THESIS ON INFORMATICS AND SYSTEM ENGINEERING

Tallinn University of Technology

Faculty of Information Technology
Department of Computer Engineering

Master Thesis

Memory Consistency and Cache Coherency in Network-on-Chip Based Multi-Core Systems

Author	:	Radomir Šebek
Advisors	:	Dr. Gert Jervan
		Prof. Dr. Thomas Hollstein
Start	:	01. November 2011
End	:	28. May 2012

Herewith I declare, that I have made the presented thesis myself and solely with the aid of the means permitted by the examination regulations of the Tallinn University of Technology. The literature used is indicated in the bibliography.

I hereby declare that this master's thesis, my original investigation and achievement, submitted for the master's degree at Tallinn University of Technology, has not been submitted for any degree or examination.

Tallinn, June 2012

Radomir Šebek

This thesis was created as a final step in the international master's program in Software Engineering - specialization track of Embedded Real-Time Software Engineering jointly delivered by the two largest public universities in Estonia - University of Tartu and Tallinn University of Technology (TUT).

I would like to show my great appreciation and gratefulness towards effort, guidence and constructive advices given to me by thesis supervisors, Dr. Gert Jervan and Prof. Dr. Thomas Hollstein, from Department of Computer Engineering at TUT. Also, big thank you to Dr. Mohsin Amin and Dr. Mihkel Tagel, research scientists at same department, for persistence in long discussions, creative ideas and programming tips.

Last, but not the least, I would like to acknowledge and express gratefulness towards support my family back in homeland provided me during entire master program.

Abstract

The complexity of modern Systems-on-Chips (SoC) is increasing with technology innovations. Designers of such systems are devoting significant attention not only to computation attributes, but increasingly more and more on communications characteristics. Having in mind scalability challenges, Networks-on-Chip (NoC) are already de facto standard for the communication backbone of SoC systems. As such, those systems are targeting more and more parallel execution of user defined, real-time applications, but the computer engineering society aims at hiding underlying platform specific characteristics and providing user with platform-independent services. Shared memory services are quite often a needed crucial property of such systems, therefore providing a coherent view, ensuring memory consistency, and still achieving the desired performance system characteristics is a huge challenge for scientists nowadays. With the invention of 3D integration, and opportunities of stacking memory modules on top of it, the concept of scalable shared memory will be one of the main memory access concepts besides message passing.

In this thesis, the concept of a scalable coherency protocol which dynamically adopts to inputs of system and shared resources, is presented. Protocol ingredients, structure and internal modules interaction are described in detail. The conceptual idea of this protocol, influenced by widely accepted best practices in bus based systems as well of other NoC systems, is implemented for one particular type of NoC platform - XhiNoC (extendable Hierarchical Network-on Chip). The feasibility of the presented concept for distributed shared memory (DSM) coherency within NoC-based SoC architectures is confirmed by simulation-based experimental results.

Abstrakt

Kaasaegsete kiipsüsteemide (System-on-Chip - SoC) keerulisus suureneb koos tehnoloogia innovatsiooniga. Taoliste süsteemide disainerid peavad pöörama tähelepanu mitte ainult arvutuslikele omadustele vaid üha rohkem ja rohkem ka kiibisisestele ühendustele. Et edukalt toime tulla skaleeritavuse probleemidega on kiipvõrgust (Network-on-Chip - NoC) saanud "de facto" standard kiipsüsteemide sisemiseks kommunikatsioonilahenduseks. Ühelt poolt on selliste süsteemide eesmärgiks teostada üha rohkem kasutaja poolt määratud reaalaja rakenduste paralleeltöötlust, samal ajal on aga disainerite eesmärgiks peita platvormi karakteristikuid ning pakkuda kasutajatele platvormist sõltumatuid teenuseid. Jaotatud mälu on sageli selliste süsteemide vajalik ja oluline omadus. Seetõttu on vajadus pakkuda koherentset vaadet, tagada mälu terviklikkust ning samal ajal saavutada vajalikud jõudluskarakteristikud kõik suured teaduslikud väljakutsed. 3-D integratsiooni- ning mälu virnastamistehnoloogiate arenguga seoses hakkab skaleeritava jaosmälu kontseptsioon koos sõnumiedastusega olema üks põhilistest mälu ligipääsu kontseptsioonidest.

Käesolevas magistritöös on esitatud skaleeritava ja dünaamilise koherentsusprotokolli kontseptsioon. Detailselt on esitatud protokolli komponendid, struktuur ja sisemiste moodulite omavaheline suhtlemine. Protokolli kontseptuaalne idee, mis on mõjutanud siinipõhiste süsteemide ja kiipvõrkude kontseptsioonidest, on realiseeritud ühe konkreetse NoC platformi - XhiNoC (extendable Hierarchical Network on Chip) - näitel. Esitatud kontseptsiooni teostatavus jaotatud hajussmälu koherentsuse tagamiseks NoCi-põhistes SoC arhitektuurides on tõestatud simulatsioonidel põhinevate eksperimentidega.

Used Abbreviations

API	**A**pplication **P**rogramming **I**nterface
CE	**C**omputer **E**ngineering
CPU	**C**entral **P**rocessing **U**nit
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DSM	**D**istributed **S**hared **M**emory
DSP	**D**igital **S**ignal **P**rocessors
ICT	**I**nformation and **C**ommunication **T**echnology
GPU	**G**raphical **P**rocessing **U**nit
LD	**L**ocal **D**irectory
LRU	**L**east **R**ecently **U**sed
MC	**M**emory **C**ontroller
MESI	**M**odified **E**xclusive **S**hared **I**nvalid
MOSFET	**M**etal **O**xide **S**emiconductor **F**ield **Effect** **T**ransistor
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
MSI	**M**odified **S**hared **I**nvalid
NI	**N**etwork **I**nterface
NoC	**N**etwork **o**n **C**hip
PE	**P**rocessing **E**lement
SoC	**S**ystem **o**n **C**hip
TSV	**T**hrough **S**ilicon **V**ias

Contents

List of Figures

List of Tables

1 Motivation

The Great need for certain computing real-time tasks to be processed faster using less power and still fault tolerant in behavior, is pushing innovations and advances in computer society world. In high performance computing areas such as astrophysics, genomics, image and biomedical signal processing, homeland security, automotive industry, simulations and many others ICT (Information and Communication Technology) areas there is a constant demand for better performance. State of the art in cutting edge tech companies dealing with computer systems is the trend that instead of further increasing the processors clock rate, many processors and processing units are simply connected together, forming a high performance computer system. There is a simple reason for that. Following the technology scaling trend in past decade, today's scientists are dealing with 32nm technology with intentions of going to 16nm (or even less) by 2016. Further scaling would not be possible due to physical constraints (very small number of Si atoms in MOSFET channel). Scientists are aware that going towards less nm technologies is not a solution for faster computer systems, and several different research areas are under examination as basic technologies for development of future computer systems. One of them is the concept of Network-on-Chip, also known as NoC.

1.1 Multi core to Many core era transition

On conferences and workshops in Computer Engineering (CE) related areas one can often hear a sentence: "The core is the logic gate of the 21st century!". True. In portable devices and smart phones people are using in daily life, uniprocessor systems are getting out of fashion. During the time interval when this work has been written, state of the art in commercially available mobile devices is based on dual cores architectures (for example iPhone 4S or Samsung galaxy S2 and first quad-core architectures, Samsung Galaxy S3). Desktop Personal Computers(PCs) available on the market today are generally quad core (several lines of popular Intel's i-3, i-5 and i-7). On the other hand, Intel's Polaris 80 cores processor was officially announced on February 11th, 2007 mainly for research purposes but there are no doubts that increasing number of processing cores will become a trend in commercial electronics in time interval that is ahead of us. The number of cores to be integrated on a chip is expected to rise according to [BDK+05], moving from multi-core to many-core architectures. This implies that the scientific community will soon be dealing with computer systems consisting of hundreds of cores on a single chip.

1.2 Communication and power issues

According to the Moore's law, number of transistors integrated on a single chip grew rapidly over the years, making Central Processing Units (CPUs) faster. Memory remained significantly slow viewed from CPU operational speed perspective. The need for frequent communication between the two is obvious. The Issue of communication bottleneck between those two main computer systems ingredients (gap between the speeds) is known as memory wall issue, also known as memory bottleneck.

Another issue in the chip design area is regarding power consumption, also known as power wall problem. As chips are becoming smaller in size and containing more and more transistors (clock rate grows), more power is needed. From equation 1.1, one can clearly see that dynamic power dissipation Pd in logic gates is proportional to clock frequency f and that further clock increase is not realistic scenario for next generation of computer systems since market is demanding for less power consuming systems that can be embedded and will generally use battery as power source. It is a known fact that in state-of-the-art smart phones, one battery charging is just enough for one day of usage for average not so demanding user. The industry giants are trying to increase this period by reducing power consumption. It would be great if this time frame could be increased for few more days, wouldn't it?

$$Pd = C \times Vdd^2 \times f \tag{1.1}$$

At the present moment, a commonly acknowledged fact is that power wall in terms of uniprocessor chips is hit, even some prevention attempts from hitting the power

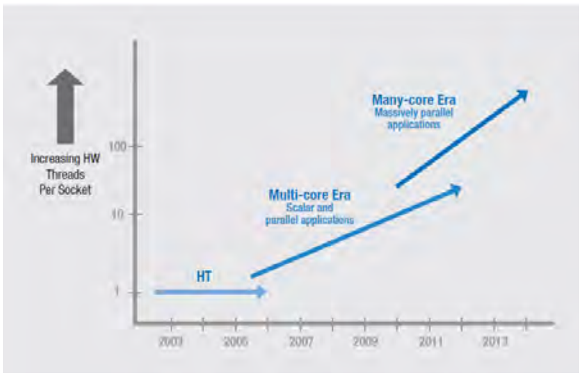


Figure 1.1: Multi to many core transition. Source: [BDK+05]

wall are visible [MJ08]. In this thesis there will be no more elaboration on this topic, but it is important to mention it as one of main reasons for switching to multi-core architectures, since scientific community is aware that MPSoCs (Multi Processor System on Chips) can be less power hungry.

1.3 Network-on-chip paradigm

A good article, which is recommended to the reader, is one by [BM06]. Traditional bus based systems, including hierarchical bus based systems are obviously not scalable enough and applying them in high performance computing area is not likely to happen in future architectures. Since the number of processing elements and different Intellectual Property (IP) components that computer architecture designer wants to attach to bus based systems grows, one needs to accept certain increase in delays, communication overload and decrease in performance. Accent in bus based architectures is on computation. Observed from the context of network-on-chip (NoC) as a mean of communication for scalable MP(Multi Processor) systems-on-chips, the emphasis in design is not only on on computation, but equal on communication.[1]. Also, in terms of optimization of bus based systems, engineers tried to improve performance by increasing the amount of work performed in each cycle which is known as ILP (instruction level parallelism), while on the other hand, real power of parallelism in NoC systems is exploited via principle of TLP (thread level parallelism). NoCs represent a communication subsystem separate from resources in a way that NoC is operationally independent from PE (Processing Element) connected to NoCs nodes. The NoC operates in its own, predefined frequency and all PE "plugged in" have to adapt to NoC protocol via Network Interfaces. PEs can operate in their own clock frequencies with respect to GALS principle (Globally Asynchronous Locally Synchronous). Systems-on-chip based on the NoC paradigm are especially suited for running multi-threaded applications.

During the last decade, many NoC research platforms were developed for scientific purposes by different research groups, and in the last few years commercial platforms are also starting to be present on the market.

During experimental part of this thesis, XHiNoC (extendable Hierarchical Network-on-Chip) the NoC platform was used (Samman, Hollstein, & Glesner developed in 2008), and more details about the concepts of this NoC architecture can be examined in [Sam10] and publications referenced in that PhD thesis by scientists from TU

[1]By NoC systems in future text author is referring to MPSoC systems based on NoC as communication medium

Darmstadt. In this thesis, deeper explanations of all properties of this NoC platform, s topology, switching method, routing mechanism, flow control, router architecture, quality of services and similar are not present, since it is outside of the scope of this thesis.

1.4 Research in the Field of High-Performance Computing

Parallel computer is nothing more than a collection of usually heterogeneous processing elements that cooperate and communicate to solve large problems. Performance matters and the general idea is to have very fast parallel computer systems. In real-time systems, where parallel computer systems are being used, time constraint plays crucial role. Hard real-time systems are for example defined as such that delivering computational answer is strictly not allowed outside a specified and predefined time interval. When processing critical cyber-physical systems, delivering an answer late can cause serious consequences. It is of highest importance that parallel systems behave as expected from the computational as well as from communicational side. Here there is a need to clearly understand the concept of what **happens-before** relationship.[2] What does it mean from programmer's perspective? Programmers need to take care about parallelism. Sequential execution of code is what majority of programmers are used to, but several years back this practice started to change. Let us examine part of typical C code for parallel programs:

```
cnt=0;
for (i=0; i=very_big_number; i++) {
  pthread_mutex_lock(&mutex);
  cnt++;
  pthread_mutex_unlock(&mutex);
}
```

Programmers are aware that in multi-processor platforms where code is going to be executed, several threads (we assume here that each thread is executed on different core) are going to be involved in this calculation in order to generate faster results, variable cnt is shared, and each local cache of the involved CPUs will have copies of this variable in order to process it. If the programmer does not protect critical section with mutexes - counter in this case, an unexpected result will occur. Cache controllers (cache coherence mechanisms) are going to invalidate or update value

[2]By closer look at [Ora] from where happens-before keyword origin is, terms becomes clear and reading is helpful also to understand how Java programmers are dealing with consistency issues

of this variable in local cores caches and READ and WRITE operation to same variable originally stored in main system memory, is going to be performed many times. Looking deeper into the assembly implementation of simple increment of cnt, it is obvious that three operations need to be realized: LOAD cnt, INCREMENT cnt and STORE cnt. If not protected with the mutexes, realization of code becomes chaotic, some cache will fetch value of cnt from the main memory while in the same instance some other node tries to write to it, etc. As a result one would have cnt significantly less than very big number variable. The concept of mutexes is purely clear and simple, but programmers need to keep this and similar concepts in mind while dealing with programs for multiprocessors systems. Such concepts were not generally used while dealing with uniprocessor system, except in a scenario with several threads running on a single core, are sharing the same address space. In the Java world the thread interference and memory consistency errors are still giving developers sleepless nights. By declaring variable as *volatile*, programmer is sure that READ and WRITE operations to all types of primitive variables in the code will be executed as atomic. Yes, this helps a lot in fighting with the thread interference issue. Regarding memory block consistency views, there is a keyword *synchronized* which applies both to methods and statements.

Programming of parallel systems is not a trivial task. Few languages are widely used today for programming for multi-core systems like OpenMP, MPI, etc. The computer engineering community is going into direction of trying to hide platform specific implementation from end users. MOSART [mos] is European Commission funded program that among other issues addresses: "The difficulties in programming heterogeneous, multi-core platforms, in particular in dynamically managing data structures in distributed memory."[CAS+10]. Here the idea is to create middle-ware services that will hide platform architecture from APIs (Application Programming Interfaces) that run on top of it. This will require a global rethinking of software and hardware design principles. HiPEAC is open European Network of Excellence on High Performance and Embedded Architecture and Compilation which monitors and increases European research in the area of high-performance and embedded computing systems, [hip] and the author of the thesis is looking forward to future times when personal professional contribution plans to these networks will become true.

Curious reader by now might be wondering: Does it mean that same job that one used to do on uniprocessor platform now can be processed 4 times faster on quad-core architecture? Well, not necessarily. This depends on the nature of program code that is being executed on quad core platform. In simple words, if the program is strongly sequential and can't be partitioned in separate execution modules that do not depend on previous modules results, a significant speedup cannot be achieved. That is why programmers nowadays need to change the way of thinking sequentially, (which is most natural way of thinking while writing a code) and think little bit in terms of underlying platform. Back in 1967 Gene Amdahl announced an equation nowadays known as Amdahl's law, which specifies the possible speedup as a result of parallelization. From equation 1.2 where N represents number of processors, S describes percentage of serial program code one can conclude that the speedup will always be less then number of available cores.

$$Speedup = \frac{N}{S \times N + 1 - S} \tag{1.2}$$

A speedup analysis was also done by the scientists Finnigan and Gustafson, but all three of them do not consider communication overhead that is present due to the fact that threads sharing same address space (looking from uniprocessors perspective) or cores (looking from multi-core distributed shared memory perspective) frequently need to exchange informations. Stone in [Sto93] presented a metric that takes into consideration computation time R and communication time C like described in equation 1.3, where M stands for number of jobs (tasks) and N stands for number of available processors in underlying platform where task are executed.

$$Speedup = \frac{N \times (\frac{R}{C})}{\frac{R}{C} + \frac{M \times (N-1)}{2}} \tag{1.3}$$

By now, it is clear that multi- and many-core architectures will only provide more speed when used with multi-threaded software and regarding this thought, authors of [LC09] have in medias res questions that CE community is still struggling to answer: "First, in terms of computer architecture, the question is: How can we build interconnect systems that minimize the latency of communications between processors? And second, in terms of parallel algorithm design, the question is: How can we partition our problems such that we can minimize the communications between processes?". Yes indeed, first thing we should have in mind is to have smart application mapping of our programs and services on top of multi-core hardware platform, and this will in essence decrease communication overhead between the tiles that we would have otherwise. In addition, one can also try to find inspiration in idea of hiding latencies of assessing slower part of system memory.

2 Introduction and background

2.1 Memory Hierarchy

Every processing element, connected to a NoC, consists of at least one CPU core which has several layers of caches. With the increase of the number of cores in a single CPU the number of cache levels might increase in the future. The scope of this thesis is examining network memory hierarchy rather than internal PE memory hierarchy (registers, caches, main memory, etc). Having this in mind, main memory of entire SoC under our 2D target architecture is distributed DRAM modules connected to nodes in the NoC. In terms of 3D target architecture and memory stacking possibilities it is assumed that memory is stacked on chip, being used as distributed local or shared memory.

2.1.1 Memory organization in NoC based 2D MPSoCs

The latency of memory operations between several entities within NoC as communication backbone, for example between processing elements and local memory, distant processing elements, processing element and distant memory blocks,etc is the key factor that is being examined in context of MPSoCs performance. One can make classification of NoCs based systems as homogeneous and heterogeneous. If all processing elements connected to NoC are identical, such system is *homogeneous*. On the other side, if processing elements are different (CPU, DSP, GPU ...) such systems are known as *heterogeneous*. Within the scope of this thesis the author assumes that target architecture (the same will apply to 3D MPSoCs) is heterogeneous.

Other nodes (processing unit nodes) have local caches that contain copies of main memory blocks. Caches are realized in several levels usually L1, L2 and L3 depending on processor architecture. Trend in modern multiprocessor design is to have the last level of cache shared between local processors cores (for example in case of Intel Core i7 64-bit x86-64 processor where L3 cache is shared between four cores, but it does not have to be the rule). [1] In essence, starting from main system memory which is measured in GBs of data, until CPUs register which are capable of containing limited

[1]Some Intel processors with L3 shared cache between cores allow disabling it

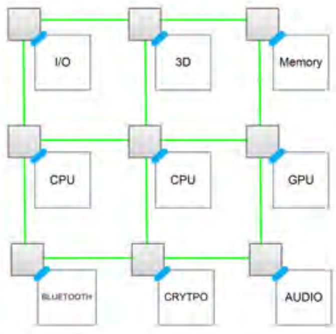


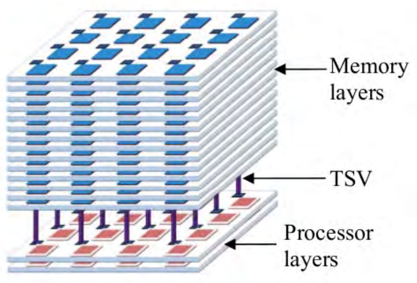
Figure 2.1: 3x3 fully connected mesh heterogeneous target 2D NoC system

size data, all memory blocks in memory hierarchy actually can be seen as caches of bigger block precedents. The L3 cache contains a certain number of exact same copies of the memory block lines from the main memory, the L2 cache is doing the similar job viewed from perspective of the L3 cache etc. Looking back in time, computer engineers tried to solve the memory bottleneck problem by introducing yet another block in between of existing memory blocks in global hierarchy.

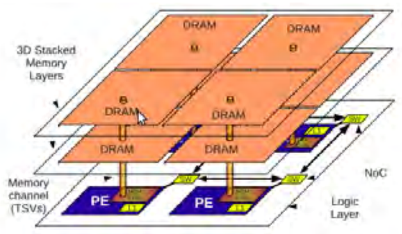
2.1.2 Memory organization in NoC based 3D MPSoCs

The entire memory structure is on chip. Fast accessible via TSV (Through Silicon Vias) comparing to accessing time of off-chip memory. Several memory layers can be stacked on top of one or more logic layers, like in Figure 2.2a. Target architecture under examination in this thesis is assumed to consist of single logic layer and single memory layer accessible via TSVs.

Multiple layers of memory modules are vertically stacked on top of logic layer, using



(a) Dance hall architecture. Source: [WLWT09]



(b) Target 3D architecture. Source: [LB10]

Figure 2.2: Target 3D architectures

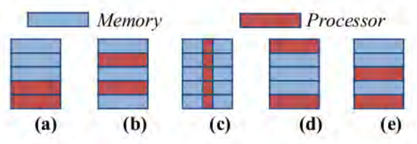


Figure 2.3: 3-D memory architectures. Source: [WLWT09] Dance hall (a), Sandwich (b), Per layer (c), Terminal (d), Mixed (e)

TSVs as communication medium. Nowadays we speak about several millions of TSVs in one square centimeter. State of the art in TSV manufacturing process according to [Sem07] is that the industry is able to produce approximately 4 micro meter square vias on a pitch of 4 micro meters in 2011. Generally, with 3D technology there exists a possibility of integrating DRAMs into a single, tightly-coupled chip stack.

Every node (PE element) contains memory controller and it plays two important roles. On one side, it has to serve all memory operation requests coming from local PE computation unit, and from the other side it also has to serve memory requests coming from other PEs through NoC NI it is connected to. If some distant PE unit (from second row in Figure 2.2 from example) needs certain memory block from first PE unit stacked memory (first row in 2.2) than request must come trough NoC. If we imagine many core architecture with hundreds of PEs connected to the NoC, than **keeping event ordering among memory operations and assuring memory and data consistency is really a challenging task**, and aim of this thesis is to contribute to this challenge.

In the scope of this thesis, focus is on request operation that nodes(PE elements) will introduce towards shared memory regions, distributed homogeneously over NoC platform, and requests performed on local, non-shared memory are not in focus. In further text, one can always assume that once memory controller is mentioned, it refers to memory controller that operates on shared memory regions. Also it is important to mention that, XHiNoC simulator, or platform itself, that is our main and only simulator and platform used during time frame of this thesis, have potential of 3D integration, but main focus was on 2D NoC platform. Author's future research directions are to tackle memory related challenges that 3D XHiNoC will bring.

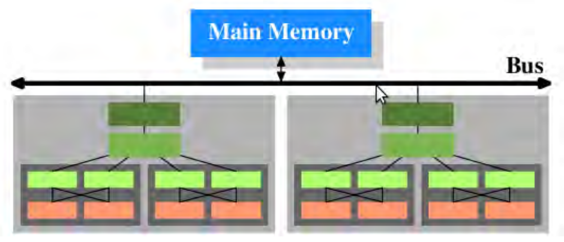


Figure 2.4: Simplified bus based architecture. Source: [Dre07]

2.2 Coherency

2.2.1 Cache coherency in bus based architectures

Most modern processors with underlying bus based architectures can be simply represented as on the Figure 2.4. Main memory stores data and local caches contain copies of certain memory blocks from it. Normally local caches will have a need to issue READ or WRITE operation to main memory and since only single entity can use bus at one instance of time, there must exist some kind of arbitration unit, which will assign mastership over the bus. Central task is to maintain coherency and event ordering (memory and data consistency topic will be discussed in 2.3) in such systems. Imagine the case where for example two local caches of two processors (P1 and P2) fetch the same data from main memory and P1 change the value of certain cache line (we assume that cache sizes are same in both processors for simplicity reasons). Naturally, this change should reflect main memory where original data is stored. However, it must not be allowed for the second processor to continue operations on that shared memory line, since it has been changed by P1. Therefore, existence of cache coherence protocols is a must. How is it solved in bus based systems? Widely used approach is **snooping**. Cache controllers (as many as there are) have the possibility to snoop (monitor) fluctuation of memory operations on the bus at the same time, since cache controllers are directly connected to the bus. Accordingly, they either update or invalidate cache lines in local caches and what has just been described is known as *write-through* and *write-back* cache coherence mechanisms. Write-through policy is very intuitive and rather strict. As soon as cache line is modified in the local cache, update is propagated further toward the main storage. On the other hand, idea of write-back writing approach is to mark cache line in the local cache as *dirty* once it has been modified. Change will not be propagated immediately to main system memory, it will wait until replacement of cache content occurs. Typical and very efficient replacement policy is LRU (*L*east *R*ecently *U*sed) that will in advance of arrival of new cache line discard least recently used cache line. Meaning, while discarding cache line from local memory, if line is marked as dirty, cache controller will know that it needs to be written to main storage as well. Write back mechanism is more represented in modern architectures.

Central question is how will rest of the processors in interconnection network be aware that shared cache line is dirty in other processors? This information must be delivered to all relevant nodes in real time, since no further operations (READ or WRITE) should be allowed on shared cache block marked as dirty until it gets update of latest values.

2.2.2 Coherency in NoC based systems

Cache coherence actions increase traffic overhead on shared bus. When the number of CPUs connected to bus becomes bigger than just a few, snooping solution is not applicable any more. In Network-on-Chip systems cache coherence cannot be implemented using snooping.[2] Snooping is not applicable for coherency of distributed shared memories. Let us take analogy from real life. In a ten floor building with single entrance and circular stairs in the middle of the building, if one stands on top of stairs, one can easily monitor when someone enters the building and on which floor he stopped. On the other hand, if we now imagine several of identical buildings, connected via bridges for example, one cannot monitor all entrances at the same time and collect all information at once.

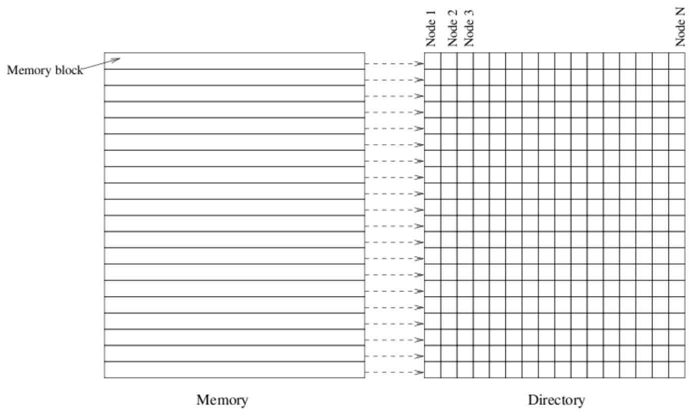


Figure 2.5: Directory based protocol "idea". Source:[SJ11]

Commonly accepted solution for cache coherency of large scale multi-core systems is *directory* based protocol. As one can see from Figure 2.5 each node keeps track of all nodes that have copies of certain memory block from it. Every moment, when

[2] However, principles of snooping are applicable in NoCs terms

some node changes value of certain memory block, protocol knows which nodes should receive invalidation or update information. To be more precise, here full-map directory based protocols are referred, where every single sharer keeps track with whom particular memory block is being shared. As you can already conclude, this approach is just not good enough in terms of large scale MPSoCs, since it introduces large memory overhead in a system. Another disadvantage, besides memory overhead, are long transaction delay.

2.2.3 MESI protocol - general idea

Many protocols were developed and upgraded over the years. In the beginning, MSI protocol (Modified Shared Invalid)represented very efficient and good enough solution for coherence problem. More comprehensive and specific solutions were proposed as improvements of MSI protocol and in this section intention is to examine MESI protocol, having in mind computer system high level organization as in Figure 2.6.

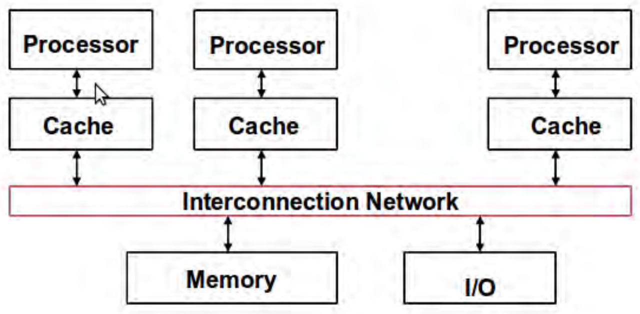


Figure 2.6: Simplified MPSoC based architecture. Source: Benini Luca.

Name of the protocol comes from the four states a cache line can be in when using the MESI protocol (Modified, Exclusive, Shared, Invalid).

- **Modified - M**: cache line has been modified by local CPU core, and it is known as *dirty.* [3] All sharer copies, if any, which have this particular value were in advance of this action invalidated, and if ever needed for processing, they will have to fetch value again,

[3]By dirty cache line referred is cache line which content has been changed comparing to original value in main system memory (distributed shared memory) and in general case it is required to write the line back to memory at some point in the future, before permitting any other READ from main memory on that specific line

- **Exclusive - E**: cache line is unmodified from main memory perspective, it is *clean*[4] and present only in this particular CPU cache, nowhere else,
- **Shared - S**: cache line is shared between at least two cores and it is *clean*
- **Invalid - I**: cache line is invalid since some other node, that has the same copy, modified that copy and invalidated its value.

In bus based systems, it was such a case that certain operations that processor performed, regarding cache/memory handling, were announced on external pins, so outsiders cache/memory controllers can get this information. In NoC context, local directory structures are used to keep track of what changed where and how, so this allows to always know from where one can get latest valid desired memory block content.

It is allowed to execute a READ from any state except Invalid. An Invalid cache line must first be updated. If executing READ operation from M or E state only source and destination nodes local directories will be updated with latest happening. If one performs it from S state, all relevant sharers, as well, need to be informed about this action.

A WRITE operation can occur if the cache line is in the Modified or Exclusive state, similarly as for READ. If in the Shared state, all other nodes' cached copies must be invalidated first. This is typically done by a broadcast operation known as Request For Ownership (check the diagram path which leads to RWITM (Read With Intention To Modify) in appendix A). A WRITE operation is not allowed from Invalid state. Other possible transitions are shown in Figure 2.7. They show which state will

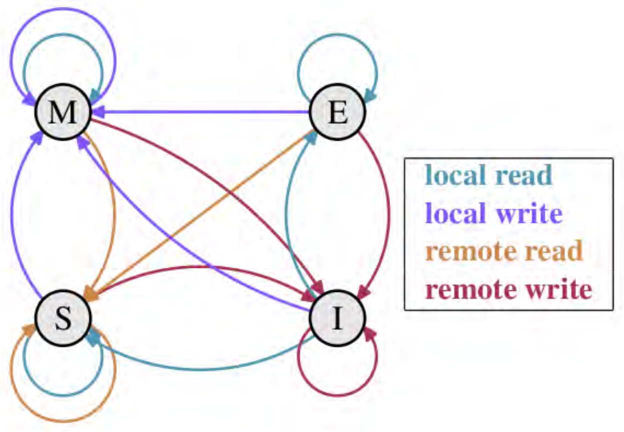


Figure 2.7: MESI protocol transitions. Source:[Dre07]

be the next logical state after certain local or remote operation has been executed.

[4]By clean cache line here referred is such one that corresponds to the one in main system memory

Generally, transitions caused by memory operations coming from local CPU core are less expensive comparing to ones caused remotely.

Other protocols also exist (MOESI, MERSI, MESIF, Firefly protocol, etc) but they are out of the scope of this thesis.

2.3 Memory consistency

In a way, one can look at the coherence protocols as means of implementation of consistency models. Cache coherence protocol aims at updating every relevant node that contains block of shared memory with latest values of that particular block which can be changed in any of those relevant nodes that share the same memory block. Preserving memory operations order in MPSoCs is a highly important task and different memory consistency models have been developed until now. Early work in this field [Gha95] elaborates memory consistency problem in general. From one side there is a *strict* memory consistency model with following main idea: Once something is done, inform everybody before doing the next step. On the other side, *relaxed* consistency models are proposed. Further division of relaxed consistency models is on *weak* and *release*. Idea behind relaxed memory models is to allow the programmer to specify when relaxation is possible by reordering some of the memory operations and specifying when to synchronize all events. In essence, intention of relaxed models is to allow either software or hardware optimization in the system in order to gain more performance. The disadvantage of strict memory consistency model is that it disable us to make hardware optimizations (write buffers for example) as well as optimizations on compiler level (for instance, if you have something like following in your code

```
asm volatile("" ::: "memory location");
```

that would forbid GCC compiler to reorder read and write commands around that memory segment). There is a great thought on this topic by authors of [Gha95]:"The *illusion* of sequentiality can be maintained by only preserving the sequential order among memory operations to the same location".

To ensure correct behavior regarding release consistency authors of [PGG06] propose:

- " Request and responses packets issued by a given initiator to the same target are delivered in-order. This ensures the sequentiality of memory access for a processor. This can be done either by a NoC with FIFO behavior, or by having an initiator wait for the response of packet i prior to send the request of packet i + 1,

- If a given initiator needs to send requests to two different targets, then it awaits the response from the first target prior to send the request to the second target.
- the interconnect is not allowed to arbitrary drop packets when it cannot handle them. "

Application that is executed on NoC has to take care of certain re-sending actions/requests, if for example coherency protocol was not able or did not allow the desired request at specific instance of time. (This will happen rather rarely in case of 2x2 NoC but in larger scale NoC will not be that rare.)

In further discussion it is assumed that underlying NoC must have all these services available.

3 Concept of assuring memory and data consistency

3.1 Vision

In this thesis, heterogeneous Multi Processor System-on-Chip (MPSoC) architecture with Distributed Shared Memory (DSM) modules and local memory modules in each node is assumed. Concept of DSM is illustrated on Figure 3.1. Each node can be loaded with tasks differently, and percentage of total processing job to be done vary from node to node, depending on its performance and strength characteristics. Single shared address space is distributed over multiple physical memory entities and accessible to all processing elements in NoC.

In short, question that this thesis demonstrates and contributes to is how to guarantee that on every memory request in our MPSoC, received memory block value will be coherent. *It is a must, to always have in mind that action that will lead towards incoherent memory sharing or memory usage should not be allowed*! At the same time, system performance must be taken into consideration, meaning that designer needs to be aware how performance expensive is protocol. As mentioned before, directory based protocols are commonly proposed solution to this issue. Within this focus, designers also have to be careful, not to inject unnecessary traffic into NoC.

This thesis provides contributions on alternatives of full map directory based protocols that in general solve issues of memory consistency and memory coherency in MPSoC with NoC as underlying backbone, but are not scalable. They represent static solution and memory usage reduction improvements are noticeable. Scalable NoC memory coherency solution is still an open problem since general aim is to maintain (or speedup) solid (good enough) overall system performance, not only to guarantee memory and data coherency. The main drawback of traditional solutions (full map, chained, and similar flat distributed directory based protocols) are identified to be: memory overhead and long latencies (communication delays). This is point where *smart management* of local directory structures and choosing of appropriate sharer from list of sharers in protocol is important. By optimizing already proposed full map distributed directory based coherence protocols, it is realistic to expect reduction of memory overhead and introduction of less communication messages in this particular

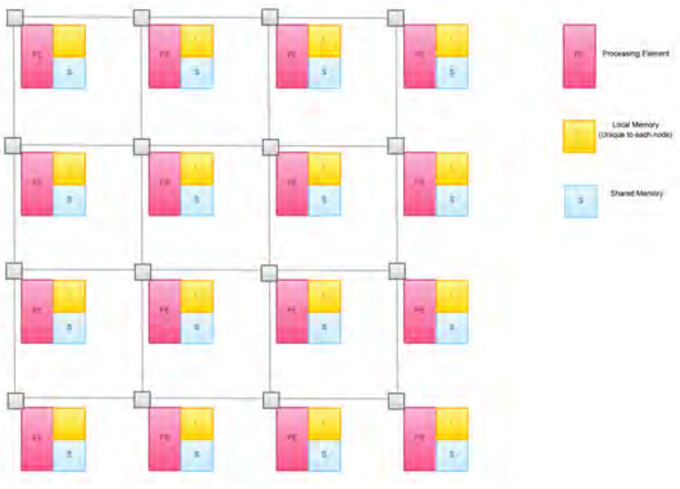


Figure 3.1: Distributed Shared Memory

protocol.

3.1.1 Architecture and Example of assumed application data mapping

Let us imagine some image processing application that runs on 2x2 NoC [1] and architecture of single node is represented on following figure:

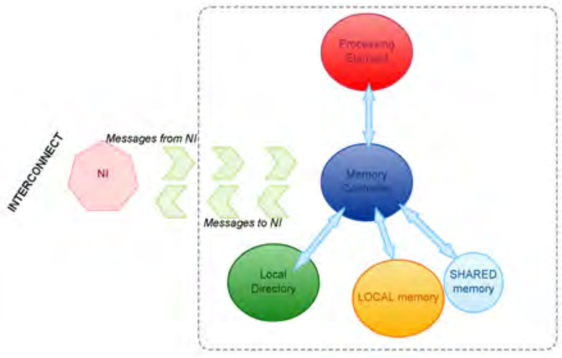


Figure 3.2: High Level Architectural Concept

Meaning, every node will have LOCAL memory and SHARED memory. Local memory will be collaborator while processing/performing largest part of the work,

[1]Here simple NoC is selected for demonstration purposes, but idea is general and can be applied to any NoC size

and this type of operation will not introduce/inject any messages into NoC. *Processing on shared regions will require cooperation between nodes and sharing resources, so memory coherency protocol is highly needed to guaranty coherent view on shared memory and injection of coherency protocol messages into NoC is unavoidable.*

Now let us consider for example an image that needs to be processed. Internal processing logic and used algorithms are out of scope and not relevant here. Let us split image in four segments since 2x2 NoC platform is used. Here, it is chosen to split the image homogeneously for simplicity reasons, but one could split it in any other heterogeneously way since it is assumed that underlying nodes are with different characteristics.

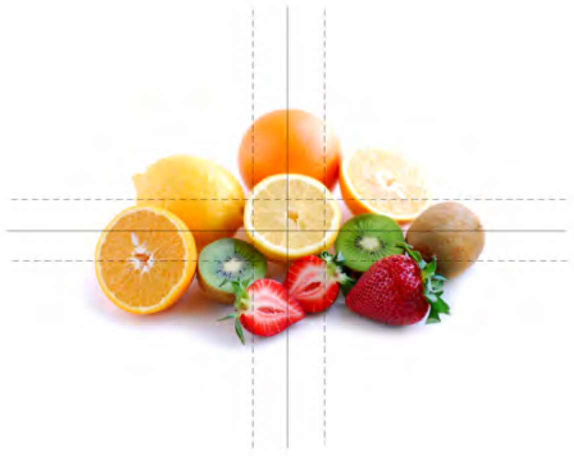


Figure 3.3: Image to be processed partitioned in regions

Tasks will be mapped/distributed as such that every node will receive part of the image to process. Assignment of processing those image parts that can be processed independently by one node, every node will do in collaboration with its LOCAL memory (00, 01,10 and 11 LOCAL). Majority of the job is done in previously mentioned manner - yellow colored regions on Figure 3.4. On the intersections, nodes will be working on the same part of image and there is a need of sharing resources and coherency. It is important to mention that application is mapped as such that for very small percentage of the entire image there is a need to be processed with several nodes (intersections/collaboration segment on Figure 3.4). In order to be processed properly, central part of image requires collaboration of all nodes, symbolically marked as 'ALL' in same Figure. Having this in mind, coherency protocol will inject additional overhead into NoC (looking form perspective of entire job that needs to be done, this percentage is reasonably small) and this is the only reasonable way in terms of scalability to enable functionality of coherence protocol.

What is interesting at this point is to observe here is how shared data is mapped

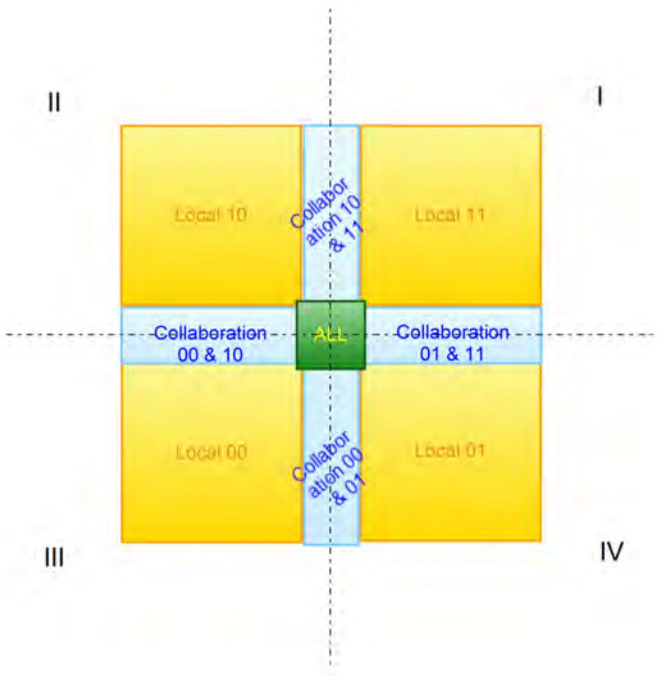
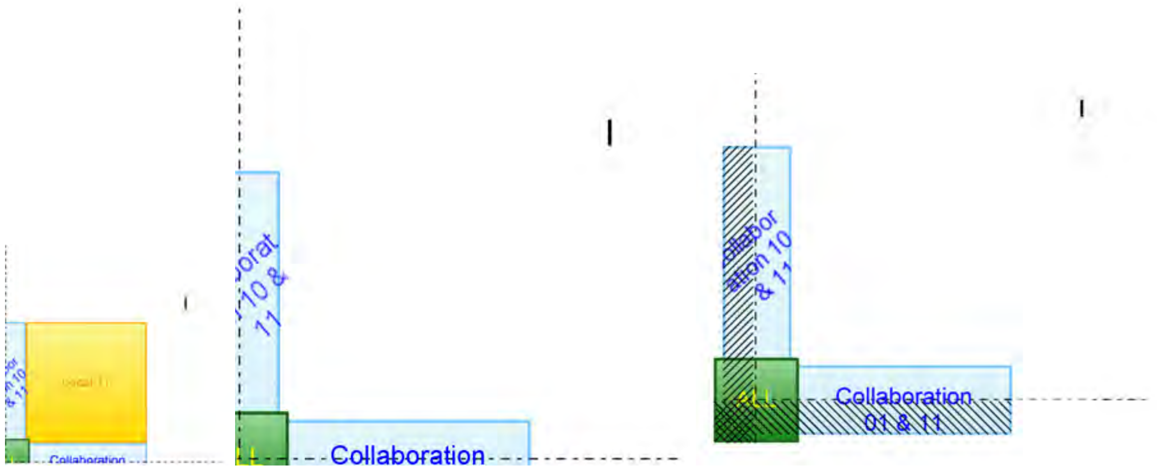


Figure 3.4: Distribution of tasks within processing job from data-memory mapping perspective

into DSM (Distributed Shared Memory). How data is mapped into local memory blocks is not interesting within this thesis scope, and from now on it will not be depicted on diagrams and figures, even it is assumed (meaning - disregard yellow sections from now-on). Every node will store certain portion of shared data in its shared memory blocks, accessible to all other nodes via NoC. While collaborating, nodes will come into situation that they share same copy of data, needed mutually. As one can see from Figure 3.5, segment b), *node 11* will have data in its shared memory blocks (that is part of global DSM) mapped as such that:

- some of those memory blocks represent data that will be used by *node 11* but also node *node 01*
- some of those memory blocks represent data that will be used by *node 11* but also node *node 10*
- some of those memory blocks represent data that will be used by *node 11* but also all other 3 nodes

This is why there will be a need for node operation that will read data from another node (READ in future text) as well as this is why there will be a need for informing sharers about certain changes node made on shared data, or asking sharers for



(a) Initial mapping of shared & local data for node 11 (b) Initial mapping of shared data for node 11 (c) Initial mapping of shared data for node 11 with info about sharers (part marked with parallel lines)

Figure 3.5: Mapping idea of data into memory(part marked with parallel lines/hachures)

permission to do something on top of the shared data (idea of WRITE operation in future text).

Part of the same Figure, section c) illustrate *initialization* idea from single node perspective, meaning illustrates what information besides shared data is being mapped to *node 11*. The idea is same for all other nodes. Segment that is crossed with parallel lines, actually that shared data represented with this lines, is not mapped in *node 11*, but crucial is fact that information about sharers of those specific memory blocks is encoded at the very beginning. In conclusion, every node is aware with whom certain memory blocks is shared, and this information is recorded in nodes local directory.

Now, let us focus on allowed operations. **The difference is that when READ operation is under elaboration, in context of our thesis, it only makes sense to READ from the foreign nodes, but not from the caller node. Why: if the processor wants to Read shared Data from the local memory and it is available and valid, then everything will be o.k. as well. In case of WRITE operation, it makes sense to always WRITE only to the caller node, never to the foreign node, but other relevant nodes should always be informed and asked for acknowledgement prior this action**.

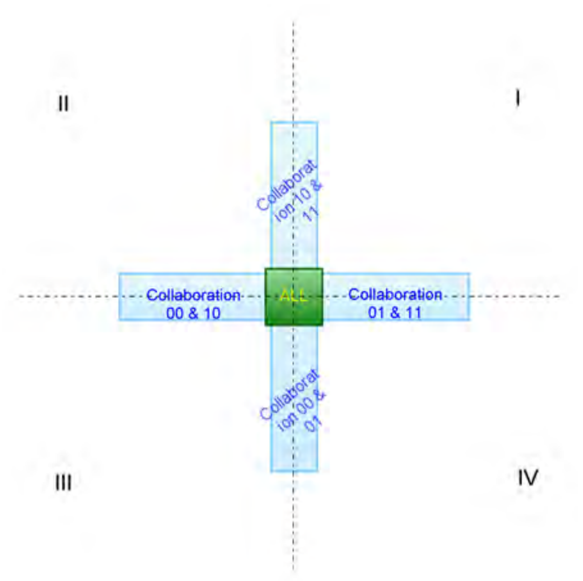


Figure 3.6: Shared memory interactions

3.1.2 Logic

Assuming that high level idea is clear, as well as necessity of such a concept, let us step one level down and see which modules are needed and under which assumptions, so that such a services can be guaranteed.

As stated before, coherence protocol is a designers' toolbox of implementing consistency models. Main entities in coherency protocol are Memory Controller (MC) and Local Directory (LD). Main focus is on MC unit that controls requests on shared memory regions, and as stated in previous chapters, one can think of it as of MC for shared memory. As visually represented on Figure 3.7, MC communicates both ways with Network Interface (NI). Messages (flits) that are available to specific node via NI will be delivered to MC in the same order as NI received them. MC contains internal logic that interprets incoming messages. While interpreting messages MC collaborates and accesses its own LD as well as SHARED memory that is physically stored in that specific node. This collaboration and internal logic will result in majority of cases in a way that MC needs to inject messages into NoC traffic. Messages will be injected via NI in the same order as they have been created in the MC.

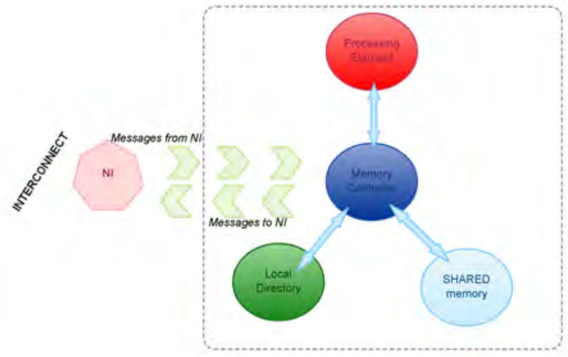


Figure 3.7: Conceptual collaboration diagram between entities in coherency protocol

General concept observed from perspective of single NoC node is represented on Figure 3.7 and Figure 3.8 represents conceptual view from the perspective of an entire NoC examined in this thesis, which is, 2x2 NoC with mesh topology.

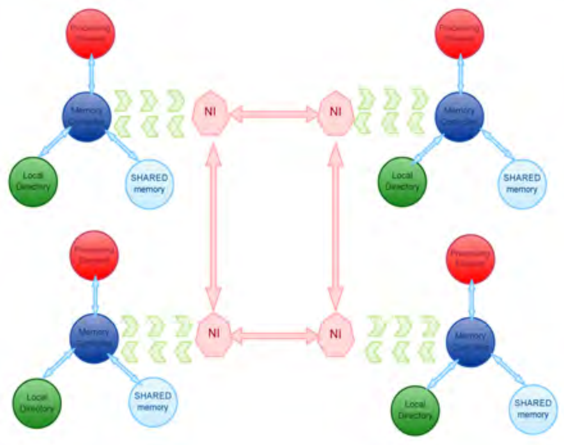


Figure 3.8: Conceptual diagram between entities in coherency protocol in 2x2 NoC

As represented on Figure 3.9 possible states that memory block can be found at are *Invalid*, due to write operation invalidation or *Valid*, that typically happens due to read operation or write operation.

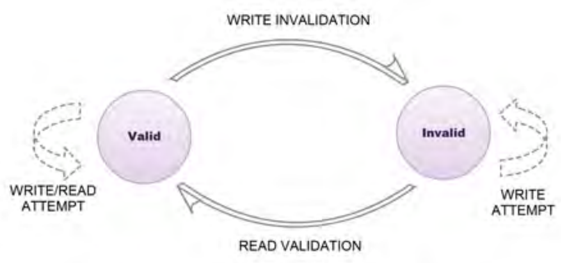


Figure 3.9: Finite State Diagram of shared memory block/line

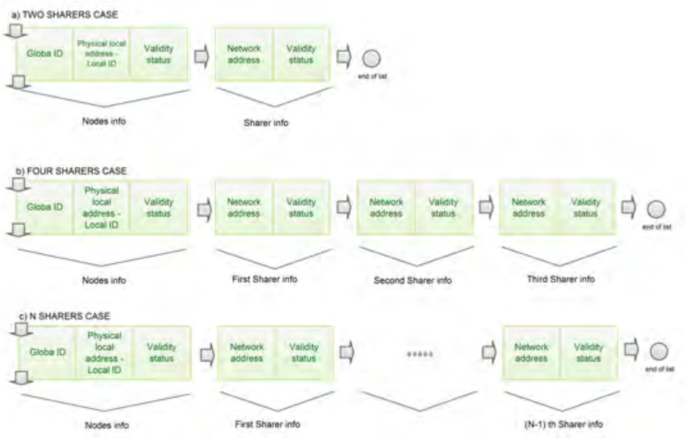


Figure 3.10: Local Directory structure idea for single shared memory block

Represented on Figure 3.10, idea of Local Directory structure per single entry is provided. In general, it is sufficient to elaborate diagram c). The structure represents a list of n connected elements. First element always contains data about shared memory block virtual address (unique for all nodes),local physical representation of that memory block(length), and validity flag/status. All other elements are based on same concept. They provide information in which node same shared memory block is used - network address field, and its validity status - second field. Looking vertically, there exist pointers to entry before or to entry after, and point here is that as much shared memory blocks one node is involved in, that much entries will be created and connected also as a list. This 'vertical' connection will allow programmer to manipulate and iterate big, entire double linked list, also known as LD.

In the following sections demonstration of concrete illustrative examples (concrete scenarios) how coherence protocol should look like from algorithmic programmer's perspective is presented. Every scenario has short introduction that specifies pre-conditions and setup at the beginning. To understand the use cases easier please have ideas of shared memory interactions between the nodes in network illustrated on Figure 3.6, even though in following scenarios nodes are enumerated according to

XHiNoC standard, not as above, Figure 3.10 and Figure 3.9.

3.2 Scenario 1. READ operation, two sharers

Let us examine following use case PE doesn't want to read from PE01, it just accesses the local memory controller, which detects, that the locally available shared data under this virtual address is invalid and the data has to be read from the memory of PE01; elaborated on Figure 3.11. Suppose that *PE 00* wants to read specific available shared memory block value from *PE 01*, or to be fully correct, shared memory block value is stored within *node's 01* shared memory space.

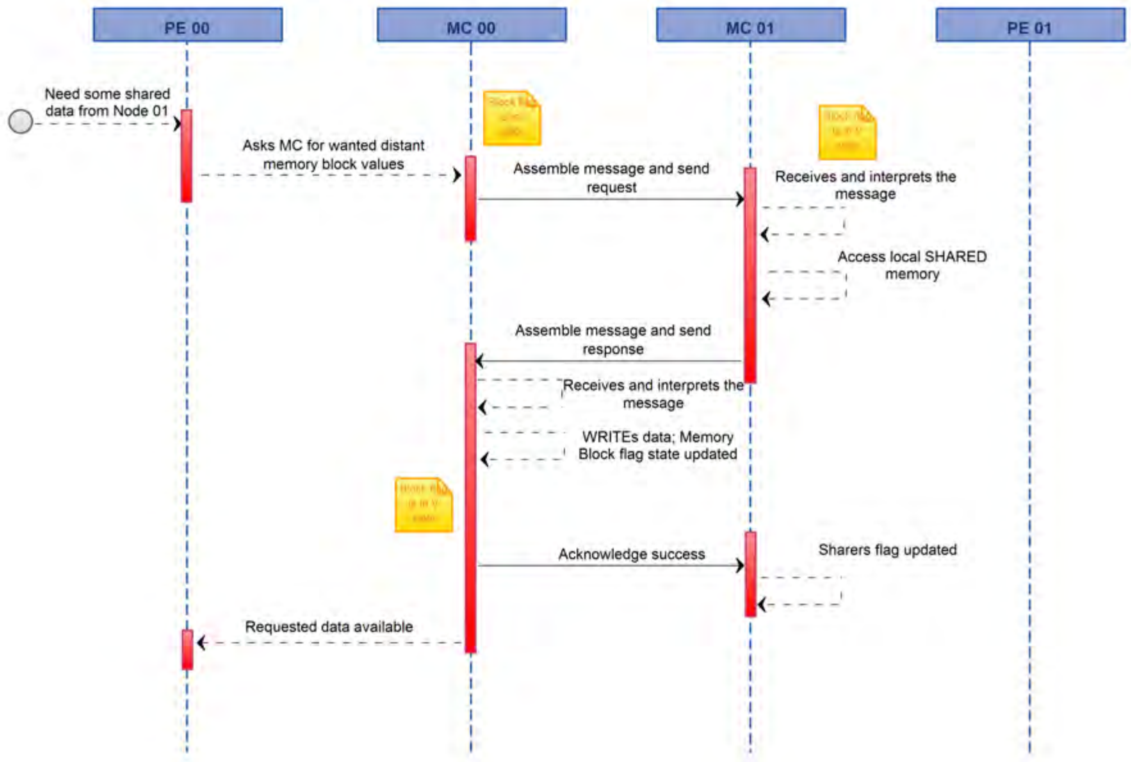

PE 00
MC 00
MC 01
PE 01
Need some shared data from Node 01
Asks MC for wanted distant memory block values
Assemble message and send request
Receives and interprets the message
Access local SHARED memory
Assemble message and send response
Receives and interprets the message
WRITEs data; Memory Block flag state updated
Acknowledge success
Sharers flag updated
Requested data available

Figure 3.11: Transactional diagram of READ operation, two sharers scenario

MC that belongs to *node 00* will assemble and sent request that express a need to get desired memory block value from *PE 01.* Once message is received and memory controller of *node 01*, MC 01, interprets the message, first thing to happen is that MC 01 will internally detect that the content of requested memory block is in Valid state according to protocol, meaning that *PE 01* it is the only place where such a content is available, regardless possible prior local processing. Since there are no other sharers for example or similar actions to take care off in advance, requested memory block (its content) can be sent back to initiating *PE 00*, or to be more correct, to its MC 00. Next, the message is sent back to initiator. Initiator will interpret received message as data response on his READ request generated before. Newly arrived value will be saved into available memory block in shared address space. Operation is successfully performed and data is available for potential processing in *PE 00*. Validity statuses are updated according to diagram.

```
− − − − − > (dashed line arrow) represents local operation
——————————> (full line arrow) represent operation that is
             performed via NoC
```

First thing that is noticeable in this moment might be - there is a certain delay from the moment MC 01 start to send response to *PE 00*, and the moment when MC 01 receives acknowledgement. This is true. Intention here is not to allow any WRITE operation in noticed time interval on same memory location in *node 01*. Second notification represented above, from one side can be interpreted as acknowledgement of *node 00* that operation has been performed successfully and, from other side, that all sharers flags, one it this scenario will get updated.

From Figure illustrating this scenario, one can notice that requester and its memory controller exchanged exactly two messages. Once first is triggered, node (via MC) needs to make sure that there will be memory block reserved for the result of READ operation. This reserved block will be available for memory controller to store received operation result. Similar concept applies for next scenario's.

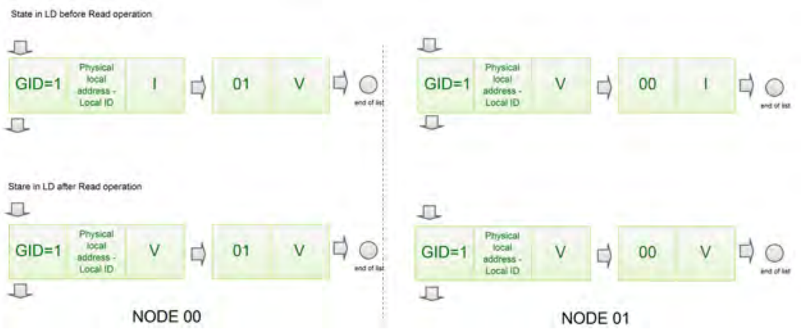

State in LD before Read operation
GID=1
Physical local address - Local ID
I
01
V
end of list
GID=1
Physical local address - Local ID
V
00
I
end of list
Stare in LD after Read operation
GID=1
Physical local address - Local ID
V
01
V
end of list
GID=1
Physical local address - Local ID
V
00
V
end of list
NODE 00
NODE 01

Figure 3.12: LD's pre and post states for Scenario 1, assuming that GlobalID of shared block is 1

3.3 Scenario 2. READ operation, four sharers

Now, let us interpret Figure 3.13 that represents one of the more complex scenario's that may occur. Imagine as **precondition** of this use case that *node 01*, *node 10* and *node 11* are sharing certain memory block value, meaning that all of them have valid copy of memory blocks and awareness of this is noted down in each of the LDs', and of course all nodes are aware that *Node 00* is also sharer, but at this moment his copy is invalid. *Node 00* decides that content of that same memory block is also needed for its own purposes and it wants to get this data from its northern neighbor *node 01*. Once MC that belongs to *node 01*, MC 01, receives and interprets a message, response will be assembled and sent back. (It is perfectly not necessary to activate lock mechanism (more words in write scenario) at any sharer, since it is sharers who receives response message (with read data) responsibility to inform all other sharers of its flag change on the same memory block context in meanwhile).

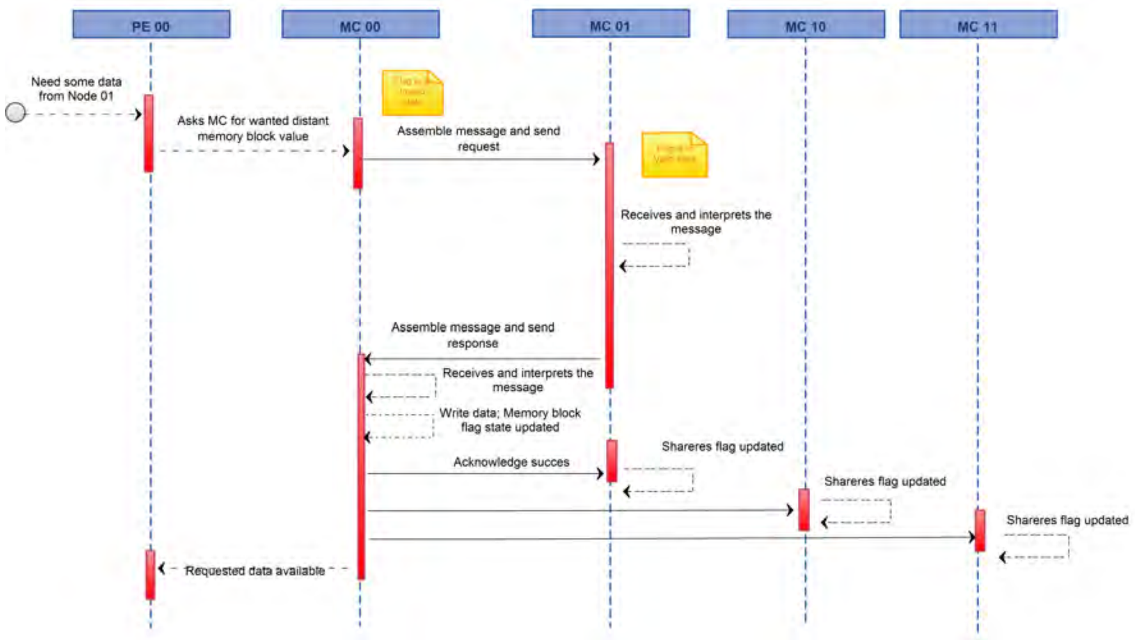

PE 00
MC 00
MC 01
MC 10
MC 11
Need some data from Node 01
Asks MC for wanted distant memory block value
Assemble message and send request
Receives and interprets the message
Assemble message and send response
Receives and interprets the message
Write data; Memory block flag state updated
Acknowledge succes
Shareres flag updated
Shareres flag updated
Shareres flag updated
Requested data available

Figure 3.13: Transactional diagram of READ operation, four sharers

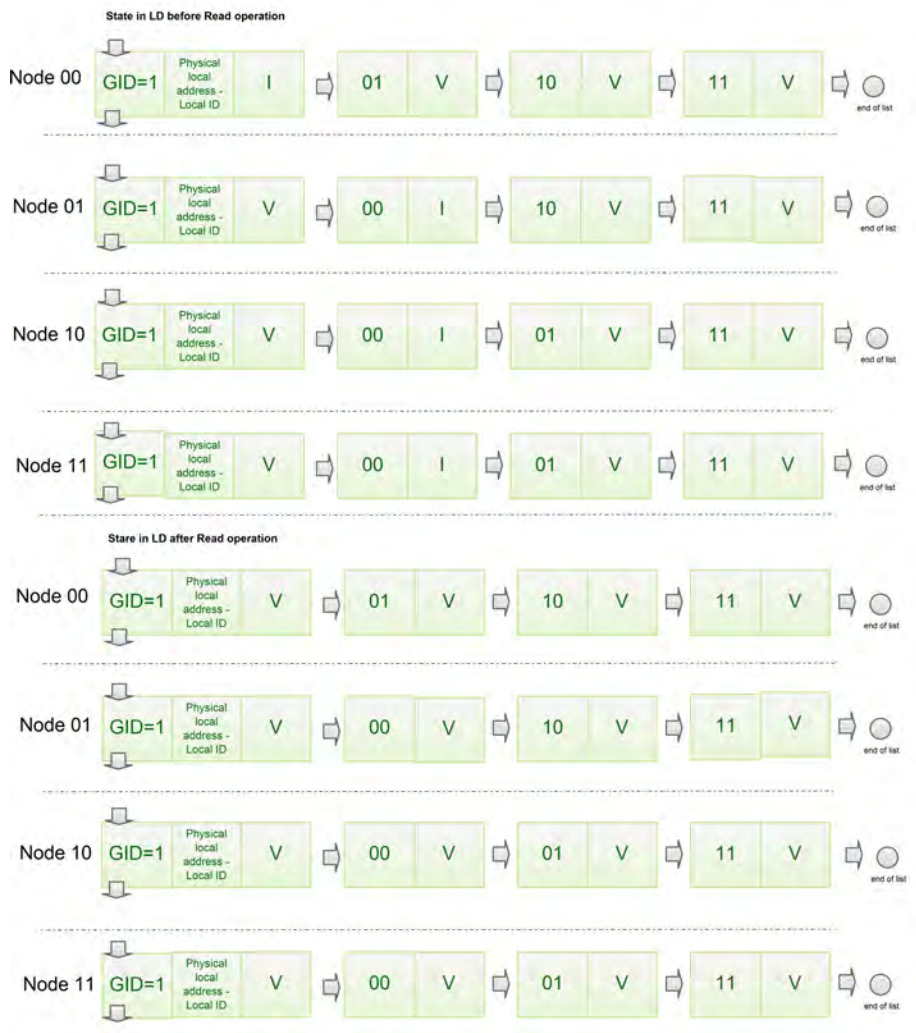

State in LD before Read operation
Node 00
GID=1
Physical local address - Local ID
I
01
V
10
V
11
V
end of list
Node 01
GID=1
Physical local address - Local ID
V
00
I
10
V
11
V
end of list
Node 10
GID=1
Physical local address - Local ID
V
00
I
01
V
11
V
end of list
Node 11
GID=1
Physical local address - Local ID
V
00
I
01
V
11
V
end of list
Stare in LD after Read operation
Node 00
GID=1
Physical local address - Local ID
V
01
V
10
V
11
V
end of list
Node 01
GID=1
Physical local address - Local ID
V
00
V
10
V
11
V
end of list
Node 10
GID=1
Physical local address - Local ID
V
00
V
01
V
11
V
end of list
Node 11
GID=1
Physical local address - Local ID
V
00
V
01
V
11
V
end of list

Figure 3.14: LD's pre and post states for Scenario 2, assuming that GlobalID of shared block is 1

Now, message with response will be delivered to requester node. MC 00 will get desired data and it can now store it locally at available shared memory block location. Next, it is time to inform the sharers about success of transaction. Sharers will update their LD's. However, more than one parallel reading should be allowed by protocol since it will not inject incoherency, but one needs to pay attention on properly updating LDs. Following flow of scenario and diagram - READ operation is now marked as completed and initiator's PE unit has ability to potentially process this value further, of course by the rules of protocol.

In conclusion, READ operation is non-blocking and this is important to emphasize.

3.4 Scenario 3. WRITE operation, four sharers

The preconditions for use case presented on Figure 3.15: all nodes share certain memory block, they are all aware of it and details are mapped in each of LD's and memory block is in Valid state in all sharers. Node 00 processed block value locally in its PE, and now attempts to write to it. For this MC 00 needs to ask for permission from all sharers, meaning all other copies need to be invalidated before WRITE can be triggered. MC 00 will assemble and send invalidation messages to all 3 other sharers and will activate lock on that memory block in the same time. MC 01, MC 10, and MC 11 will act accordingly on received invalidation and invalidate their own copies. MC 00 issued 3 invalidation requests, and to proceed further 3 tokens (acknowledgement) messages need to be received. Once this condition is met, WRITE can be performed, then lock can be deactivated and operation is now marked as successful. After writing, node will inform all sharers above write success. *In a way, this idea of* ***'lock activation/ deactivation'*** *reminds us of idea of mutexes present in C programming language world for example*

In future, if any of the sharers needs to process its own local copy, for example if he wants to WRITE to it, this will not be allowed by our protocol, node will be forced to fetch valid data first.

From Figure illustrating this scenario, it is noticeable that requester and its memory controller exchanged exactly two messages. Once first is triggered, node (via MC) needs to make sure that *any* other operation on this memory block should not be allowed until current operation success is confirmed via second message. Similar rule applies for next scenarios as well. Also, to clarify what is happening in each of LDs' in this scenario, context of each LD is given on Figure 3.16, assuming that GlobalID of shared block is 1.

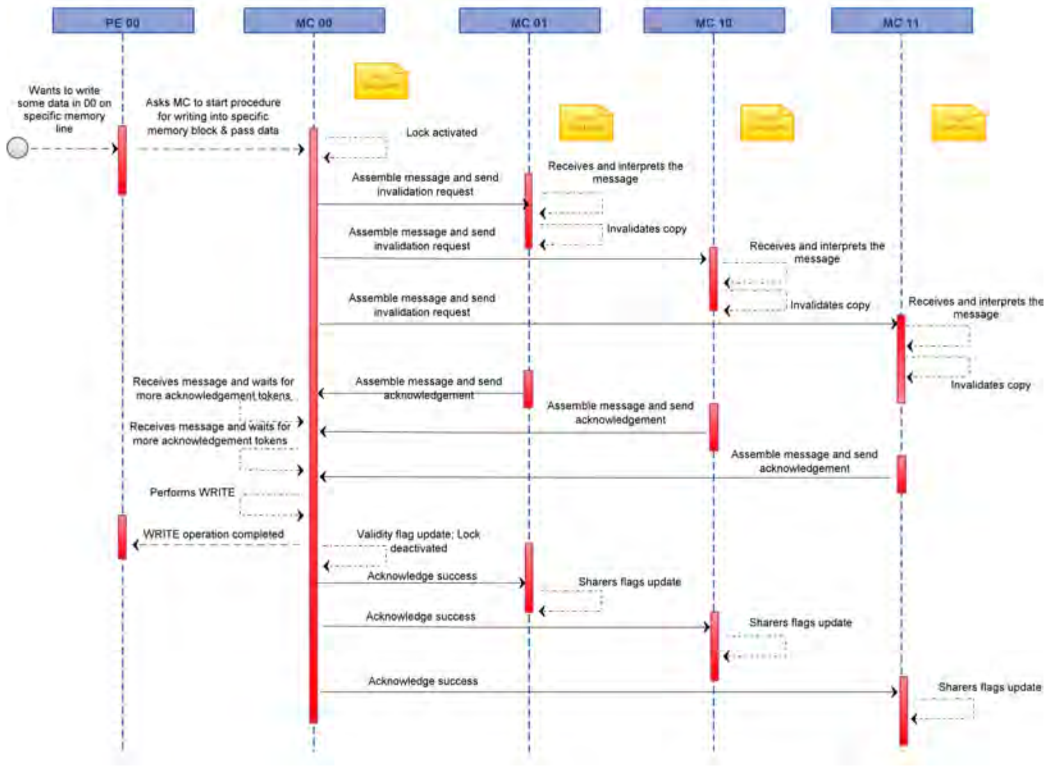

PE 00
MC 00
MC 01
MC 10
MC 11
Wants to write some data in 00 on specific memory line
Asks MC to start procedure for writing into specific memory block & pass data
Lock activated
Assemble message and send invalidation request
Receives and interprets the message
Invalidates copy
Assemble message and send invalidation request
Receives and interprets the message
Invalidates copy
Assemble message and send invalidation request
Receives and interprets the message
Invalidates copy
Receives message and waits for more acknowledgement tokens
Assemble message and send acknowledgement
Assemble message and send acknowledgement
Receives message and waits for more acknowledgement tokens
Assemble message and send acknowledgement
Performs WRITE
WRITE operation completed
Validity flag update; Lock deactivated
Acknowledge success
Sharers flags update
Acknowledge success
Sharers flags update
Acknowledge success
Sharers flags update

Figure 3.15: Transactional diagram of WRITE operation, four sharers

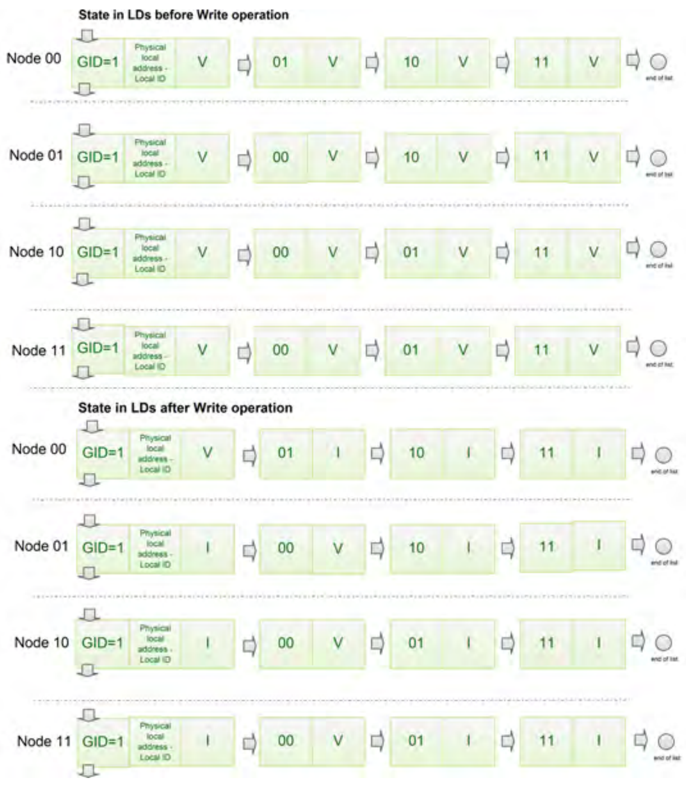


Figure 3.16: LDs' pre and post states for Scenario 3, assuming that GlobalID of shared block is 1

3.5 Scenario 4. WRITE operation, n sharers, Invalid state

Attempt of WRITING into memory block that is in INVALID state will result with failure status. Protocol must not allow such an operation. Simply - if memory is in INVALID state, that implies that same value is shared with one or many other nodes in the network and some of them has valid copy. But however, this scenario of node attempting to write into INVALID block may occur due to a fact that node(its PE), let us call it node I, can take certain value that is originally in Valid state, and PE I can process it during longer time interval for example. In meanwhile, some other sharer (node II and PE II) processed its copy of same value faster and already executed WRITE operation (this was acknowledged by MC of node I but here general assumption is that MC I has no permission to stop or invalidate any operations in PE I). Therefore, node I shall not be allowed to perform WRITE since it appears that he was working with outdated value for a certain time interval lately. Upon failure, node will understand that next logical step would be to update marked memory block value and to try with processing it again[2].

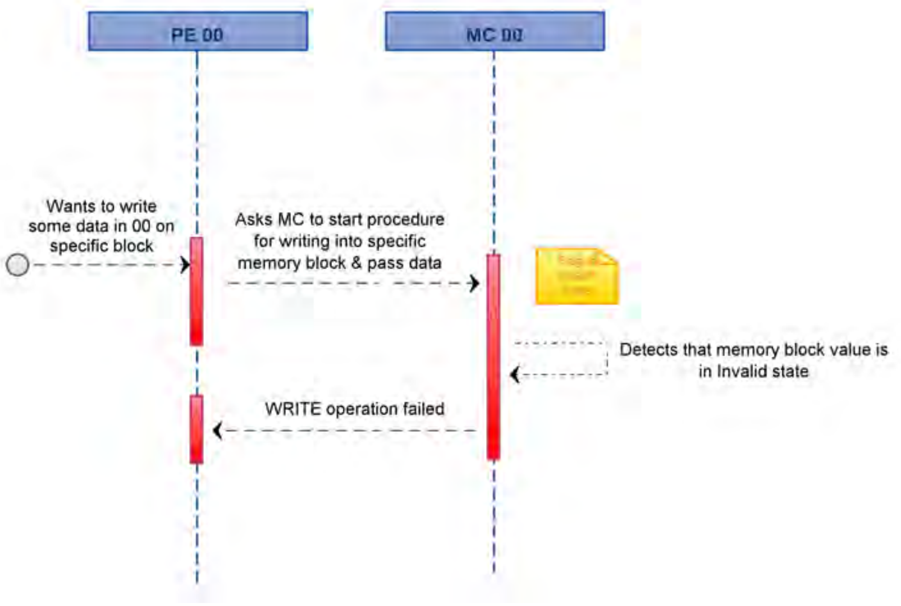

PE 00
MC 00
Wants to write some data in 00 on specific block
Asks MC to start procedure for writing into specific memory block & pass data
Detects that memory block value is in Invalid state
WRITE operation failed

Figure 3.17: Impossible to Write to Invalid memory block location

[2]Probably an improvement (after the thesis) would be here, that WRITE operations first cause a temporary invalidation (TI) in all sharer's memory controllers. During TI state a local write (as considered in this section) is not allowed. With the S-flag update the state is set from TI to I. If the local flag is V, then WRITE is allowed, following the standard procedure (invalidating all potential sharers to TI state before performing the local WRITE).

3.5.1 Introducing node's priority levels

Analysis showed that there exist one possible scenario that might happen and that protocol itself would not be able to solve without additional assumptions. Let us imagine situation where there is 2x2 NoC and each out of 4 nodes contains in its own SHARED memory (on specific memory block) certain memory content that is shared between all 4 nodes. And let's assume that current status of all those shared memory blocks is Valid (according to protocol notation).

- Typical scenario

One node decides to write on that location. Meaning, node has processed that data and now it want to write new value to this location. Protocol will prior to allowing this node to execute WRITE operation send via NoC invalidation messages to all other sharers (3 in this case) and after receiving 3 acknowledgements confirmation, ability to write is granted to requesting node and WRITE operation can occur now. Acknowledge about success is sent to all sharers.

- More complicated scenario

What happen if all four nodes (more than one in general) decide in the same time to perform WRITE operation? Same time here refers to same clock cycle. All nodes will attempt to send (inject into NoC) invalidation messages to other shares. Let's imagine that node 01 will get invalidation from node 00, and then node 11 will get invalidation from 10, and when for example node 01 informs node 11 regarding invalidation of that specific memory block that has recently been invalidated by 00, and this confusion happens within other nodes as well, it has to be stated that chaotic scenario will occur and simple protocol could not be able to solve current issue. This would result and lead to incoherent view of specific memory location and its computer engineer task to prevent it.

- Solution

As a solution, idea about nodes priority levels was introduced, where protocol designer would in advance specify hierarchy of nodes, so coherency protocol would know in above described complicated scenario which invalidation request has greater priority and advantage among several incoming one. Here, one can also think of scenario when lower level invalidation comes before higher level invalidation request, but in this less likely scenario one of the possible solutions can be issuing cancellation of invalidation request given earlier.

3.6 Concept Summary

Key points of this chapter are:

- Modules MC, LD and their interaction and maintenance are heart of the protocol.
- Intelligent application data mapping with predefined sharers knowledge for every memory block is needed.
- No matter how many sharers exist for certain memory block value, while performing READ operation, it is not necessary to activate lock mechanism for any sharer. READ operation is meant to be non-blocking. This is implied by WRITE policy, which states that protocol needs to invalidate all sharers and get write permission for every single one of them. Diagrams given in this chapter demonstrate typical scenarios in 2x2 NoC, but concept is intuitively applicable to any NoC size.

4 Protocol initialization phase and Data Structures

In this chapter an overview of protocol initialization is presented. Before initialization phase, certain data is assumed to be given. This data is observed as an input for protocol initialization phase, and from the other side, data structures and it's content created during this phase are observed as an phase output. Protocol is designed to be fully dynamic and to adopt to given inputs, meaning there are no limitations in terms that protocol allows limited number of shared memory blocks or limited number of sharers, moreover, this characteristics are determined by protocol inputs and one can experiment with the same. However, in general, system memory is limiting factor.

Given inputs are:

- **1.** Information about an amount of shared memory blocks and its sizes. For the purpose of this thesis memory blocks content is observed as an abstract.
- **2.** Information about which nodes are sharing which memory blocks with whom,
- **3.** Time information about the PE operations triggered from each node (read or write),(testbench).

Figure 4.1 represents class diagram of main protocol elements, MC(memory controller), LD(local directory), shared memory modules and its interaction. Memory Controller object, instantiated for each router in NoC, contains several elements but most significant to mention are pointers to: NI buffer, first element of node's shared memory array, first element of VerticalDirectory list also known as Local Directory. With this pointer MC can access and communicate respectively with NI (inject messages into NoC), node's shared memory and entire Local Directory structure.

Let's elaborate each class/list and its role in class diagram:

- **SharedMemory**: represents an dynamic array of shared memory blocks within single node. Array is of a certain fixed size, generally configurable.
- **VerticalDirectory**: represents node's LD. Represented also on Figure 4.2, it is in essence list of pointers, where each pointers points to first element of horizontal list. As many shared memory blocks certain node is assigned to as an sharers, same number of elements of VerticalDirectory will be initialized.

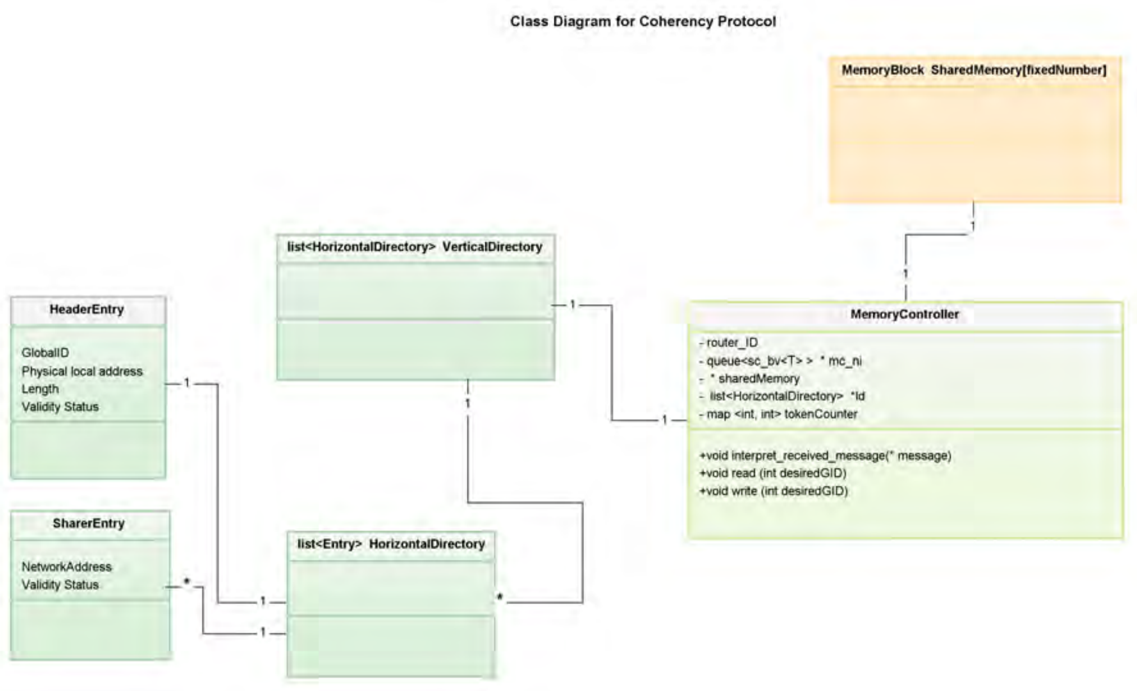

Class Diagram for Coherency Protocol
MemoryBlock SharedMemory[fixedNumber]
list<HorizontalDirectory> VerticalDirectory
HeaderEntry
GlobalID
Physical local address
Length
Validity Status
MemoryController
- router_ID
- queue<sc_bv<T> > * mc_ni
- * sharedMemory
- list<HorizontalDirectory> *ld
- map <int, int> tokenCounter
+void interpret_received_message(* message)
+void read (int desiredGID)
+void write (int desiredGID)
SharerEntry
NetworkAddress
Validity Status
list<Entry> HorizontalDirectory
1
*

Figure 4.1: Class diagram of main protocol elements and interactions

- **HorizontalDirectory**: Represents a structure that contains sharers info and their statuses for single shared memory block. HorizontalDirectory has only and only one HeaderEntry and multiple SharersEntries which depends of sharers total count for specific shared memory block. See also Figure 4.2. Both structures, VerticalDirectory and HorizontalDirectory and their subcomponents, are driven by input data 2.
- **HeaderEntry**: Information about shared memory block and its validity status in local node
- **SharerEntry**: Information about sharers of the same memory block(as the one in HeaderEntry) and its validity statuses.

Methods of MC class, are examined deeper in next chapter.

Finally, input data 3. represents degree or amount of real time operations on top of shared memory in each node of NoC system, and basis of experimental work/experiments is driven by this input data.

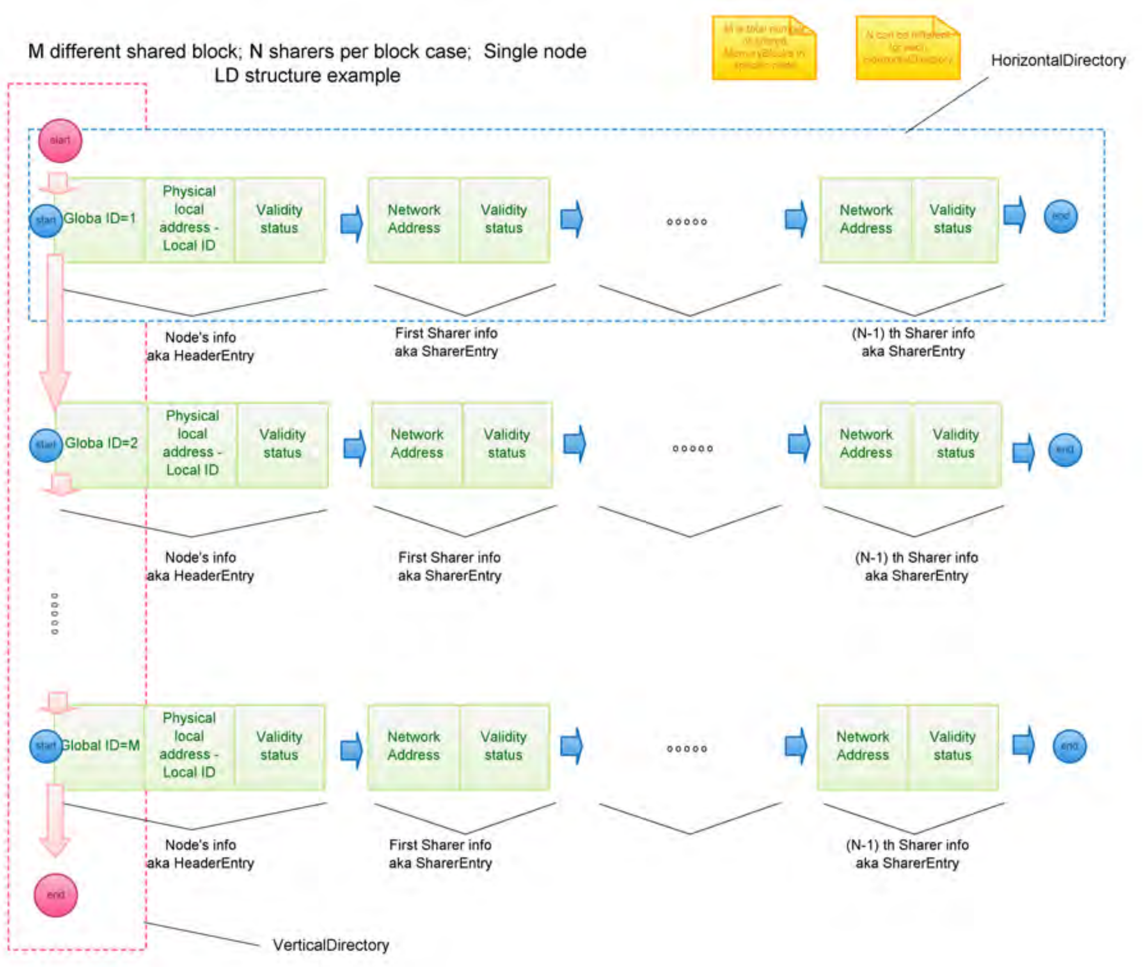

M different shared block; N sharers per block case; Single node LD structure example
M is total number of shared MemoryBlocks in specific node
N can be different for each HorizontalDirectory
HorizontalDirectory
start
start
Globa ID=1
Physical local address - Local ID
Validity status
Network Address
Validity status
ooooo
Network Address
Validity status
end
Node's info aka HeaderEntry
First Sharer info aka SharerEntry
(N-1) th Sharer info aka SharerEntry
start
Globa ID=2
Physical local address - Local ID
Validity status
Network Address
Validity status
ooooo
Network Address
Validity status
end
Node's info aka HeaderEntry
First Sharer info aka SharerEntry
(N-1) th Sharer info aka SharerEntry
ooooo
start
Global ID=M
Physical local address - Local ID
Validity status
Network Address
Validity status
ooooo
Network Address
Validity status
end
Node's info aka HeaderEntry
First Sharer info aka SharerEntry
(N-1) th Sharer info aka SharerEntry
end
VerticalDirectory

Figure 4.2: Local Directory (LD)

5 Experimental Work and Results

5.0.1 Environment

- *Some words about Jonas Bargon's thesis and XHiNoC code*

  As a starting point of this thesis, Bachelor thesis titled 'Mixed Level Simulation for Network-on-Chip' by Jonas Bargon defended at Technical University of Darmstadt, Germany in 2011 was selected. In Bargon's thesis, the goal was to create an interface that will be able to instantiate and make operational XHiNoC routers (TU-Darmstadt internally developed routers) together with the Atlas tool (allows creation of different size NoCs). As a main result the SystemC interface that performs above mention tasks was created, and it provides end user with an ability to use XHiNoC routers originally written in Very-high-speed integrated circuits (VHSIC) hardware description language (VHDL). Simulation of code (both VHDL and C++/SystemC) is done in ModelSim environment and more words about this programming languages and tools will come later. In this master thesis Bargon's interface is used as a starting point, on top of which implementation of directory based protocol is provided. The original code written in VHDL language that implements XHiNoC router was used as it is and available functionality and features of XHiNoC were used as an underlying NoC.

- *C++/SystemC*

  SystemC is a C++ class library. At the moment of creation of Bargon's thesis SystemC was determined and distributed as an Open SystemC Initiative publicly available standard also known as as IEEE Std 1666-2005. During the time interval of this master thesis same class library got updated and now it corresponds to IEEE Std 1666-2011, available via same medium. Even though there were certain changes and deprecated features in the newer version of library, that could not create any potential misunderstanding in this thesis case. In terms of C++, refered is C++ programming language represented in ISO/IEC 14882:2003 standard.

- *ModelSim*

  Represents popular simulation tool by Mentor Graphics. ModelsSim SE 6.6d, Revision 2010.11 was selected since it allows simulation of two different Hardware Description Languages (HDL) languages in the same time, for example in this thesis case VHDL and SystemC.

- *Available code (VHDL + SystemC)*

  Current interface is configurable in such a manner that it can correspond to various sizes of NoC (2x2, 3x3, 4x4, 6x6 and 10x10). However, for the research conducted in this thesis 2x2 NoC has been selected as a simplest scenario and all experimental coding was done against this NoC size. For the future work,testing and improving protocol to an arbitrary NoC size is planned.

Simulating mixed-language designs with ModelSim is the feature that makes this tool popular and very practical simulation software. In this thesis case, simulation initialization and execution includes these general steps:

1. Compiling VHDL source code using *vcom* command. Compiling SystemC C++ source code using *sccom.* Compile all modules in the design following order-of-compile rules.
2. For designs with SystemC code linking of all objects in the design is achieved using *sccom -link.*
3. Elaborating and optimizing design using the *vopt* command.
4. Simulates the design with the *vsim* command.
5. Runs design.

VHDL source code is compiled by vcom, from other side SystemC/C++ source code is compiled with the sccom command. Therefore, there exist two separate compilers. By default, ModelSim version used during this thesis, in general does not come with preinstalled compiler for C++ code, therefore there was a need to install it additionally in the beginning. More detailed information about mixed-language simulation and other specification are available within Document folder of installation file of ModelSim.

5.0.2 Available interface and situation in the beginning of thesis from programmer's perspective

The cycle-accurate VHDL model of XiNoC with SystemC intefrace was provided as a starting point. Author's practical task was to implement coherency protocol into existing simulation environment. Main guidelines and software requirements were to always think into implementation direction which will leads towards the scalable protocol solution. Assuming that one is familiar with XiNoC code, before this protocol was implemented situation was as follows: central logic of an interface is implemented in local_interface class. The local_interface systemC module is responsible for the main tasks of the complete interface: the sending, receiving and logging of the traffic. As a main functions available via this interface two functions have been identified:

```
void ni_send(sc_bv<T> flit);
void ni_receive_test();
```

Function *ni_send* takes a flit and injects it into NoC. Function *ni_receive_test* is constantly watching is there something new at *dout_port* and what it is basically performing is polling strategy. Once something arrives (and that must be flit, since it is the smallest unit of data that can be moved around the NoC) flit will be stored and available to be processed further on the receiver side. It is important to mention that function *ni_send* is functionally same for all routers (no matter of number of routers, router ID), and it is important to know that programmer needs to call this function from each specific router separately. Flit will "know" where to go (since it contains info about destination router), meaning on receiving side, where there is *ni_receive_test* function only flits meant for this specific routers will arrive.

In the *TopNOC.cpp* file let us examine line:

```
#ifdef NoC22
  local_interface r00_l,r01_l,r10_l,r11_l;
#endif
```

This means that there exist 4 local interfaces (identical) instantiated in case of 2x2 NoC, which is the primary experimental focus at this moment of master thesis. So far, processing cores have been "simulated" as void functions within local_interface.h file and within them programmer needed to specify actions for each router.

5.0.3 How protocol works from programmers' perspective?

Implemented protocol represent pure software integration into existing simulation environment. For overview of simulation environment of XHiNoC, relevant for understanding of this protocol, the best reading is most probably thesis by Bargon, mentioned above. The scalability requirement influenced entire software development life-cycle. Here, entire flow of actions relevant for protocol simulation is described. Once simulation is initialized with **do run.do** command in ModelSim console (assuming that project is properly created and both VHDL and SystemC source codes are imported), compilers will perform their tasks, design is loaded and simulation starts. Let us concentrate on SystemC part here, since protocol is implemented in this part. SystemC kernel will among others, identify protocol test-bench thread and simulate it:

```
SC_THREAD(fake_processing_core);
sensitive<<this->clk->posedge_event();
```

With every positive clock edge (on every 2ns) test-bench thread will be triggered. Within local interface (routers) main file, function named *fake_processing_core* is the one actually being watched in this case by kernel. By examining the code given below (which is generic), it becomes clear that as long as there exist more operations to be executed, the simulation will continue until all operations have been completed. Those actions will trigger main protocol component, Memory Controller and it will further cooperate with Shared Memory and Local Directories as elaborated in conceptual chapter. More details are given in sections below describing MC in detail. The approach is very generic and fully scalable in terms of test-bench actions triggering protocol reactions.

```

void fake_processing_core(){


  this->totalNumberOfActions = this->triggeringActions.size();
  // Given as system input. Testbench info
  this->numberOfActionsToGo = this->totalNumberOfActions;
  list<triggers>::iterator it;
  // triggers is just a list with details of when to trigger ↩
     which action by specific node
  // list is unique for all nodes, that's why keyword this is ↩
     used.



  for(it=this->triggeringActions.begin(); it!=this->↩
     triggeringActions.end(); ++it){
    //nexttriggeringTime=(*it).t_time;

    while(this->numberOfActionsToGo==this->totalNumberOfActions)↩
       {
      if(sc_simulation_time()>=(*it).t_time){
        // >= is because we have 2ns cycle accuracy in current ↩
           setup
        if((*it).t_action=='R'){
          this->memoryController.read((*it).t_gid);
          // READ operation has just been triggered from this ↩
             node by calling its MC
          total_number_of_read_requests++;
        }else if((*it).t_action=='W'){
          this->memoryController.write((*it).t_gid);
          // READ operation has just been triggered from this ↩
             node by calling its MC
          total_number_of_write_requests++;
        }else{
          cout<<"Error in input file with R or W. Should be in ↩
             capital letters!"<<endl;
        }
        this->numberOfActionsToGo--;
      }else{
        wait();
        // waiting for times to match so next action can be ↩
           triggered;
      }
    }
    this->totalNumberOfActions--;
  }

}
```

MC will further trigger additional protocol actions, as described in conceptual chapter, and those actions will also be recognized by kernel and simulated. Those actions represent actual READ (memoryController->read()) and WRITE (memoryController.write()) operations. On the end of simulation, statistics are displayed in ModelSim console, similar to ones available in experimental results table. Statistics include total simulation execution time, as well as information of operations success details.

5.1 Notion of Shared memory

Shared memory is the one available to every router at (almost) any moment. Physically, this memory is distributed homogeneously among all routers in the NoC. Shared memory consists of certain number of memory blocks per node, which are multiples of chunks of 32 bit. Meaning is there is such a case that certain memory block is 31 bit, then protocol will allocate one chunk of 32 bit, and that would represent this memory block. If there is a need to have block of 55 bits, protocol will allocate 2 chunks, all together 64 bits for this memory block. Just like as previous collaborators, protocol operates on abstract data, since the idea was to prove concept. Working with real data that would be given to protocol as an system input is easily implementable, but within this thesis this task was considered out of scope. In conclusion, protocol is just moving around the NoC abstract flits of the same sizes as it would have been doing in case of real application data.

5.2 Directory Structure

Each router/node has dynamic local directory structure based on assumed system inputs elaborated before. Manipulating with directory structure is crucial in terms of optimization. Here one can make a differentiation in terms of simple protocol and future work, modified protocol in following manner. If there is a need for node to fetch its outdated value from valid sharer, memory controller unit will look under specific directory, specific shared memory line entry from where this is possible. Simple protocol will just iterate thought list of shares and will fetch data (assemble READ request) from **first** sharer in the list. Modified protocol can be smarter here, and iterate via list till its end, and calculate on the way who of available sharers has minimum distance from requester, meaning who can deliver desired data fastest. Also, immediate multi-cast implementation would calculate who else might be needing same info soon, so protocol could delivered data in advance. Note that modified protocol would show off it benefits in larger size NoCs, so in terms of 2x2 NoC, results are expected to be on the same level, so in this thesis simple protocol is implemented.

An example of Local Directory structure for one single node (data token from random experiment setup) is presented below. Refer to Figure 4.2 from protocol concept chapter for clearer understanding of below given structure generated by protocol for each router. In this specific case, experiment started with an assumption that all memory blocks are valid (val_1) in entire NoC system.

```
Start of VD
|
1(val_1)--->>0(val_1)--->>1(val_1)--->>2(val_1)-->> End of HD
|
8(val_1)--->>0(val_1)--->>1(val_1)-->> End of HD
|
4(val_1)--->>0(val_1)--->>1(val_1)-->> End of HD
|
3(val_1)--->>0(val_1)--->>1(val_1)--->>2(val_1)-->> End of HD
|
7(val_1)--->>0(val_1)--->>1(val_1)--->>2(val_1)-->> End of HD
```

```
|
2(val_1)--->>0(val_1)--->>1(val_1)-->> End of HD
|
9(val_1)--->>0(val_1)--->>1(val_1)--->>2(val_1)-->> End of HD
|
5(val_1)--->>0(val_1)--->>2(val_1)-->> End of HD
|
100(val_1)--->>1(val_1)-->> End of HD
|
25(val_1)--->>2(val_1)-->> End of HD
|
35(val_1)--->>2(val_1)-->> End of HD
|
40(val_1)--->>1(val_1)-->> End of HD
|
45(val_1)--->>2(val_1)-->> End of HD
|
 End of VD
```

In conclusion, flexibility of the local directory structures complies with scalability of current software solution, which makes protocol applicable to any NoC size.

5.3 Memory Controller

Memory controller, unit which tightly cooperates with LD, operates on the informations available in LD and takes care of updating the same. As such MC has same the logic for every node, irrelevant of how many nodes there are in the NoC.

Memory controller is in charge of all actions, requests, issuing invalidation or update messages, etc; which will affect directory content and values of abstract shared memory. Memory controller unit is the central part of the protocol. It communicates with network interface, both with ni_send and ni_receive functions. Let me elaborate it in more details. Improving current sending and receiving functions in a way that communication is available on the message granularity level is part of programming task covered in this thesis. This was done in C++/SystemC and simulated in ModelSim. Memory controller is the unit assembling messages, passing it to Network Interface (NI), that will inject message into an existing traffic. On the other side (receiving) logic is a bit different. NI will assemble messages out of incoming flits and as a such atomic units, deliver messages (to be more precise, implementation uses only pointers) to Memory Controller object for interpreting it and acting correspondingly.

In theory this approach is connected to work of T.Bjerregaard and S. Mahadevan,

[BM06] where flits/phits are identified as mediums that correspond to Link level, packets that are built out of flits correspond to Network/Session/Transport level, and as a highest level in Open Systems Interconnection (OSI) layer there is Application level, where granularity of medium is represented via messages built out of packets. In this thesis, main lower level goal is to implement protocol and demonstrate its correctness, and as such one of assumption was to consider packet always built out of 3(three) flits (header, body and tail), unless data is transported by protocol, when there are 4+ flits(header, body, body+ and tail). Also for simplicity reasons, assumption was made that 1(one) packet represents single message. It is important to always keeps this fact in mind, but as of future research work, directions are that this protocol has to support random size of flits constructing packet (header, n x body, tail), as well as arbitrary number of packets constructing message. This assumption cannot be restrictive in terms of proving correct behavior of protocol.

It is a priori known and guaranteed, (and here author of the thesis has to give HUGE credit to previous developers and teams that created XiHNoc platform) that flits of the same message issued by single router are being delivered in order to destination router. However, due to complex traffic scenarios, incoming traffic into a single router may arrive from multiple directions, in general 2 on power of n, where n is homogeneous NoC x dimension, for any size of NoC under examination. Already available feature, implemented by previous teams, assigns flit ID that distinguishes same type of incoming flits. Meaning for example if two header flits arrive one after another into single router, one can differentiate them by flit ID which clearly states that header flits come from different direction/different physical routers. What does it means for protocol? Modified ni_receive function was implemented, in a way that there exist buffers/containers that will step by step be loaded with flits in correct incoming order and where messages will be assembled. Complete messages(once tail flit has arrived) are passed to Memory Controller.

Main memory controller functions are:

- **read** : Checks if shared memory block is valid locally (read hit), or in case it is invalidated (read miss), send read request to first valid sharer from horizontal list directory structure.
- **write**: Triggers sending of token requests (acknowledgement requests) from all sharers in case write is allowed (memory block in valid state - see Figure 3.9). Otherwise, cancels write requests and informs PE.
- **interpret_received_message**: The most complex MC method. Decodes what's been encoded in first data-body message flit data region - see Figure 5.2, and acts accordingly. Regarding Figure 5.2, 32 available bits are used to encode info about protocol message type (P_TYPE), memory block involved in operation (P_GID) and P_EXTENSION is left for future protocol improvements.

Sizes are configurable in main XiHNoC config file.

Below, simplified pseudo code of the complex MC method is given. Based on flit count within message and based on decoded protocol types MC will perform appropriate actions.

```
void interpret_received_message( message ){

  if(message.size()<3){

    // Within protocol there are no messages with
    // less that 3 flits. Signal error

  }else if(message.size()==3){

    //temprary extract flits
    //gets 32 protocol data body flits where protocol is ↩
       encoded into variable tempProtocolData
    //see what is encoded into protocol type and act ↩
       acordingly
    // range(protocol_type_hi, protocol_type_low) defines what↩
        type of protocol message is encoded

    if(protocol_type_token_request arrived){

      //CASE WHERE TOKEN REQUEST ARRIVED
      //invalidate its own flag in LD
      //send response token back to message sender

    }else if(protocol_type_token_response arrived){

      //CASE WHERE TOKEN RESPONSE ARRIVED

        if(( all tokens arrived){
            //perform write
            //lock release on specfic GID
            //ittereate over HD and send acknowledgements to ↩
               all sharers

        }else{
            //wait for more tokens to arrive
            // decrease expected token counter
        }
    }else if(protocol_type_write_success arrived){

      //CASE WHERE SOME NODE IS INFORMING OF ITS WRITE SUCCESS↩
          on CERTAIN GID
      //invalidate all sharers flags for same GID in LD
      //make sender flag valid for specific GID in LD

    }else if(protocol_type_read_request arrived){

      //CASE WHERE SOME NODE NEEDS VALID DATA
      // starts to send valid memory block of GID to requestor
      // header + protocol databody + as many databody flits ↩
         needed for real data + tail

    }else if(protocol_type_update_am_also_valid arrived){
      //update senders flag in HD for specific GID


    }else{
      //some other protocol type, for future extension...
    }

  }else{
    // Message with more than 3 flits arrived
    // Real application data arrived
    // Store data
    // Assemble update messages; itterate over HD and inform ↩
       all sharers
  }

}
```

In the code itself, more helper's functions are noticeable and their purpose is to supporting above mentioned primary functions or logging desired values to standard output. In general, main software and hardware architectural view of protocol and basic interaction is given of Figure 5.1.

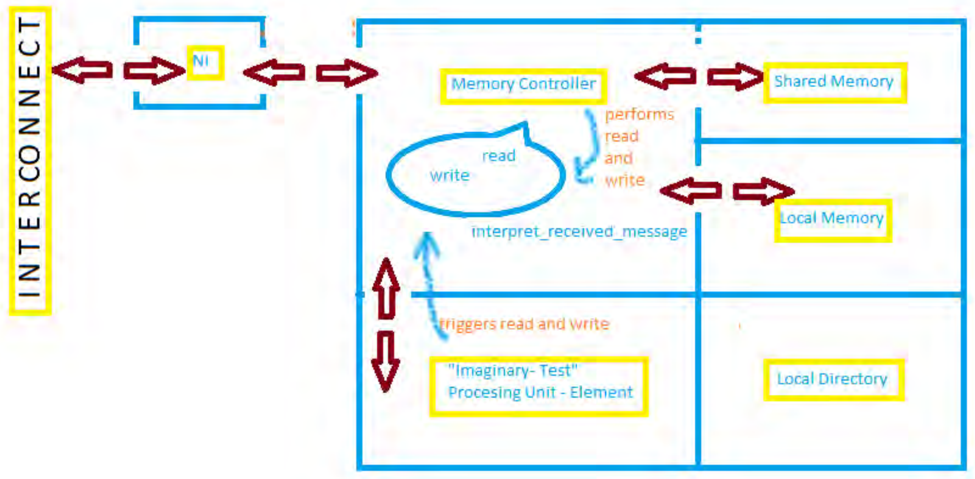


Figure 5.1: Architectural view

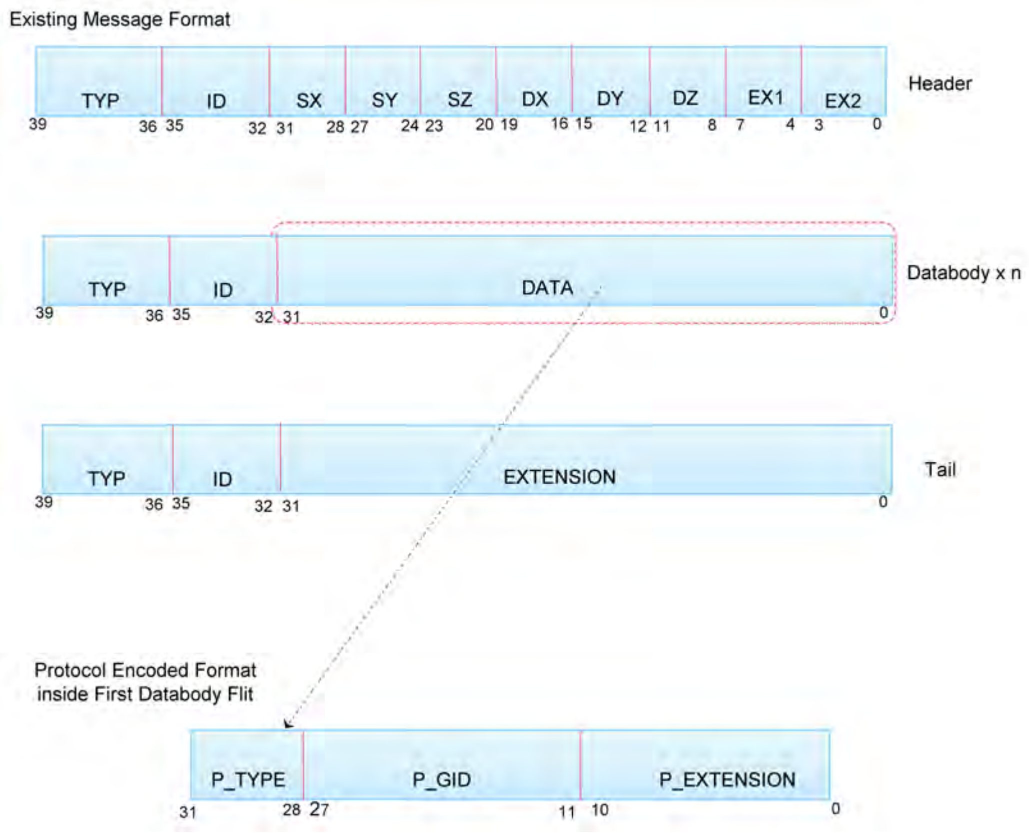


Figure 5.2: Flit formats

5.4 Results

As an underlying experimental platform, 2x2 XiNoC with 4 nodes (each node connected to single router and contains 1 PE, 1 MCs, 1 LD and certain amount of shared memory) was used, similar as ilustrated on Figure 3.8. The experiments were conducted based on different equivalence classes of system inputs. As an input, random number of total shared memory blocks in system memory (5 or 10 or 50 or 100), random overall sharers distribution (2 or 3 or 4) per shared memory block and random number of actions(read or write operations) to be performed on same shared Memory Block(sMB) with ID: 102 (in case of two sharers) or 103 (in case of three sharers) or 104 (in case of four sharers) were selected in each experiment. Time instances (with 1 ns accuracy in our setup) in which operations were triggered and above mentioned scenarios are shown in tables Tables 5.1, 5.2 and 5.3. Tables for 10, 50 and 100 or even more shared memory blocks look same as in case of 5, since no perations were performed on other sMBs but only one stated in tables. Simulation is run for 18668 seconds, just to be on safe side, even this number could go lower. All shared memory blocks in all sharers local memories were valid at the beginning of simulation [1].

2S	**3S**	**4S**	**OP Type**	**at (ns)**	**2S sMB ID**	**3S sMB ID**	**4S sMB ID**
I	I	I	W	89	102	103	104
II	II	II	R	300	102	103	104
I	III	III	R	305	102	103	104
II	II	II	W	315	102	103	104
II	II	IV	R	325	102	103	104
I	I	I	R	344	102	103	104
I	III	IV	W	556	102	103	104
II	II	II	R	590	102	103	104
I	I	I	W	600	102	103	104
II	II	II	W	740	102	103	104

Table 5.1: Experimental setup in case of 5 sMB; 5 READ and 5 WRITE operations, with operations start times in case of 2 Sharers (2S) or Three Sharers (3S) or Four Sharers (4S)

[1] Table 5.1 should be observed as actually 3 seperate tables, case with 2S (disregard all collumns with 3S and 4S), case with 3S (disregard all collumns with 2S and 4S) and case with 4S (disregard all collumns with 2S and 3S). Same applies for Tables 5.2 and 5.3

2S	**3S**	**4S**	**OP Type**	**at (ns)**	**2S sMB ID**	**3S sMB ID**	**4S sMB ID**
I	I	I	W	89	102	103	104
II	II	II	R	300	102	103	104
I	III	III	R	305	102	103	104
II	II	IV	W	315	102	103	104
II	III	III	R	325	102	103	104
I	I	II	R	344	102	103	104
I	I	I	W	556	102	103	104
II	II	III	R	590	102	103	104
I	III	III	W	600	102	103	104
II	II	II	W	740	102	103	104
I	I	IV	W	889	102	103	104
II	II	II	R	900	102	103	104
I	I	I	R	950	102	103	104
II	III	III	W	1100	102	103	104
II	II	II	R	1120	102	103	104
I	I	I	R	1130	102	103	104
I	I	IV	W	1201	102	103	104
II	II	III	R	1204	102	103	104
II	II	I	W	1299	102	103	104
I	III	III	W	1390	102	103	104

Table 5.2: Experimental setup in case of 5 sMB; 10 READ and 10 WRITE operations, with operations start times in case of 2 Sharers (2S) or Three Sharers (3S) or Four Sharers (4S)

2S	**3S**	**4S**	**OP Type**	**at (ns)**	**2S sMB ID**	**3S sMB ID**	**4S sMB ID**
I	I	I	W	89	102	103	104
II	II	II	R	300	102	103	104
I	III	III	R	305	102	103	104
II	II	IV	W	315	102	103	104
II	III	III	R	325	102	103	104
I	I	I	R	344	102	103	104
I	III	IV	W	556	102	103	104
II	II	II	R	590	102	103	104
I	I	I	W	600	102	103	104
II	III	III	W	740	102	103	104
I	I	I	W	889	102	103	104
II	II	IV	R	900	102	103	104
I	III	III	R	950	102	103	104
II	II	II	W	1100	102	103	104
II	II	II	R	1120	102	103	104
I	III	IV	R	1130	102	103	104
I	I	I	W	1201	102	103	104
II	II	II	R	1204	102	103	104
II	III	III	W	1299	102	103	104
I	I	I	W	1390	102	103	104
I	III	IV	W	1489	102	103	104
II	II	II	R	1500	102	103	104
I	III	III	R	1503	102	103	104
II	II	IV	W	1540	102	103	104
II	II	II	R	1550	102	103	104
I	III	III	R	1560	102	103	104
I	I	I	W	1600	102	103	104
II	II	IV	R	1610	102	103	104
I	III	III	W	1670	102	103	104
II	II	II	W	1740	102	103	104
I	I	I	W	1889	102	103	104
II	III	IV	R	1900	102	103	104
I	I	I	R	1950	102	103	104
II	II	II	W	2100	102	103	104
I	III	III	R	2120	102	103	104
I	I	I	R	2130	102	103	104
I	I	IV	W	2201	102	103	104
II	III	III	R	2204	102	103	104
II	II	II	W	2299	102	103	104
I	I	I	W	2390	102	103	104

I	III	IV	W	2889	102	103	104
II	II	II	R	2900	102	103	104
I	III	III	R	2950	102	103	104
II	II	II	W	3100	102	103	104
II	II	IV	R	3120	102	103	104
I	III	III	R	3130	102	103	104
I	I	I	W	3201	102	103	104
II	II	II	R	3204	102	103	104
II	III	IV	W	3299	102	103	104
I	I	I	W	3390	102	103	104
I	I	I	W	4089	102	103	104
II	III	II	R	4300	102	103	104
I	I	IV	R	4305	102	103	104
II	II	II	W	4315	102	103	104
II	III	III	R	4325	102	103	104
I	I	I	R	4344	102	103	104
I	I	IV	W	4556	102	103	104
II	III	III	R	4590	102	103	104
I	I	I	W	4600	102	103	104
II	II	II	W	4740	102	103	104
I	III	III	W	4889	102	103	104
II	II	IV	R	4900	102	103	104
I	I	I	R	4950	102	103	104
II	III	III	W	5100	102	103	104
II	II	IV	R	5120	102	103	104
I	I	I	R	5130	102	103	104
I	II	II	W	5201	102	103	104
II	II	II	R	5204	102	103	104
II	II	II	W	5299	102	103	104
I	III	IV	W	5390	102	103	104
I	I	I	W	5489	102	103	104
II	II	II	R	5500	102	103	104
I	III	III	R	5503	102	103	104
II	II	II	W	5540	102	103	104
II	II	IV	R	5550	102	103	104
I	III	III	R	5560	102	103	104
I	I	I	W	5600	102	103	104
II	II	II	R	5610	102	103	104
I	III	III	W	5607	102	103	104
II	II	IV	W	5740	102	103	104

I	I	I	W	5889	102	103	104
II	III	III	R	5900	102	103	104
I	I	I	R	344	5950	103	104
II	III	II	W	6100	102	103	104
II	II	IV	R	6120	102	103	104
I	I	I	R	6130	102	103	104
I	III	III	W	6201	102	103	104
II	II	IV	R	6204	102	103	104
II	II	II	W	6299	102	103	104
I	III	III	W	6390	102	103	104
I	I	I	W	6889	102	103	104
II	II	IV	R	6900	102	103	104
I	III	III	R	6950	102	103	104
II	II	II	W	7100	102	103	104
II	II	II	R	1120	102	103	104
I	III	IV	R	7130	102	103	104
I	I	I	W	7201	102	103	104
II	II	II	R	7204	102	103	104
II	III	IV	W	7299	102	103	104
I	I	I	W	7390	102	103	104

Table 5.3: Experimental setup in case of 5 sMB; 50 READ and 50 WRITE operations, with operations start times in case of 2 Sharers (2S) or Three Sharers (3S) or Four Sharers (4S)

Observations after experiments:

- *General*: Protocol takes more time to complete all actions as number of operations increases. As number of sharers grows for specific shared memory block, it is more expensive to complete all coherency actions. Same experiments were conducted in NoC with 20, 50, and 100 shared memory blocks and it is clear that overall number of shared memory blocks in NoC does not have an impact on protocol efficiency. Why there is significant number of canceled operations? Explanation consists of two facts. First, if same node, via its MC, attempts to WRITE on shared memory location in specific time interval, when exact same operation (same node, same operation type and same shared memory location) was already triggered before and still not have been completed, protocol will cancel such an attempt. According to Table 5.4 it is clear that in experimental setup there exist certain number of described cancelation. Second fact refers to READ request cancelations on such shared memory locations that have ongoing WRITE operation already started and still not completed, either localy or in

Number of Sharers	Total number of READs	Total number of WRITEs	Number of completed READs	Number of canceled READs	Number of completed WRITEs	Number of canceled WRITEs	Last message arrived @ ns
2	5	5	5	0	3	2	823
	10	10	9	1	5	5	1391
	50	50	31	19	22	28	7391
3	5	5	5	0	2	3	871
	10	10	10	0	4	6	1391
	50	50	42	8	19	31	7391
4	5	5	5	0	2	3	883
	10	10	10	0	3	7	1455
	50	50	39	11	18	32	7391

Table 5.4: Experimental results for coherency protocol based on setup; 5 Memory Blocks in NoC (100, 101, 102, 103 and 104 where no operations have been performed on sMB 100 and 101)

some distant nodes. Let us elaborate those facts on example from Table 5.1, case of two sharers (2S). For easier understanding of same table, let us diregard all collumns with cases of 3S or 4S. Therefore, 10 opetarions in total, 5 of type READ and 5 of type WRITE in predefined time instances, were triggered by nodes on memory block block that has two sharers, marked as I and II. In experiments results table 5.4, first subrow after the table headings text, it is stated that there were 5 successful READs, 0 unsuccessful, 3 completed WRITEs and 2 canceled WRITEs. Protocol canceled WRITE on sMB with ID 102 by node II at 315 ns, because of reason that his copy was invalid, and invalidation came because node I performed WRITE operation on same SMB starting at 89 ns. Second cancelation is for case of node I attempting to WRITE on sMB 102 at 600ns. This is due to fact that same node started exactly the same operation on same SMB just 44ns ago, and this request was still not completed, therefore protocol cancelled it. In this particular scenario, there were not any READs cancelations, but in general they exist in other scenarios, due to above mention reasons.

- *Advantages*: Concept of dynamic local directory structures is fully scalable and can adapt to any size of sharers and any size of shared memory blocks in NoC. Communication between NI and MC and vice versa is as expected and as described in conceptual protocol idea.
- *Disadvantages*: If two or more sharers start to write to same memory block

location in approximately same time, this will cause inability of protocol to complete above specified write actions. Each sharer will ask for acknowledgement tokens from others, and will not give away its own to rest of sharers. This will lead to deadlock situation and current implementation of protocol is unable to handle such a scenarios. However, solution to this problem has been analyzed and presented within earlier section where node's priority notation was introduced. Also, many cancelations due to too soon start of exactly the same operation by same nodes on same sMB cannot be classified as protocol disadvantages, but more likely as too time dense experimental sutup selection.

- *Conclusion*: All assumptions mentioned during the thesis, created general playground where this protocol behaves as correct. Having in mind complexity of SoC system with NoC with communication backbone, size of this playground is rather small at the moment, and going into direction of enlarging its size is now possible, since the foundations have been established.

6 Future research directions

The three key fields where optimization or new ideas are needed are identified, having scalability constraint always in mind :

- **coherence protocol**,
- **dynamic directory based coherence schema's**,
- **compatibility with underlying NoC structure and its behavior**.

Innovative Ideas:

a) Idea (immediate multi-cast) of calculating shares probabilities (in the same time frame [same task graph cycle]) of need to fetch the same data from distant node can save us some communication cost.

b) Flexible, dynamic (in terms of size, not homogeneous) hierarchical clusters, to reflect non uniform work distribution over the NoC is one potential idea for possible contribution. This would reflect implementation of local directory structures present in our protocol, and would have great positive effect in large scale NoC networks. Here, one needs to think in terms and based on what input designer of hierarchical directory based protocol needs to propose solution for optimal directory structure? In essence, one should be aware of several factors. As one of important influencer turns out to be the way of how applications are being mapped on MPSoC. Typically, since heterogeneous platform is assumed, distribution of work load on NoC nodes is not equal (symmetric) and this can be as well, one of determining factors of optimal directory structure and size. In my opinion, it is evident that size of optimal hierarchical solution will not be fixed, but rather should be reconfigurable (resize-able) in such a way that can adopt to different (non-balanced work load) mapped applications.

Also, future work may include:

- Testing and optimization of coherency protocol on bigger size NoCs (up until 10x10)
- 3D integration brainstorming

Bibliography

[BDK+05] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner. Platform 2015: Intel processor and platform evolution for the next decade. *Technology*, 2005.

[BM06] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1):1, 2006.

[CAS+10] B. Candaele, S. Aguirre, M. Sarlotte, I. Anagnostopoulos, S. Xydis, A. Bartzas, D. Bekiaris, D. Soudris, Zhonghai Lu, Xiaowen Chen, J. Chabloz, A. Hemani, A. Jantsch, G. Vanmeerbeeck, J. Kreku, K. Tiensyrja, F. Ieromnimon, D. Kritharidis, A. Wiefrink, B. Vanthournout, and P. Martin. Mapping optimisation for scalable multi-core architecture: The mosart approach. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pages 518 –523, july 2010.

[Dre07] U. Drepper. What every programmer should know about memory. 2007.

[Gha95] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[hip] European network of excellence on high performance and embedded architecture and compilation. http://www.hipeac.net. Accessed: Dec 27th, 2011.

[LB10] I. Loi and L. Benini. An efficient distributed memory interface for many-core platform with 3d stacked dram. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 99 –104, march 2010.

[LC09] S.P. Levitan and D.M. Chiarulli. Massively parallel processing: Its deja vu all over again. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 534 –538, july 2009.

[MJ08] C.H. Meenderinck and B.H.H. Juurlink. (when) will cmps hit the power wall? In *Proceedings of the Euro-Par 2008 Workshops (HPPC)*, August 2008.

[mos] Mapping otimization for scalable multi-core architecture. www.mosart-project.org. Accessed: Dec 27th, 2011.

[Ora] Oracle. The java tutorials. http://docs.oracle.com/javase/tutorial/. Accessed: Dec 27th, 2011.

[PGG06] F. Petrot, A. Greiner, and P. Gomez. On cache coherency and memory consistency issues in noc based shared memory multiprocessor soc architectures. In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pages 53 –60, 2006.

[Sam10] F.A. Samman. *Microarchitecture and Implementation of Networks on Chip with a Flexible Concept for Communication Media Sharing*. PhD thesis, TU Darmstadt, 2010.

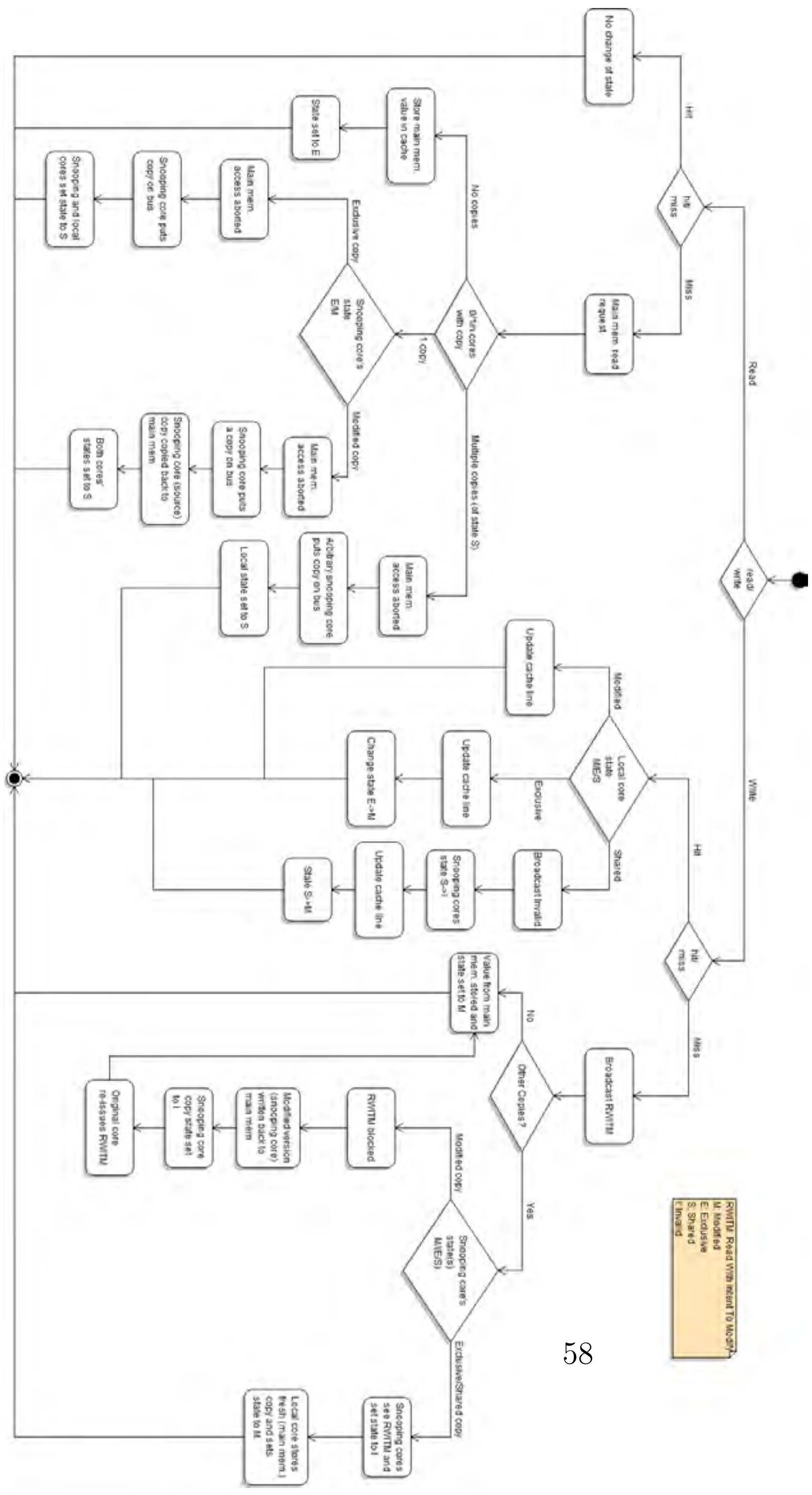
[Sem07] International technology roadmap for semiconductors, 2007.

[SJ11] Dimitrios Soudris and Axel Jantsch, editors. *Scalable Multi-core Architectures: Design Methodologies and Tools*. Springer, 2012 edition, 2011.

[Sto93] Harold Stone. *High-Performance Computer Architecture*. Addison-Wesley, MA, 1993.

[WLWT09] A.Y. Weldezion, Zhonghai Lu, R. Weerasekera, and H. Tenhunen. 3-d memory organization and performance analysis for multi-processor network-on-chip architecture. In *3D System Integration, 2009. 3DIC 2009. IEEE International Conference on*, pages 1 –7, sept. 2009.

A Activity diagram of MESI protocol in bus systems


read/write
Read
Write
hit/miss
Hit
Miss
No change of state
Main mem. read request
0/1/n cores with copy
No copies
Store main mem. value in cache
State set to E
1 copy
Snooping core's state E/M
Exclusive copy
Main mem. access aborted
Snooping core puts copy on bus
Snooping and local cores set state to S
Modified copy
Main mem. access aborted
Snooping core puts a copy on bus
Snooping core (source) copy copied back to main mem
Both cores' states set to S
Multiple copies (of state S)
Main mem. access aborted
Arbitrary snooping core puts copy on bus
Local state set to S
Local core state M/E/S
Modified
Update cache line
Exclusive
Update cache line
Change state E->M
Shared
Broadcast Invalid
Snooping cores state S->I
Update cache line
State S->M
Broadcast RWITM
Other Copies?
No
Value from main mem. stored and state set to M
Yes
Snooping cores state(s) M/E/S
Modified copy
RWITM blocked
Modified version (snooping core) writes back to main mem
Snooping core copy state set to I
Original core re-issues RWITM
Exclusive/Shared copy
Snooping cores see RWITM and set state to I
Local core stores fresh (main mem.) copy and sets state to M
RWITM: Read With Intent To Modify
M: Modified
E: Exclusive
S: Shared
I: Invalid

1

[1]Source: Wikipedia. Accessed: Dec 27th, 2011