refers to: l (load), s (store), f (fetch), and b applied to l or s (burst load-store). When a burst is detected the data field is extended to the total amount of data exchanged. Both addresses and data are coded in hexadecimal format.

4.1.3 Applications

We have obtained traces with five of the many different algorithms in the OpenSSL suite. OpenSSL is a well-known suite of open-source library and tools implementing cryptographic algorithms used for authentification and secure data transfers over networks. It is used by many services such as https and ssh. As indicated by its documentation, it implements the following cryptographic functions:

- Creation of RSA, DH and DSA key parameters
- Creation of X.509 certificates, CSR and CRL
- Calculation of message digests
- Encryption and decryption with ciphers
- SSL/TLS client and server tests
- Handling of S/MIME signed or encrypted mail.

The algorithms selected to obtain the traces are: three hash algorithms based on Secure Hash Algorithm (SHA1, SHA256 and SHA512), and two encryption algorithms based on the Advanced Encryption Standard (AES-128-ECB and AES-256-ECB).

4.2 Sharing Patterns Analysis

As commented above, there exist several recent proposals that take advantage of memory block classification for different purposes, such as enhancing efficiency of directory caches, reducing coherence overhead or better taking advantage of NUCA caches. All of them are mainly based on the classification of blocks in private (P) and shared (S). Moreover, some others extend this classification to read (R) only and written (W).

So the scheme we propose in order to analyze blocks is made classifying them as:

- PR (Private Read-only): Only one processor accesses the block. All accesses are loads. Thus, the block is private to the core and only that core reads the block but does not write it.
- PW (Private read-Write): Only one processor accesses the block. At least one access is a store. Thus, the block is private to the core and this core reads and writes that block.
- SR (Shared Read-only): At least two processors access the block. All accesses are loads. Thus, the block is shared by several cores but no one writes on that block.
- SW (Shared read-Write): At least two processors access the block. At least one access is a store. This is the most interesting mode as it requires coherence protocol support. In this mode the block is shared and is written by at least one core.

Considering this classification, the only blocks that actually need coherence maintenance are the SW ones and therefore we can take advantage of the fact that the remaining blocks do not need it, either because they are accessed by just one core or because they are only read by any number of cores. So special attention will be paid to SW blocks.

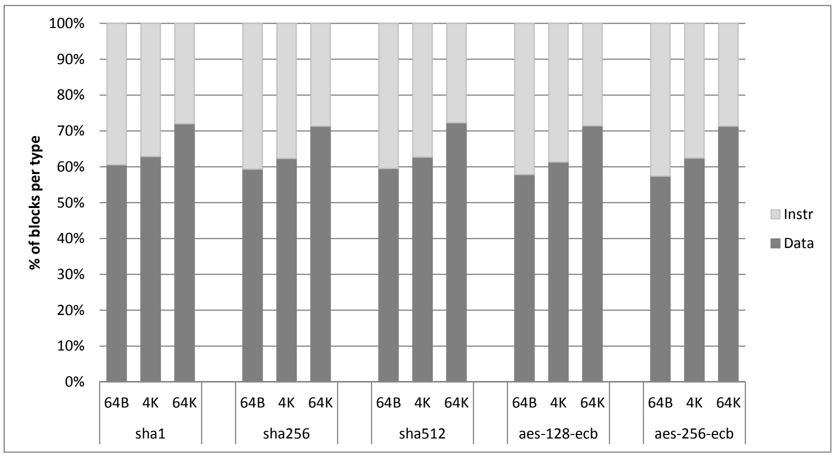
The classification schemes proposed in the literature have used different granularities: blocks [HDH11, PSNB10] and pages (OS-based schemes, as can be seen in [CRG$^+$11, HFFA09, KAKH10]), looking for a trade-off between detection accuracy and the required overhead. So our analysis is made with three different granularities based on blocks and pages: 64 bytes block, 4 Kbytes pages, and 64 Kbytes pages. This is interesting since coherency with page granularity is easier to implement and manage. Working at page level allows us to rely on the operating system to detect whether coherence needs to be applied or not, aiding to reduce the hardware overhead and complexity. On the other hand, the use of page level granularity allows us to analyze how critical is the block misclassification introduced with coarser grains. In addition, studies at page level granularity are intended to identify the viability of applying cache coherency at page level instead of block level.

Basically, we provide two kinds of analysis. The first one, referred to as static analysis, is intended to count the number of blocks included in each category. The second one, referred to as dynamic analysis, shows the number of accesses to blocks for each category. Both views complement each other and allow us to identify which categories of blocks are the most frequent and which ones are the most accessed.

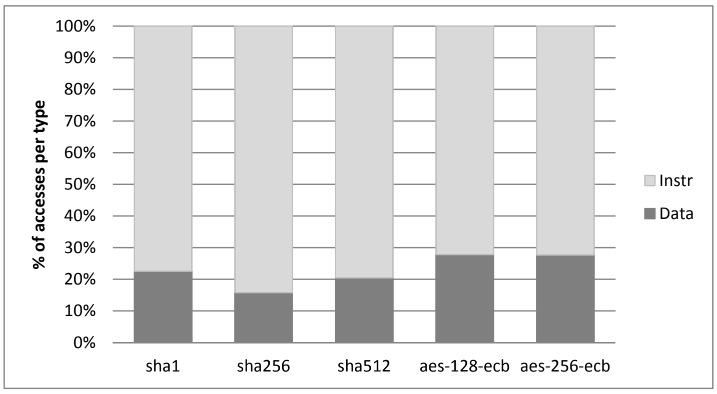
The previous classification will help us later to identify which are the best optimization opportunities when developing the appropriate coherence protocol.

4.2.1 Analysis Results

The results presented are focused on the data blocks or pages. As can be seen on the Fig. 4.3(a) and Fig. 4.3(b) the proportion of data blocks and accesses is more relevant. Also, due to the fact that most of the instruction blocks are shared through all four cores, the classification between Private and Shared instruction blocks is less interesting in this case. Finally, we detected no interleaving between data and instruction blocks at different page granularities.



(a) Proportion of blocks



(b) Proportion of accesses

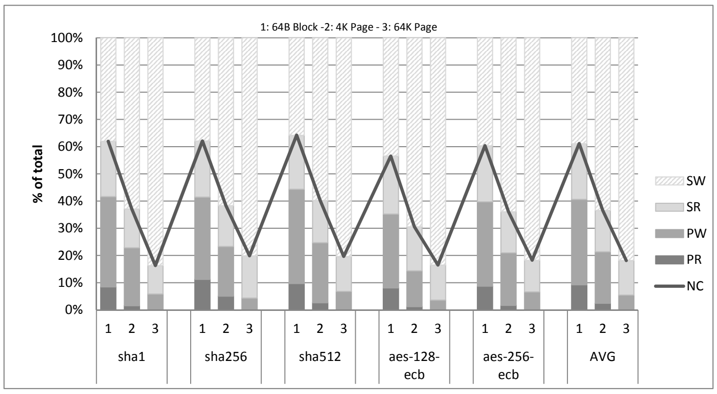
Fig. 4.3: Data/Instruction classification

In all figures, the results obtained for each of the analyzed applications, together with the resulting average values, are displayed.

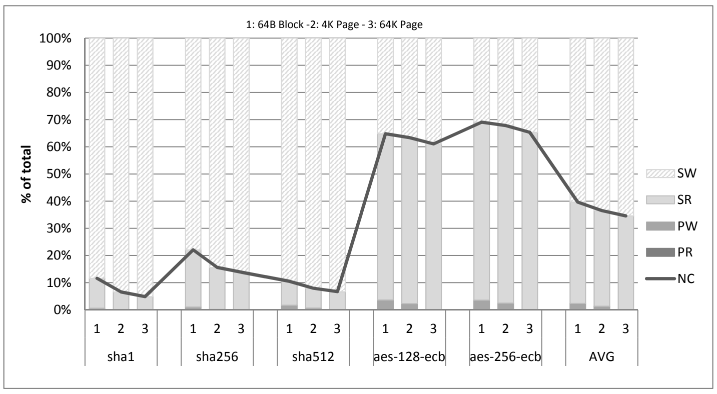
Fig. 4.4(a) shows the block classification based on the detection of the Private-Shared Read-Write scheme for every block requested at different granularities. First of all, it is observed that, on average, 40% of data blocks are private (PR or PW). The remaining blocks are shared (60%), but notice that indeed only 40% of data blocks are SW, that is, they require coherence maintenance. However, this promising result vanishes when the granularity used for classifying the blocks is increased. As can be observed, the coarser grain used, the more shared and written blocks are found. In particular, for 4KB pages, the percentage of SW blocks is greater than 60%, whereas this percentage, on average, exceeds the 80% for 64KB pages. This means that SW blocks are concentrated in a certain number of pages, but they are mostly distributed among them. As a consequence, the detection accuracy decreases as far as the granularity is increased in order to simplify the detection process, leading to a misclassification of page blocks. Notice that just one SW block contained in a page will cause the page to be classified as SW.

Fig. 4.4(b) shows the dynamic analysis. It is observed that despite the fact that SW blocks just represent 40% of the total number of blocks, as was shown above, they agglutinate tough the larger number of accesses (60% on average). Furthermore, the number of accesses to private data blocks is indeed negligible. Unlike the static analysis, the larger differences between applications are observed here. Also, the number of accesses classified as SW hardly increases as granularity becomes coarser. On average, it reaches a 70% for 64KB pages. Notice that this is an expected result as long as the highest percentage of accesses inside a page is destined to SW blocks. Given that most of data memory accesses require coherence maintenance, the design of the cache coherence strategy will be a key element to provide high performance.

In order to offer a deeper insight into the sharing degree of data blocks, let us analyze to what extent they are shared, that is, how many cores share each of these data blocks. So, in Fig. 4.2.1 we analyze the number of sharers per block. If there are no sharers and the block is only accessed by one core, it corresponds with a Private block detected in the previous analysis. We consider also both static (Fig. 4.5(a)) and dynamic analysis (Fig. 4.5(b)). In Fig. 4.5(a), we can observe how, on average, about 20% of the blocks are shared by just two cores, 15% are shared by three cores, and 25% of them are shared



(a) Static Analysis

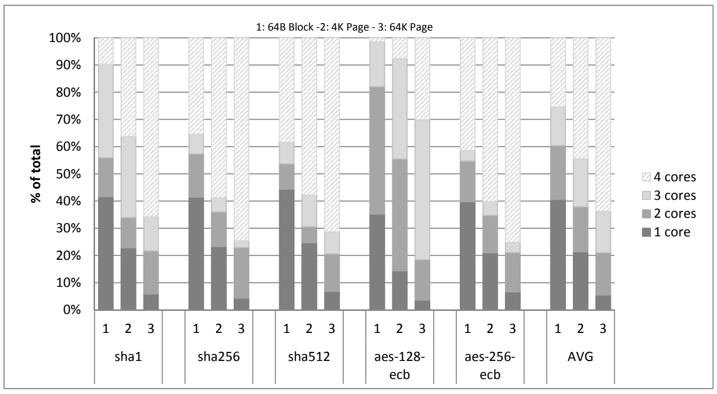


(b) Dynamic Analysis

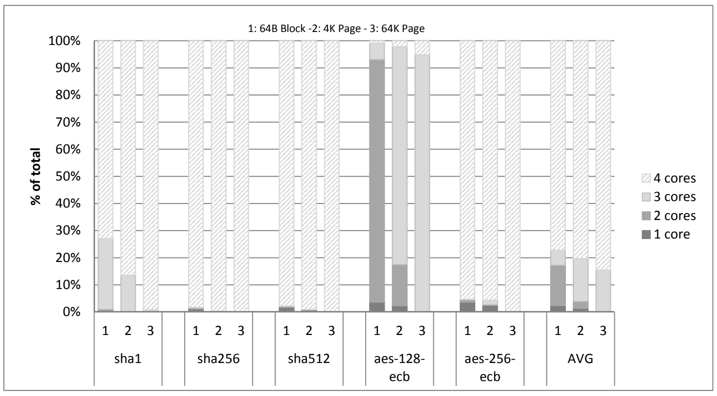
Fig. 4.4: Static and dynamic Analysis per block type

by four cores. As the detection granularity increases, the number of blocks shared by all the cores is larger. The reason is the same as that pointed out with respect to Fig. 4.4(a). Moreover, from Fig. 4.5(b), it is observed that the most accessed data blocks are those shared by all four cores. This result corroborates even more the importance of carrying out an appropriate design of the cache coherence mechanism as far as a large number of cores are usually involved in the coherence maintenance of the data blocks.

Until now, the analysis has been focused on classifying blocks by using different detection granularities. However, it is also interesting to just classify pages. It may be important to assess the convenience of managing cache coherence in a per page basis instead of the



(a) Static Analysis

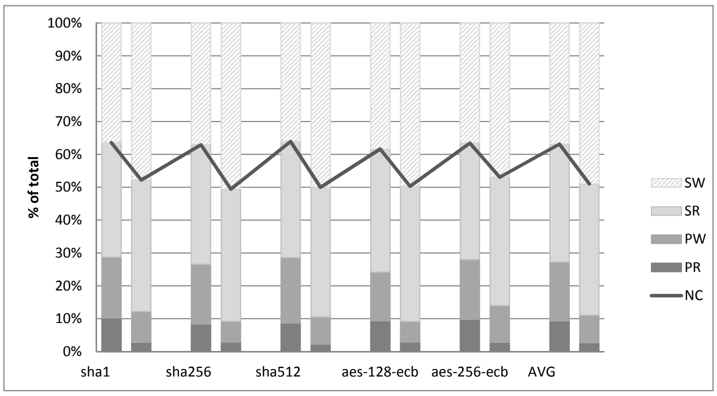


(b) Dynamic Analysis

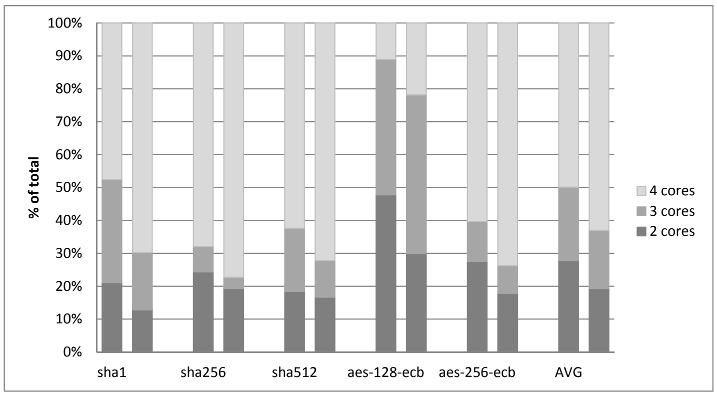
Fig. 4.5: Static and dynamic Analysis per number of sharers

usual strategies based on block tracking. In this case, page classification in PR, PW, SR, and SW classes is as follows. A page is classified as SW when it contains at least a SW block. Otherwise, it will be classified as SR if at least one of their blocks is SR. On the contrary, if the page does not contain neither SW nor SR blocks, it will be classified as PW if at least it contains a PW block. Otherwise, the page will be classified as PR. In this sense, Fig. 4.6(a) shows the page classification for 4KB and 64KB page sizes. As can be seen, more than 35% of the 4KB pages require coherence (they are SW), whereas this percentage increases until near 50% for 64KB pages. This means that SW blocks are not spread over all the pages, but they are indeed distributed between a limited number of pages, larger as the page size increases.

In order to offer a deeper insight into SW pages, from now on, SW pages become the focus of our analysis. Firstly in Fig. 4.6(a) we classify pages per access type and subsequently in Fig. 4.6(b) we discern the number of sharers on pages classified as SW. As can be seen, like it was observed on analyzing block sharing, most pages are shared among three or more cores, as more as larger the page size is. In particular, half of the blocks are shared between 4 cores for 4KB pages, whereas the percentage exceeds the 60% when page size is 64KB.



(a) Classification of pages per type



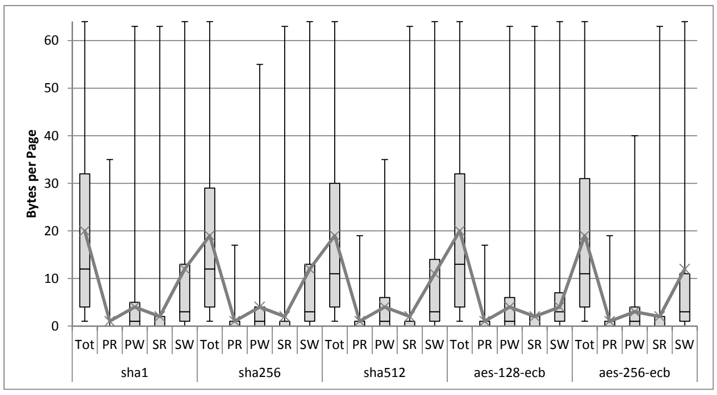
(b) Classification of SW pages per number of sharers

Fig. 4.6: Page Classification

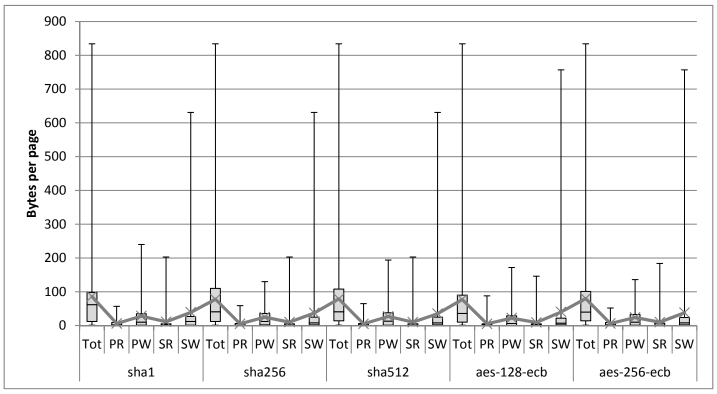
As commented before, the main disadvantage of using page granularity to detect blocks requiring coherence is the misclassification that page blocks may suffer. Notice that a single block can determine the classification of the rest on the same page. In order to analyze this effect, we study in depth the internal anatomy of SW pages. The study is performed by considering the type each block in the page had been assigned in case of having assumed block granularity in the detection process. In particular, we proceed to count the number of each of the block types contained in the page. Information is represented making use of box-and-whiskers plots. This will help to determine how populated the pages are and the real significance of block misclassification.

Fig. 4.7(a) and Fig. 4.7(b) show results for 4KB and 64KB page sizes, respectively. First of all, it is observed that pages are hardly populated, so much less, the larger the page size is. However, a great variability is observed, from pages hardly containing a few blocks until pages crowded with blocks. On average, the medium value of blocks per page is about 20 out of 64 for 4KB pages and 80 out of 1024 for 64KB pages. Anyway, the precise distribution of the total number of blocks and the number of blocks of each type can be observed in the aforementioned figures. Regarding SW blocks, it can be observed that, for a page size of 4KB, most of the blocks in SW pages are SW blocks, but when the page size is increased, the PW blocks become more frequent. Despite this, the number of SW blocks present in the page is very small in relative terms (on average, the 75% of 4KB pages have less than 12 SW blocks, whereas the 75% of 64KB pages have less than 25 SW blocks). These results may suggest the possibility of applying fine grain detection techniques inside SW pages in order to isolate true SW blocks, thus limiting coherence maintenance actions to them.

We have also performed a dynamic analysis of SW pages in order to obtain the proportion of store access. As can be seen in Fig. 4.8, about 30% of the memory accesses to SW pages are stores. Obviously, these accesses will be destined to either SW or PW blocks. This kind of studies allow us to assess the convenience of applying update strategies



(a) 4K Pages



(b) 64K Pages

Fig. 4.7: Proportion of blocks per type on SW pages

instead of invalidate ones.

Finally, it is carried out an analysis in search of the existence of producer-consumer patterns in SW pages. Fig. 4.9(a) shows the proportion of blocks presenting the producer-consumer pattern related to the amount of blocks requested within the page for different page granularities. As observed this percentage is relatively small. So, in order to definitively observe whether or not the producer-consumer pattern is relevant on the analysis, we studied the percentage of accesses done to blocks presenting the pattern within the SW pages. As can be seen on Fig. 4.9(b) not even the 0.003% of the accesses are done to these blocks.

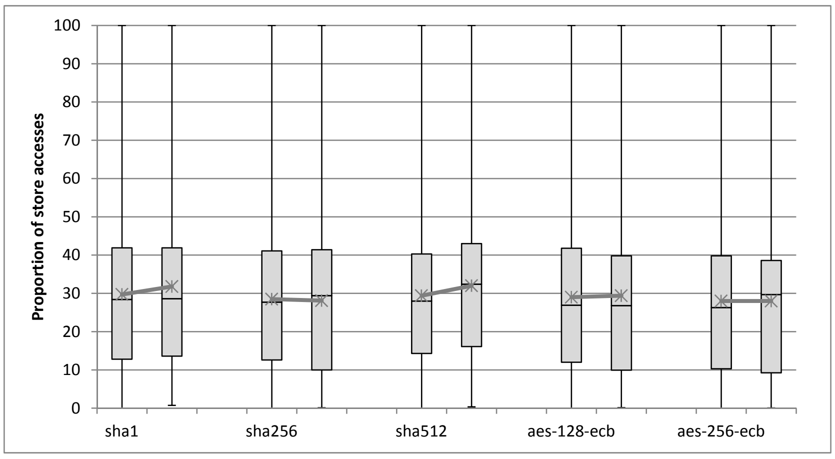
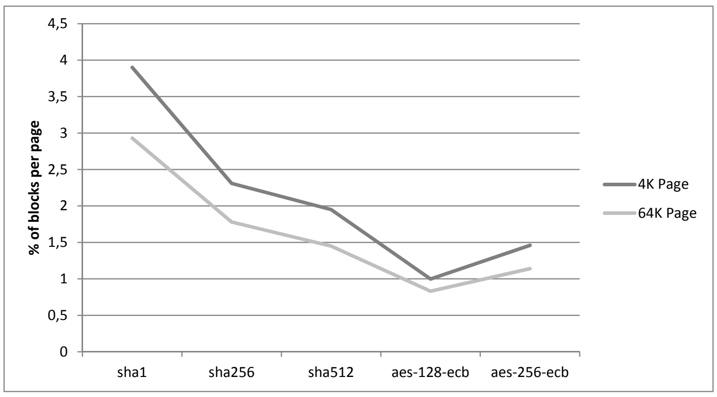
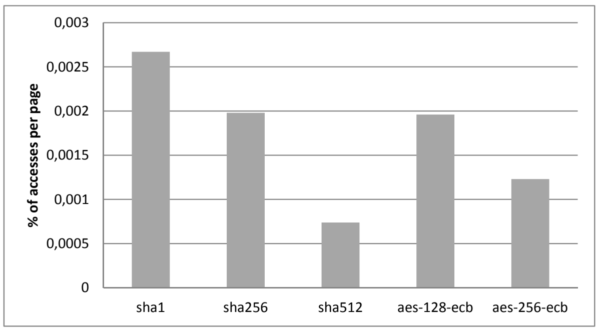


Fig. 4.8: Proportion of store accesses on SW pages.



(a) Percentage of blocks



(b) Percentage of accesses

Fig. 4.9: Proportion of blocks with producer-consumer pattern on pages SW

Chapter 5
Conclusions and Publications

5.1 Conclusions

This dissertation shows the first steps for the design of an optimized coherence protocol for an heterogeneous multicore embedded system as part of the vIrtical project.

It is first presented the coherence problem, some background for coherence models today and, finally, the State of the Art for coherence protocols. We need to analyze the current approaches and their characteristics in order to properly study the suitability of the many techniques to the target system proposed.

It is also showed the elements and characteristics of the proposed system. It is also closely studied the coherence needs of each of these elements in order to aforeseen possible refinements of the coherence protocol of choice.

To end with, it is performed an analysis of applications in order to find sharing patterns that should allow us to evaluate the efficiency of system-level low-cost distributed cache coherency protocols. Indeed, the latter objective is to develop a cache coherence framework that allows the exploration of different cache coherency protocols in order to harmonize them with virtualizations.

It can be seen that, on average, about 60% of the blocks accessed by the analyzed applications correspond to data blocks, which, unlike instruction blocks, may require coherence maintenance. Moreover, despite the fact that it is detected a high sharing degree of data blocks among cores, indeed only SW blocks (just 40% of the total number of data blocks) require coherence. However, SW blocks agglutinate the largest number of accesses (60% on average). That means that the majority of data accesses require coherence. Therefore, its impact in performance can be significant.

In order to ease data classification, different page granularity degrees can be used at the expense of causing a loss of accuracy due to block misclassification. Also, it is observed that SW blocks are not spread over all the pages, but they are indeed distributed between a limited number of pages. In particular, less than 40% of data pages require coherence. Deeping inside SW pages, we observe that they hardly are populated, containing about 30% and 10% of blocks, on average, for 4KB and 64KB pages, respectively. Among them, the number of SW blocks is the majority. Furthermore, about 30% of the accesses to blocks of SW pages correspond to store operations.

Finally, it is observed that the impact of the producer-consumer sharing pattern in the analyzed applications is negligible. Less than 1.5% of the blocks accessed in SW pages present such a behavior, which discourages the application of update policies for coherence purposes in an extensive way.

5.1.1 Discussion

Further discussion must be done in order to decide which can be the best options among all the coherence protocols described in this dissertation, taking into account also the characteristics of the target system of the study.

As aforementioned, the impact of producer-consumer pattern is insignificant and therefore a invalidation-based protocol would be the best choice, which also will reduce the amount of cache-to-cache traffic and consequently reduce the amount of cache bandwidth and energy consumption.

As the target system is an embedded multiprocessor system, it has also real time-related constraints. With the memory access latency as one of the most critical aspects for the RTOS, retrieving a memory block in the less amount of time becomes a key issue. Because of that, the indirection caused by the directory protocols may not be suitable as the appropriate strategy. Nonetheless, one of the main advantages of the directory-based protocols is their scalability when compared to a snoopy protocol. However, in this case we have a system of limited capacity (up to 4 cores for the first level of cache) and with a broadcast-based interconnection, which can make the snoopy-protocols as the best choice.

As seen on the analysis results, we found that for page granularities most blocks and accesses are classified as SW and, therefore, it may not be suitable a technique based on coherence deactivation (Section 2.2.3) at page level. However, with block granularity it is observed that 60% of blocks does not require coherence. Therefore, a fine grain detection inside SW pages may be interesting, at the expense of introducing additional hardware support. That means that in the possible case of applying of coherence deactivation techniques would be mandatory to meet a suitable trade-off between accuracy and introduced overhead.

The main benefits obtained through the use of these techniques are related to the energy reduction deduced from the lowering in coherence messages. It also improves the execution time of the system.

On the other hand, other techniques more focused on snoopy-based protocols and interconexion networks can also be applied. Most filtering techniques seen in Section 2.2.1 are based in reducing energy consumption by means of filtering unnecessary snoop traffic. However it must be taken into account that some of the proposals, as the Subspace Snooping, do the filtering at page level granularity while relying on the OS to do so and therefore, not adding much hardware complexity. As in the coherence deactivation techniques, in this case is observed that most of the accesses will require coherence when classified at page level due to mis-classification. So the improvement may be not much significant compared to do the tracking of the state at block level granularity.

5.2 Publications

This work originated a paper published in the international OMHI Workshop 2012, being it organized in conjunction of the Euro-Par annual series of international conferences dedicated to the promotion and advancement of all aspects of parallel computing. Also a poster and a paper has been published in the ACACES 2012 in Fiuggi, Italy.

- Albert Esteve, María Soler, Maria Engracia Gómez, Antonio Robles, and José Flich. Detecting Sharing Patterns in Industrial Parallel Applications for Embedded Heterogeneous Multicore Systems. *In OMHI Workshop*, Rhodes Island, Aug. 2012

- Albert Esteve, María Soler, Maria Engracia Gómez, Antonio Robles, and José Flich. Memory coherence and compression in the vIrtical Project. *In ACACES'12*, Fiuggi, Italy, 2012

Bibliography

[APJ09a] Niket Agarwal, Li-Shiuan Peh, and Niraj K. Jha. In-network coherence filtering: Snoopy coherence without broadcasts. *In The 42th Annual IEEE/ACM International Symposium on Microarchitecture MICRO'09*, 2009. 23

[APJ09b] Niket Agarwal, Li-Shiuan Peh, and Niraj K. Jha. In-network snoop ordering (INSO): Snoopy coherence on unordered interconnects. *In Proceedings of International Symposium on High Performance Computer Architecture*, 2009. 23

[CLS05] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. *In Proc. of the 32nd Annual International Symposium on Computer Architecture*, June 2005. 16

[CMR$^+$06] Liqun Cheng, Naveen Muralimanohar, Karthik Ramani, Rajeev Balasubramonian, and John B. Carter. Interconnect-aware coherence protocols for chip multiprocessors. *In Proceedings of the 33rd International Symposium on Computer Architecture (ISCA'06)*, 2006. 22

[CRG$^+$11] Blas Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. *In 38th Int'l Symp. on Computer Architecture (ISCA)*, pages 93–104, June 2011. 21, 44, 51

[EPS06] Noel Eisley, Li-Shiuan Peh, and Li Shang. In-network cache coherence. *In The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006. 22

[FLKBF11] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo directory: A scalable directory for many-core systems. *In Proc. of 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 169–180, Feb. 2011. 20

[HDH11] H. Hossain, S. Dwarkadas, and M. C. Huang. POPS: Coherence protocol optimization for both private and shared data. *In 20th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2011. 51

[HFFA09] N. Hardavellas, M. Ferdman, B. Falsa, and A. Ailamaki. Reactive nuca: Near-optimal block placement and replication in distributed caches. *In 36th Int'l Symp. on Computer Architecture (ISCA)*, pages 184–195, June 2009. 44, 51

[KAKH10] Daehoon Kim, Jeonseob Ann, Jaehong Kim, and Jaehyuk Huh. Subspace snooping: Filtering snoops with operating system support. *In the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2010. 15, 44, 51

[KG97] M. B. Kamble and G. Ghose. Analytical energy dissipation models for low power caches. *Proc. Intl. Symposium on Low Power Electronics and Design*, Aug. 1997. 14

[MBH$^+$05] Michael R. Marty, Jesse D. Bingham, Mark D. Hill, Alan J. Hu, Milo M.K. Martin, and David A. Wood. Token coherence: Decoupling performance and correctness. *In Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture (HPCA-11 2005)*, 2005. 24

[MH07] M. Marty and M. Hill. Virtual hierarchies to support server consolidation. *In ISCA-34*, 2007. 25

[MHW03] Milo M. K. Martin, Mark D. Hill, and David A. Wood. Token coherence: Decoupling performance and correctness. *In Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*, 2003. 24

[MMFC00] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. JETTY: Filtering snoops for reduced power consumption in smp servers. *In Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, Jan. 2000. 14

[Mos05] Andreas Moshovos. RegionScout: Exploiting coarse grain sharing in snoop-based coherence. *In Proceedings of the International Symposium on Computer Architecture*, page 234, 2005. 15

[MS09] J. Meng and K. Skadron. Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling. *In ICCD*, Oct. 2009. 21

[PG08] Avadh Patel and Kanad Ghose. Energy-efficient mesi cache coherence with pro-active snoop filtering for multicore microprocessors. *In International Symposium on Low Power Electronics and Design 2008*, 2008. 17

[PSNB10] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian. SWEL: Hardware cache coherence protocols to map shared data onto shared caches. *In 19th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 465–476, Sept. 2010. 21, 51

[RBM08] Arun Raghavan, Colin Blundell, and Milo M. K. Martin. Token tenure: PATCHing token counting using directory-based cache coherence. *In Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 47–58, 2008. 24

[SBL04] Taeweon Suh, Douglas M. Blough, and Hsien-Hsin S. Lee. Supporting cache coherence in heterogeneous multiprocessor systems. *In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'04)*, 2004. 26

[SLB04] Taeweon Suh, Hsien-Hsin S. Lee, and Douglas M. Blough. Integrating cache coherence protocols for heterogeneous multiprocessor systems. *In Proc. of the Conf. on Design, Automation and Test in Europe*, 2004. 26

[Ste90] P Stenström. A survey of cache-coherence schemes for multiprocessors. *IEEE Computer 23*, pages 12–24, June 1990. 5

[YZP09] Chenjie Yu, Xiangrong Zhou, and Peter Petrov. Low-power inter-core communication through cache partitioning in embedded multiprocessors. *In SBCCI 217th: Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design*, pages 1–6, 2009. 16

[ZSD10] Hongzhou Zhao, Arrvindh Shriraman, and Sandhya Dwarkadas. SPACE : Sharing pattern-based directory coherence for multicore scalability. *In MICRO 43: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010. 20

[ZSM07] Jason Zebchuk, Elham Safi, and Andreas Moshovos. A framework for coarse-grain optimizations in the on-chip memory hierarchy. *In 40th IEEE/ACM International Symposium on Microarchitecture*, 2007. 16

[ZSQM09] Jason Zebchuk, Vijayalakshmi Srinivasan, Moinuddin K. Qureshi, and Andreas Moshovos. A tagless coherence directory. *In MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009. 19