FINAL DEGREE PROJECT

COMMUNICATION BOTTLENECK ANALYSIS ON BIG DATA APPLICATIONS

Career: Telecommunications engineering

Author: Damián Roca Marí

Advisor: Mario Nemirovsky

Draftsman: Josep Solé Pareta

Year: 2012-2013

Table of contents

Collaborators

Universitat Politècnica de Catalunya (UPC) – Computer Architecture department

The **Department of Computer Architecture** teaches and promotes research on topics related to computer architecture, computer networks and operating systems. Members of the DAC have played a significant role in setting up and managing research centers and groups such as the Barcelona Supercomputing Centre (**BSC-CNS**), the Advanced Broadband Communications Centre (**CCABA**), the European Parallelism Centre of Barcelona (**CEPBA**), the CEPBA-IBM Research Institute (**CIRI**), the Data Management Group (**DAMA-UPC**) and the Security Team for the Coordination of Emergencies in Telematic Networks (**esCERT**), as well as in attracting companies to the UPC (**Compaq Labs, Intel Labs**).

The NaNoNetworking Center in Catalonia (N3Cat) has been created as an initiative of Prof. Ian F. Akyildiz and Prof. Josep Solé-Pareta at UPC (Universitat Politècnica de Catalunya) with the main goals of carrying fundamental research on nanonetworks and educating and training the new generation of students.

Barcelona Supercomputing Center (BSC)

The Barcelona Supercomputing Center-Centro Nacional de Supercomputación (BSC-CNS), established in 2005, serves as the National Supercomputing Facility in Spain, hosting MareNostrum. BSC-CNS mission is to research, develop and manage information technologies in order to facilitate scientific progress. The BSC-CNS not only strives to become a first-class research center in supercomputing, but also in scientific fields that demand high performance computing resources such as the Life and Earth Sciences. Following this approach, the BSC-CNS has brought together a critical mass of top-notch researchers, high performance computing experts and cutting-edge supercomputing technologies in order to foster multidisciplinary scientific collaboration and innovation.

Acknowledgments

To Mario, my supervisor, for his vast knowledge,
his incredible patience, and his coaching
to conclude this work

To Josep, my supervisor,
for his tips, his help and his support

To Eduard, Albert, Sergi and all the people of the N3Cat
for his work on the Graphene project
and their correspondent contribution to this thesis

To Sasa Tomic, for his invaluable help with
with the M5 simulator

To Judit Giménez and all the people from the BSC
that teach me about their tools

To my parents for their financial support
to the author of this thesis ;)

Abstract

Computers, and multicore processors in specific, need cache memory to improve memory bandwidth and overall performance. There are different types of cache (private and shared) divided into different levels of hierarchy. Keeping coherence and consistency of shared values in these caches is a major performance bottleneck on multicore systems. To address this issue, there are several protocols that invalidate or update these values when a core needs to modify them. But these protocols require broadcast communication (or similar) that in today NoCs represents a big cost in terms of cycles.

In order to improve this bottleneck, the first step in this research is to know and have an approximation of the target that represents these invalidations in the terms of performance of the system. To obtain that estimation is mandatory to use programs or simulators of a real process inside a multicore/multithreaded processor to visualize the communications between these cores and the effects of sharing a part of the space address. The reason is that these invalidations are produced by keeping the coherence between different copies of the same variable (shared space).

Once that we have a simulator that allows us to see the communications we can make different configurations to emulate a real processor in different scenarios. With these cases, we can obtain how the number of invalidations is modified depending on the parameters of the system (number of cores, size of cache memories, etc) and the applications which are running. Due to this, the results can vary for different applications since each of them uses the shared memory space in a different way.

With this information we can elaborate some statistics to extract the first conclusions and fix the bases for future work. These results also enables us to study the scalability of the actual models to see what would happen if we have more than 1000-core processor because the actual simulators do not support such high number of cores.

1. - Introduction

1.1. – Context of the project

Nowadays, companies and scientists working in areas such as complex simulators, finance, meteorology or genomics are generating extremely large datasets. Such datasets are increasingly gathered from a variety of automated systems: devices, sensors and social networks, reaching sizes in the order of petabytes. These very large datasets are referred to as BigData. It is commonly accepted that efficiently operating and analyzing BigData is a key basis when considering competition, innovation and research. In the business domain, decision-making can reach an unprecedented level of accuracy by taking advantage on a quasi-global knowledge of the context, while inexpensive simulations on top of the business data can help improve management decisions. In the research domain, analyzing huge amounts of experimental information allow finding new and unexpected trends and patterns. In this context, Information and Communication Technologies (ICT) must urgently provide a scalable solution to process BigData in order to avoid slowing down progress, innovation and research in these transverse and pervasive domains.

Existing ICT techniques to process very large datasets are based mainly on parallelization. Parallelization has been the natural trend in microprocessor architecture design for the last decades, leading towards the recent emergence of multicore and even many-core (more than 16 cores) architectures. As the technology downscaling allows the integration of even more processing cores in the same chip, the resulting Chip Multi Processors (CMP) could take advantage of the expertise gained in massive parallel computing for the treatment and analysis of such very large datasets. Recently, two very well-known microprocessor companies have prototyped CMPs with 1024 and 2048 cores, respectively.

When considering parallel processing of BigData in CMP architectures, data coherency and consistency are critical issues. Indeed, as data is modified within a core, a significant amount of on-chip communication is needed in order to guarantee coherency and consistency among the thousands of cores in a chip. As a result, current CMPs are equipped with an inter-connection network called Network-on-Chip (NoC). The existing NoC paradigm has been proposed as a scalable and efficient manner to communicate cores by means of wireline direct and switched/routed schemes. In this direction, current research efforts are focused on the employment of high-bandwidth RF transmission lines and optical links. Indeed, the main bottleneck in CMPs has shifted from computation to communication and, as a result, the NoC has become a critical component which determines the performance of the microprocessor.

Even though the aggregate bandwidth provided by such approaches scales well as the number of cores increases, a huge percentage of the NoC traffic consists of short one-to-

many (broadcast/multicast) control messages used for cache coherence and consistency. Precisely, the BigData scenario requires a network architecture that supports the simultaneous transfer of a very large number of such one-to-many messages not only as a consequence of data coherence and consistency, but also due to the specific communication needs of some massively parallel programming models**.** However, the wired point-to-point nature of current on-chip network approaches (RF/optical links) renders broadcast and multicast communications excessively costly. Therefore, *the need for such communications* creates a clear performance bottleneck in current NoC for BigData applications.

In short, current approaches to implement NoC are inherently rigid point-to-point schemes that limit the implementation of broadcast messaging and reconfigurable networks. As a result, there is a need to rethink the entire architecture of CMP in order to enable them to efficiently handle BigData. To this end, the targeted breakthrough is to provide scalable native broadcasting and multicasting capabilities to NoC by incorporating nano-antennas for on-chip wireless communications and to design new software (SW) and hardware (HW) multicore architectures that take advantage of this unique feature. We refer to this new communication technique as Graphene-enabled Wireless Network-on-Chip (GWNoC, see Fig. 1) and propose it as the enabler of our long-term vision. Deployed over a state-of-the-art NoC, the main benefit of GWNoC is that the utilization of graphene-based plasmonic nano-antennas (*graphennas*), which enable both size compatibility with each core and communication in the terahertz (0.1-10 THz) band, therefore providing means for the implementation of broadcast, multicast and all-to-all communications in massively parallel environments. The GWNoC paradigm does not aim to replace current NoC approaches (3D, RF transmission lines or optical links), but to complement them by providing an efficient scalable broadcast/multicast channel. This wireless channel will allow us to revisit the current memory management and consistency/coherency mechanisms, to provide new software APIs that will greatly simplify parallel programming and overall, to change the way we think about computer architectures.

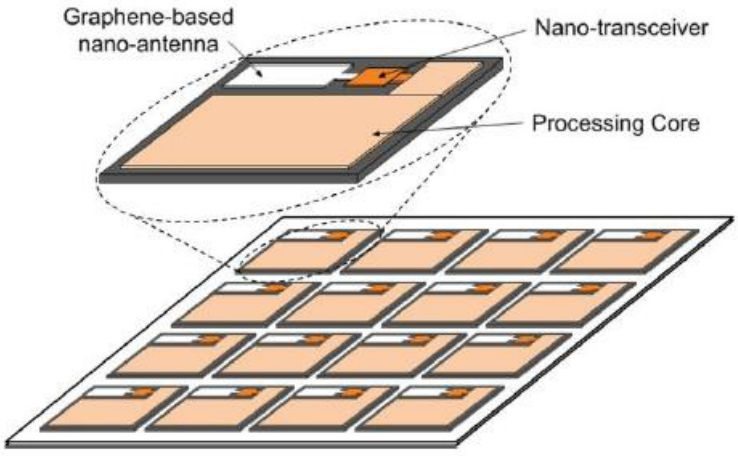

Figure 1 – Graphene- enabled Wireless Network-on-Chip

A GWNoC presents many advantages over existing NoC approaches. First, given its reduced size, (multiple) graphennas can be effectively integrated into the processing cores without incurring into a large area overhead. Moreover, using a GWNoC, these cores will be able to communicate via one-hop wireless links, regardless of the distance among them. Furthermore, the terahertz band offers enough bandwidth for transmissions of up to several terabits per second. But beyond these basic advantages, the inherent flexibility and broadcast/multicast capabilities of GWNoC opens a vast range of possibilities in terms of architecture design. Since most of the on-chip traffic in multicore environments is composed of short, generally multicast, control messages, GWNoCs allow us to completely rethink how processing cores interact among them and with memory, with particular focus on the cache coherence and consistency protocols. As an example, it is well accepted that when number of cores on die increase, current structure of cache coherence protocols cannot be efficient even for handling cache coherency within the die, since on a cache miss, the core must communicate with the memory controller and with all other cores that may contain a copy of the piece of data. This cumbersome process severely slows down write and I/O operations and creates a large amount of control messages, causing significant amount of power, pipeline stalls and reducing the overall performance. Even worse, one of the most expensive parts, in terms of performance and power, of implementing cache coherent protocols in large scale architectures is the invalidation phase of cache lines when exclusive access is needed, or in the presence of I/O traffic. With GWNoCs, these protocols could be easily re-implemented by means of a simple broadcast wireless message; this would also result in simplified and more efficient parallel software APIs. With respect to energy consumption, even though the energy per bit figures of wireless on-chip communication are projected to be greater that of its electrical or optical counterparts, the aforementioned GWNoC capabilities will lead to architectures needing way less communication per instruction, thus substantially enhancing the performance and power efficiency of CMPs.

In the GWNoC there are partners involved: the BSC-CNS, the UPC and the KTH. The first one is taking care of the computer architecture part, the second one works on the communication layer and the graphenna and the third one works on the graphene as a material.

In this context and from the computer architecture point of view, the first step is to know the estimated target in terms of performance. Related with that, this final project career wants to put the novel values of the possible improvements for future works. But now, it just consists of the study of the actual models without any changes and in the future, the architecture and the algorithms that run over these processors would be modified to obtain a better improvement and exploit all the characteristics of this new technology.

1.2. – Objectives of the this thesis

The single chip multicore/multithreaded processors present a lot of constraints in its design and fabrication. This thesis wants to study these issues in order to determine the most critique and feasible one. It seems that this constraint is the memory system, and its particular case of the cache memory. When the number of cores gets bigger and bigger, the protocols that guarantee the coherence are affecting the performance of all the system. These protocols are necessary in the actual models to ensure that the value read is the last modification done. And when the number of cores is huge, this is a critical factor. This fact is provoked by the shared values between cores. To ensure this, the protocols need to establish a procedure that consists in the invalidation of all the values of a variable before one core can modify it. The read operation is not so critical due to not require an invalidation.

For all the exposed before, this thesis focuses on the measurement of the number of generic invalidations (completely independent of how many cores get affected) while running some standard multicore applications. Obviously, the number of invalidations depends on the shared space between cores, the application which is running, and the system that supports this execution. With this, we are going to get the information to observe their influence to the performance of the full-system.

To get this objective, we have to use a simulator which allows us to emulate a full system with a multicore processor and with a few levels of cache. To force the interaction between the cores each thread will be pinned to a core, guarantying the existence of a shared space between them at the same period. With this simulator, we are going to change the parameters of the system with an easy way to study the impact of the different applications that are running.

And to see the performance of an application running on this system, we are going to need a generic benchmark with which we can simulate other applications with the modification of the input parameters. Therefore, with one benchmark we can test a lot of scenarios and cases.

The final objective is to obtain some statistics with the results of these simulations and to observe the scalability if we are working with a 1000-core through estimation via the previous results.

1.3. – Organization of this document

This is a short guide for the readers of the present document to clarify the contents included in the chapters:

- Chapter 1: is the introduction to this project and the positioning inside the context. Also includes the expected results of the present work.

- Chapter 2: describe the problem in which this project is based. These issues involve the memory system of a computer, the cache memory and his way of working.

- Chapter 3: describe the different tools chosen to realize the project. It includes the description of their main characteristics and capabilities.

- Chapter 4: explain the simulations framework; the modifications to the simulator and the benchmark. Even it includes a more detailed description of the problem that we want to observe and study.

- Chapter 5: present the results obtained through the simulations and explain them.

- Chapter 6: explain the final conclusions obtained with the realization of the present job.

- Chapter 7: describes the future steps to realize like a continuation of this work

- Chapter 8: show the bibliography used to make the entire project and the present document.

2. – Problem formulation

Technology is advancing faster and faster, included the silicon technology that enables the implementation of microprocessors and all the electronics in general. In the past, the main advances in the field of the computer architecture are to increase the clock speed of a single processor in order to execute more instructions with the same time. With the material technology, in that case the silicon, each time the industry was able to build smaller transistors that improve the area used. With both improvements the computers get more computational power with lower resources. But all has their own limits and in this case arrives when the technology rise to a certain clock speed, around 4 GHz, having problems with the heat and the power consumption of those processors. Therefore, the technology is in a point in which is really difficult to continue with the improvement following that way.

The research community arrived to the conclusion that the new way to obtain better and faster processors pass through the use of different cores inside a processor. These processors are called the Chips Multi-Processors (CMP). Nowadays the improvements are based in how many cores can work together inside a processor. With the time, the technology is enabling to put more and more cores on a single die. Each core has some private resources, while some others are shared between them to improve the performance of all the system and at the time optimize the cost of the production of those chips. An example of the shared resources can be the memory or some I/O buses.

Cores of a CMP generally are executing different parts of the code of a program, which is very useful, or also can execute different programs in a parallel way. These reasons hold the new trend in the microprocessor architecture, parallelism. Parallelism consists in the execution of some flows of code at the same time. With that technique, the processor can execute the instructions faster and only have the limitations of the locks, which can happen for example due to the wait for some result of another program.

Even, each processor can be able to execute different threads. These threads are able to execute a process themselves dividing the resources of the core to finally get completed all of them. And a thread also can execute a part of the code of a bigger program while another thread is executing another part. Those threads can share some resources between them, like some memory space; or can work with separated resources, like independent processors. However, we are not going to explain that in more detail and from this point, we are doing the assumption that each core can run a single thread which is easier and completely enough to understand the problem that is studied here.

The fact that each processor has to run at least one thread to take advantage of the multicore/multithreaded system is obvious because if a single core is running all the

threads there is no place for the parallelism and it has no sense to put more cores to not make use of them.

The most important and obvious system that need to be studied in the CMP case is the memory system. The existence of multiple cores brings the problem that different cores can work with the same information and also modifying it and these implies new problems that need to be considered in order to guarantee a correct execution of the programs. This fact depends on the memory system built at the computer. There are two ways to implement the main memory system in these processors:

- **Centralized shared memory** (see Fig. 2): each core (or processor) has its own cache but the memory is shared between them. This can be a good solution for a reduced number of cores, but when we increase its number is a problem because these memory need to process and execute a lot of petitions (read and write values) from all the cores. Even the media that support this communication, like a bus, can suffer with a big number of cores. The only advantage is that the central memory controls the accesses to the values in a symmetric way with all the cores. But, as we mention before, the scalability of this system is poor.

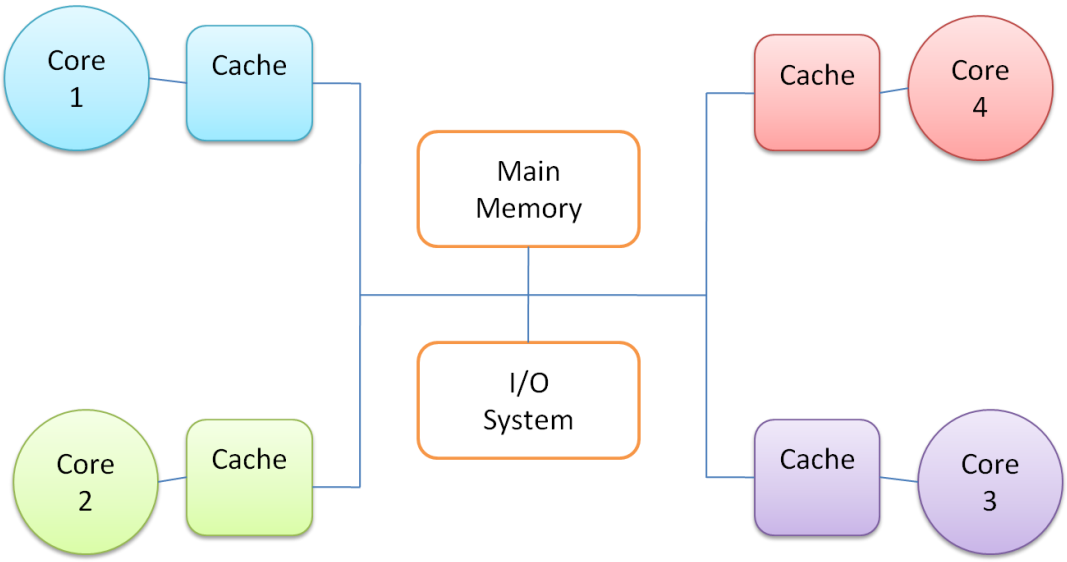

Figure 2 - Centralized shared-memory multiprocessor

- **Distributed memory** (see Fig. 3): in this option the memory is distributed among the different cores, maintaining the cache at each core. With this solution the system can provide the bandwidth required and it is not collapsed due to the huge amount of petitions to read and write values to the memory. So, this solution scales much better than the centralized memory while the number of cores is increasing. The problem here can be found in the interconnection network between the different parts of the distributed memory that needs to

support the communication between the nodes of the memory. Another big issue is the communication between the cores because now each of them has its own space of memory.

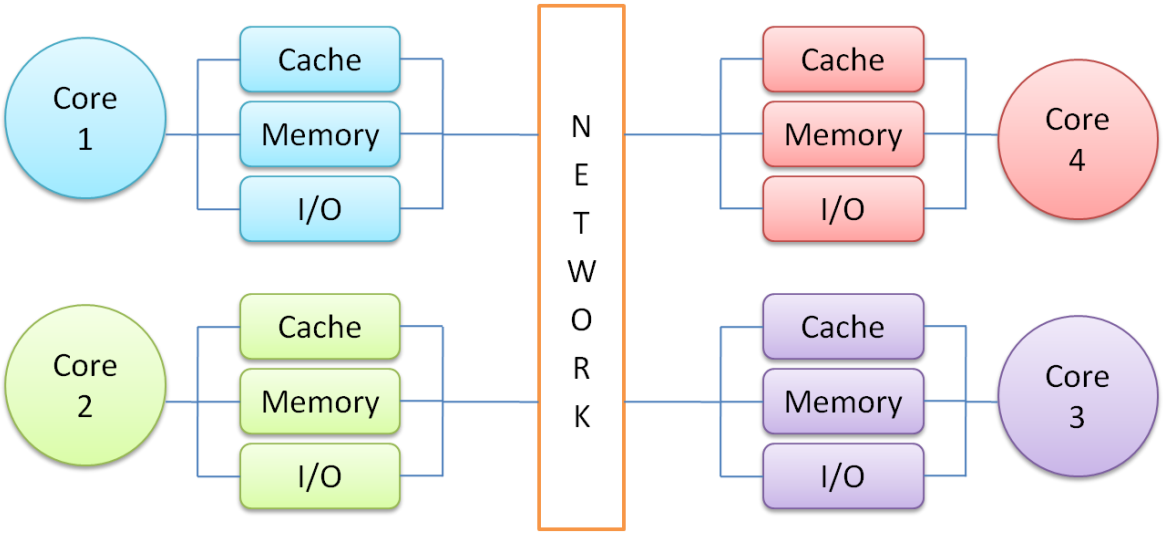

Figure 3 - Distributed memory multiprocessor

Here appears the necessity of having a communication between all the cores, not only to share data between them, but also to keep the synchronization for example. In order to guarantee a correct execution of the instructions there are two solutions:

- **Message Passing**: in this solution the cores are communicating between them using specific messages that contain the requests and the responses to obtain the data that they need from the other. This solution implies an overhead at the system to sustain that communication, but this is completely necessary. An example of those messages is the Message Passing Interface (MPI) that defines the messages that can be used.
- **Shared memory space**: this option does not imply the existence of a centralized memory; it is just that some processors have access to the same physical space of memory. With that option, a core can modify a value and the other cores that are also using it can "see" the modification and the new value. To get it is necessary the existence of some protocols to guarantee the integrity of this data

Inside the memory system, a subsystem that requires special attention is the cache system. This system has the main objective of provide enough bandwidth to satisfy the core requirements and also reduce the latency of all the system. The cache is divided into different levels depending on their physical situation and accessibility. In the case of the multicore/multithreaded processors each core has a private cache, normally named the L1 cache, which is really small but incredibly fast. It is private because only that core can see the values included in it and modify the variables. After that level, it can have more levels of cache, like L2 and L3, which are shared between different cores following a hierarchy. Those memories are bigger than the L1 cache but slower. With the combination of all the levels, the architects obtain a good cost-effective solution to

solve the memory and bandwidth problems. To obtain a better view of this item, see the Fig. 4

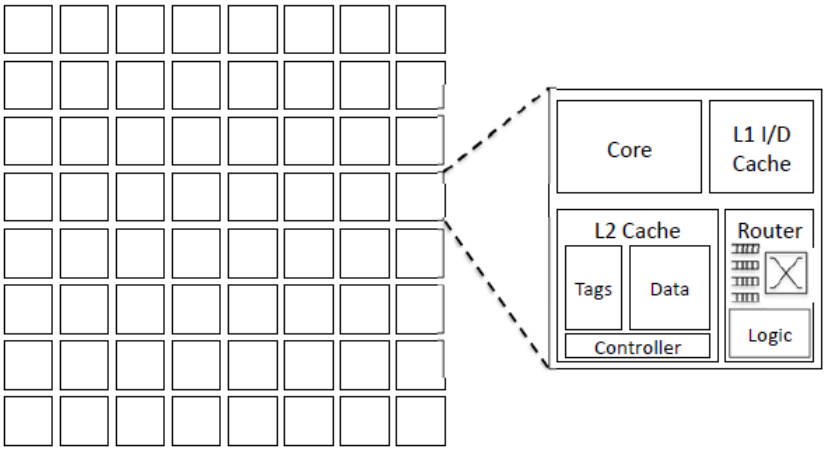

Figure 4 - Shared Memory Chip Multiprocessor Architecture [2]

The cache works on simple mode: the first access to the cache memory does not contain the value, so there is a miss (see Fig. 6) that implies to go through all the levels searching that value, and if the value is not found, the system brings it from the main memory to the cache. This new value is replicated in all the levels of the cache that implies the core that initially request this value. So, the next time that the core needs that value and request it to the cache, there is a hit (see Fig. 5) because it can find it. With that solution, the latency of the system is reduced because the cost of the access to the cache is smaller than the cost of accessing to the main memory. The problems appear when a value is contained in different caches of different cores. In that case, it is necessary to keep the value actualized for all the cores that sees that value. Here it appears the concepts of the consistency and the coherence. The first one implies that the program and the system will follow the same order of writes and reads that the established by the programmer while the second one implies that the readers of the same value contained in different caches will be the last one and also the same for all of them.

In order to show these two simple steps, the hit and the miss, let us assume that we have a multicore/multithreaded system where each core has its own L1 private cache and between all of them share a L2 cache. Over this system we are going to see what happens in both cases and also see the procedure of the system and the behavior of the resources implied, like the private resources (L1 cache) or the shared ones (L2 cache and main memory)

The hit case:

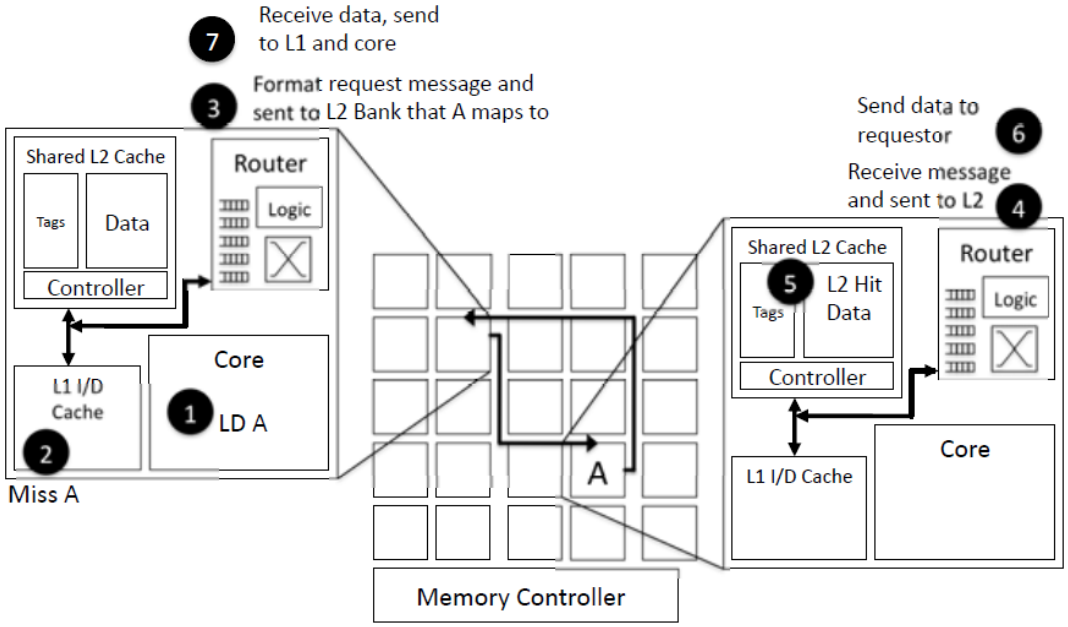

Figure 5 – Shared L2 hit [2]

In this simple case, the request to the L1 private cache of a simple core returns a miss, so it is necessary to go through the next level to find that value. Once this resource is located, the requester receives the value with a lower latency than whether it is necessary to go to the main memory.

The miss case:

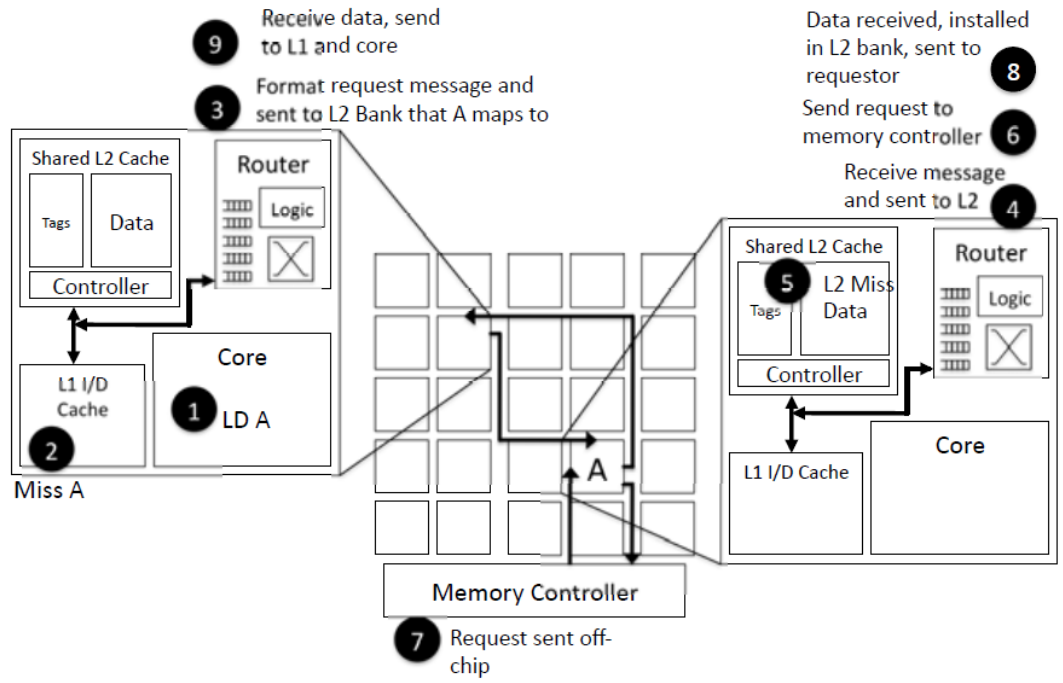

Figure 6 - Shared L2 miss [2]

That case requires more work for the system to put the value available to the core that request that value. Here, L1 and L2 reports a miss, so it is necessary to access to the main memory to obtain that value, put it in the caches and report it to the core that request it. Therefore, the latency that the cores observe is bigger than in the other case.

In order to simplify all these processes, each value in the cache has a variable that indicates the status of itself. At the same time, these statuses indicate the operations that the thread or core can do it with this variable. Here there are different protocols depending on the status that allows to the values. An example can be the MESI protocol (the name comes from the different states that can have the line at the cache). In that specific case there are four statuses allowed:

- **Modified**: The value has been recently changed.
- **Exclusive**: The thread with that value is the unique owner of this value, so if he wants to do it, he can modify it.
- **Shared**: The ownership of this value is shared between different threads, so if one of them wants to modify it, first need to invalidate all the others and obtain the exclusive.
- **Invalidated**: The value is invalidated, meaning that it can be used and that it is waiting to receive the new value because a thread is going to modify it.

If a thread wants to modify a value, first of all is necessary the invalidation of all the others copies of this value and once that he has it in an exclusive way, he can change it. Later, the system has to send the new value to the others threads that had it previously in order to continue using it in the execution of the program.

To maintain the coherence and the correct working of these protocols there are different ways to provide coherence through coherence-protocol. The most important are:

- **Directory based**: this directory contains the status of the values contained in the cache. So, when an invalidation is needed, with the consulting of that directory it can be known with cores have this value and proceed with the invalidation in an easily way. This solution implies an overhead, but with a big number of cores it has a lot of sense. Also, the directory can be founded at each core.
- **Snooping**: each single cache has a copy of the status of the variables, but it does not know who the other owners of those values are. So, that solution requires a broadcast medium to intercommunicate all the cores in order to proceed with the invalidations. The scalability is really bad because the broadcast is simulated through buses that need an access control. Therefore, with a huge number of cores this solution is completely impossible.

An example of the snooping protocols can be seen in the Fig.7:

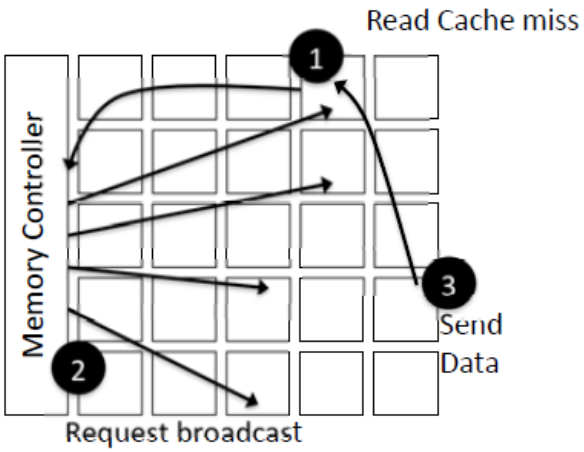

Figure 7 – Procedure in a snooping protocol [2]

While the number of cores is kept small, both protocols work fine (both of them) always than the system is able to provide a good broadcast medium between all the cores. But the problems appear when the number of cores increase and increase since that shared media is normally a bus or another network without the capacity of a real broadcast. Therefore, to do the invalidations are completely necessary these diffusions to all the cores. And now the load of this media is increasing and increasing and for a large number of cores this solution is not going to work. Even in the case of the directory-based protocol it is necessary to send messages from one-to-many because the only thing that changes is that now we know the other cores that have this value. In consequence, this is a major bottleneck in the architectures of the multicore/multithreaded processors that affects the performance of all the system.

Even there is a big issue implicit with the invalidations, their cost. With a reduced number of cores the cost of an invalidation is around one order of magnitude bigger than a normal memory operation. When we are increasing the number of cores the cost of an invalidation is also growing. So, there is a relation between the cost of the invalidations and the number of cores. This implies that with systems with thousands of cores the cost of an invalidation is really big.

And another problem is that to accomplish this invalidation process is necessary to receive the acknowledgement of every core involved at the process. Therefore, with systems with more than 1000-cores the invalidation cost is an issue but the reception of these acknowledgements implies another big issue that affects to all the system.

This project is focused on the problem formulated in this chapter and more specifically on the solutions based on the snooping protocols. The objective is the obtention of an estimation of the number of invalidations as a function the number of cores, the size of the space shared between all the cores or threads in order to provide the first input to the context of this project to estimate the target of use the GWNoC in the computer architecture of the next processors.

3. – Simulation Tools

In the simulation tools is necessary to divide it in two different parts. In the first case, we need a simulator for the full system of the computer (processors, memory, etc) and in the other side we need an application to test the performance of that system for different configurations, like different shared-spaces of memory between the cores or threads.

3.1. – Gem5 [9]

Figure 8 – Logo of the Gem5 simulator

To be able to test different problems over a machine is completely necessary to use a simulator because with the machine it is impossible to change the number of cores for example or the size of the shared space between them. In our case, we chose the Gem5 simulator due to their vast use inside the research community in the field of the computer architecture and the robustness that offers.

The overall goals that Gem5 provides are:

- **Flexibility**: to adapt in an easy way to all the necessities of the research community. So, the researchers are able to adapt the simulator to their needs and run and study their solutions with the more accurate model possible. To allow that there are three mandatory aspects: the CPU model, the memory system and the System model.
- **Availability**: this simulator wants to be used by the researchers, the scientists and the corporations and companies.
- **Collaboration**: join the efforts of all the people that use and improve this simulator in order to obtain an implied community working together to the same objective.

The simulation capabilities that this simulator presents are really good. It provides the most common used Instruction Set Architecture (ISA) of the market, like the x86, alpha, MIPS, etc that can run over the generic model of the CPUs. And of course, it allows the users to modify the parameters of the system, like the size of the cache memories, the number of cores and a lot of others parameters too.

The multicore/multithreaded integrated in the system allows to run simulations until 32-core, including all the systems of a computer. For all these reasons, we are in front of a full system simulator that allows us to extract statistics of the performance to have an estimation of the expected impact of the measures that the researchers want to try to implement.

3.2. – Eigenbench [10]

The first step before run commercial applications over that simulator is to run some benchmarks to have an estimation of the target that can be reached with a generic application and if the results are promising, try over some concrete applications.

Eigenbench is a benchmark for Transactional Memory (TM) systems. In the default configuration brings the support for the SwissTM [6] and for the Transactional Locking 2 [5]. Even it includes a non-protected version that does not guarantee the atomicity of the operations. To be able to test and observe the performance of the TM over the systems they define eight orthogonal characteristics: concurrency, working-set size, transaction length, pollution, temporal locality, contention, predominance and density. With the combination of these characteristics it is possible to test all the others characteristics and capabilities of the TM systems.

The functionality of this code to test the TM systems is easy to understand. The different threads that are running over different cores are working with three arrays contained in the memory. The first one, and the most important, is the hot array that is shared between all the cores. So, in that case, all of them are forced to interact between them and here is where will be the conflicts due to accesses to the same variable at the same variable and the modifications of them. This array is controlled by the TM system that the user decides to implement. On the second array each thread has its own space, by it is also controlled by the TM system. The third one is like the second array but in that case there is no control over the memory. The last two arrays are used to simulate the performance. Through the Fig. 9 and the Fig. 10 it can be observed the situation of the cores and threads respect to the arrays.

The user can parameterize the number of reads and writes that the program will do to each array in order to simulate some characteristics of the situations that they want to try it. These operations are completely randomized in their order, trying to simulate a normal program. The user also can choose the size of the three arrays. The number of threads that are going to execute these instructions is also a parameter, but in our case and as we explained before to take advantage of the multicore/ multithreaded system we are going to use *n* processors with *n* threads. Even, the user can define a number of loops that indicates the number that all the process is going to be repeated. This tool also allows the user to define specific parameters for each thread that is executing the code, but in our case we are going to suppose that all the threads have the same parameters.

In the case of a 4-core CMP:

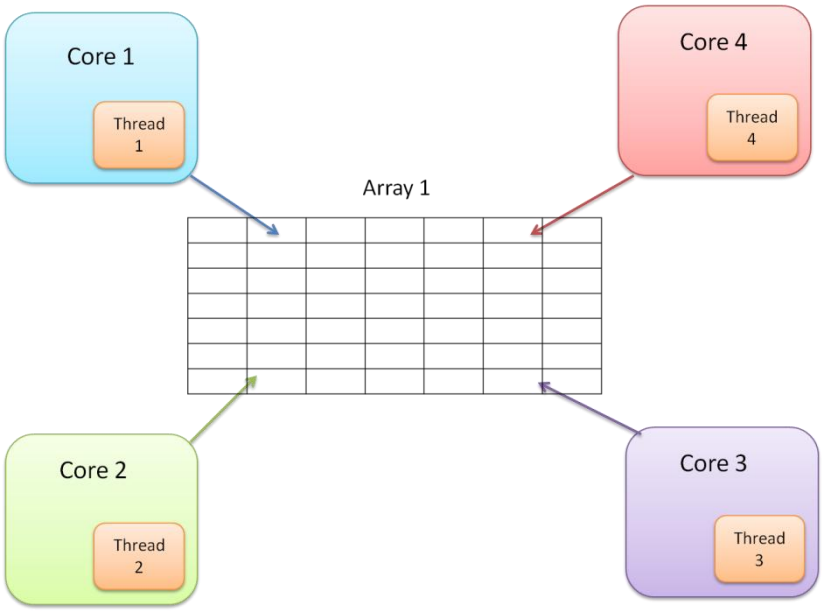

Figure 9 – Scheme of Shared space at array 1

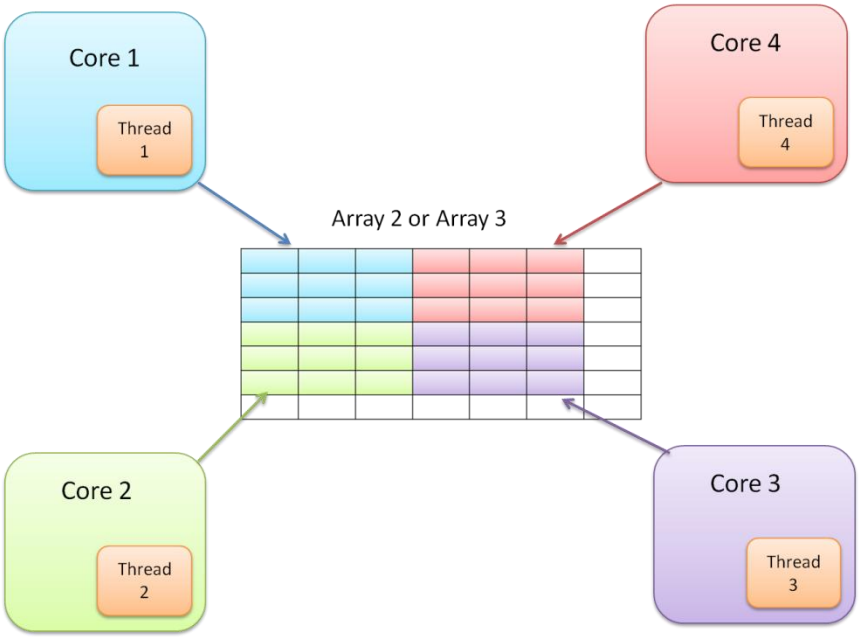

Figure 10 – Scheme of Shared space at array 2 and 3

4. – Simulation framework

With the tools described in the Chapter 4 of this document, the user has the basic mechanisms to begin with the simulations to obtain statistics. But the first step is the adaption of these tools to the requirements that we need for our tests.

With the following list we are going to explain the modifications realized to the simulator gem5:

1. **Pin the threads to each core**: in order to force the interaction between all the cores at the shared memory space, we simulate n processors with n threads. But there is the problem that if the threads are too short in execution time, the core of the system will put all of them in a single core, like a queue that is processing a list of elements in order. Therefore, in this specific situation the cores are not accessing to the shared space of memory at the same time. To solve that and force this interaction, each thread is pinned to a single core. The result of that is the running of all the threads at the same time, with their correspondents accesses to the memory space which can generate the invalidations that we want to see to quantify it.

   Imagine the case of a simulation with a four-core processor with a fully connected network between them, which is we can see at the Fig. 9 to have a better perspective of what is happening with the modifications.

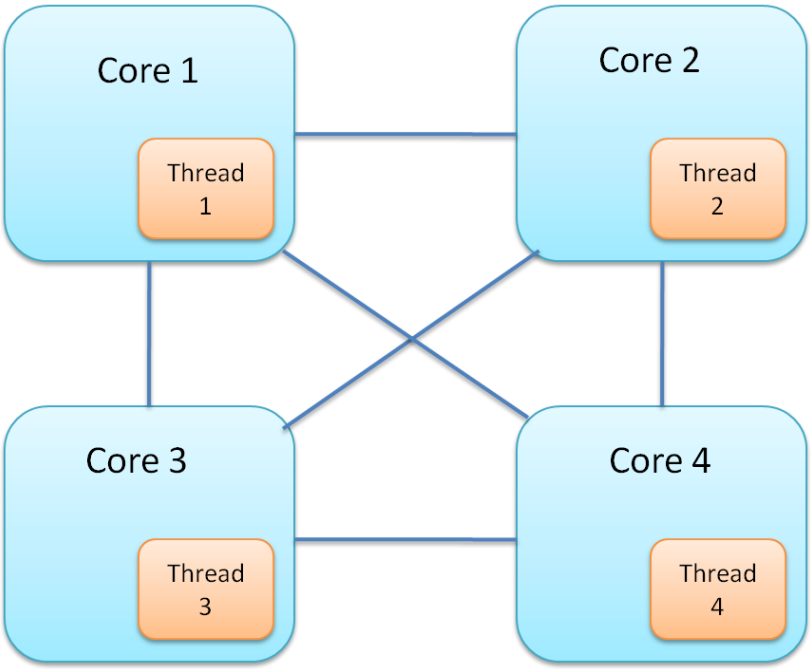

Figure 11 – Pinned threads at a 4-core processor

2. **Count the number of invalidations**: this is the point in which the main part of this project is focused, so it is really important to modify the simulator to be able to count the invalidations. To do it in a properly way, the code was modified in order to count the generic invalidations and the number of invalidations that a core receives and have to process. In addition to these two counters, the statistics take care of the hits and the misses for the levels of the cache and for the main memory.
   The first one, the generic invalidations, is the invalidations that a core generates independently of how many cores get affected for it. And the second one, the invalidations that a core receives is the petitions of the system to invalidate a value in its cache. Let us put some examples to understand it better:

   - *Multicore with 2-core* (see Fig. 12): in that case, when we present the results of the simulations it is really easy to check the correct functionality of these counters. In this situation, an invalidation generated by the core 1 just can affect to the core 2. So, there is a cross relation between the two counters that we implement, the generic invalidations and the invalidations received for a core.

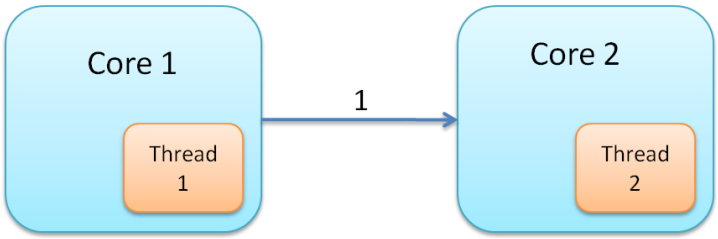

Figure 12 – Invalidation process at 2-core CMP

   - *Multicore with 4-core* (see Fig. 13): in that case the situation is a bit more complicated. An invalidation generated by the core 1 can affect one core, two cores or three cores. Even, it can happen that there is no invalidation, but the probability of that is getting smaller when the shared space is small or the number of threads increases. So, the number of cores that can be affected by a generic invalidation goes from 0 to n-1, when n is the number of total cores of the system. In that case, to check that the modification of the code is working correctly, always the total number of generic invalidations has to be equal or smaller than the total number of invalidations received by all the cores. The reason is that each generic invalidation at least causes one invalidation to one core.

This case also can be used as a proof that demonstrate that the two counters implemented at the gem5 simulator are more than enough to obtain the statistics that we want from the entire system.

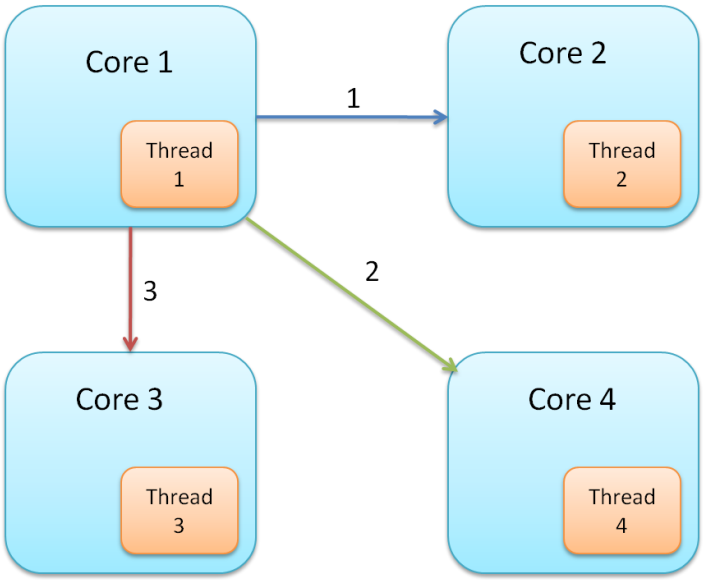

Figure 13 – Invalidation process at 4-core CMP

- *Multicore with n-core* (see Fig. 14): in the generic case from the number of cores point of view, the reasoning is the same than for the 4-core system. A core can generate generic invalidations that affect from 0 to n-1 cores.

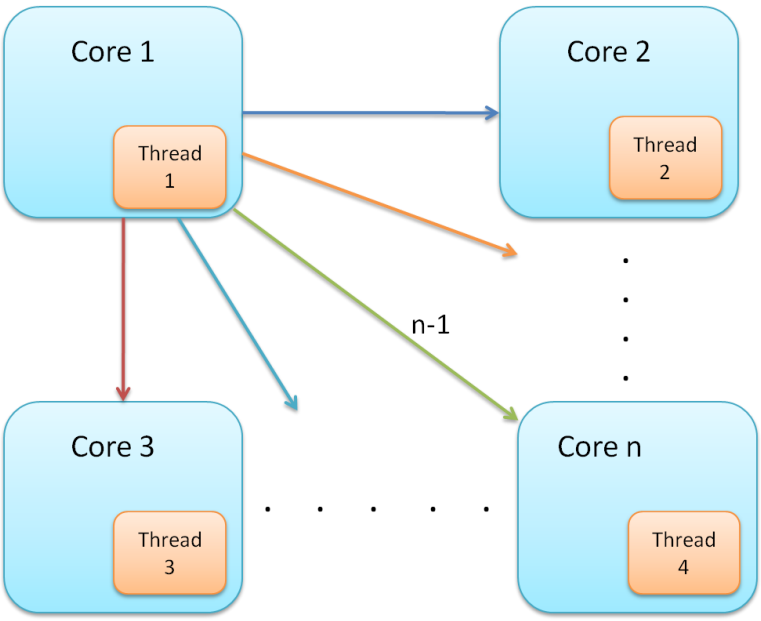

Figure 14 – Invalidation process at n-core CMP

With these modifications the simulator gem5 is ready to test what we require from the computer system. The specific situations that we want to run are simulations in which we can modify the number of cores without any problem and the gem5 enables us to do it until 32-core, which is enough for this first approach study. An important note about the invalidations is that are counted by lines, not for each specific variable (as all the systems and simulators do it like here).

The architecture used for the simulator is Alpha due to its stability and the operative system which is running over this simulated machine is Linux. The speed of the clock that controls all the cores is 2 GHz.

The configuration chosen for the cache memory is taken from a real computer to give the maximum reality to the simulations and the tests. It can be seen at the table 1:

Parameter	L1 cache (private)	L2 cache (shared)
Size	512 KB	16 MB
Sets	128	4 * 1024
Ways	8	16
Bytes per line	64	64

Table 1 – Configuration of the cache system

When the simulator is ready is the time to prepare the benchmark in order to have the application over we are going to run the tests. As we mentioned before, the benchmark chosen is Eigenbench.

With this tool the options that need to be implemented and chosen are the kind of software Transactional Memory (STM) that is going to be implemented. The two options that are provided with the source code of the benchmark are SwissTM[6] and TL2 [5] (Transactional Locking 2). But there are other options like TinySTM. In our case, to simplify and make the tests faster in order to observe the behavior of the system, we used the unprotected version. With this option, we obtain an upper bound of the available performance. It is important to note that the bound is loose because there are no atomic operations guaranteed.

Once that we have clear which is the configuration of each tool, it is time to execute Eigenbench inside the gem5 simulator, like if a commercial application is running over a server or a computer. The requisites are the installation of some packages and libraries to support the functions of the benchmark and the correct configuration of its parameters to adapt it to the machine that we have. Remember that is necessary to

compile all the files of the application for the Alpha architecture with the static option because the executable files generated will be integrated inside the simulator to execute the tests in the conditions that we want.

To study the scalability of the system we need a huge number of cores accessing to a big memory-shared space. But, as we saw before with the simulation we can arrive to 32-cores. So, in order to get our objective, we can put a small shared space and with a reduced number of cores we are going to obtain similar results to the wanted situation. With this, we obtain a good results using less computational power for the simulations.

Continuing with the configuration of the parameters of the benchmark, we are interested in the first array which is shared between all the threads. So, the reads and the writes to this thread are really important. With the reads it is impossible to generate an invalidation because you only want to see the value not to modify. So, to generate invalidations we need the write operation over this array. The other two arrays, the second and the third, conserve some operations read and write to simulate that the program is doing another operations without sharing memory space, so it helps to the performance of the test.

Related to the statistics that we are going to obtain is important to know that the booting of the system is not taken into account at the final reports in order to obtain the results only for the application running over the system (without distortions)

With the previous combination of elements we have all the necessary to run and extract the statistics we need to see the behavior of the system in determine circumstances and extract the conclusions for this project.

5. – Simulation results

The schedule followed in this chapter is in first place to present the parameters and later expose the results obtained. This process is repeated for the different simulations realized to complete the thesis.

5.1 – Simulation 1

Parameters for the simulation 1

Number of threads: 2

Loops: 100

Size of Array1: *512 KB* (65536*64)

Size of Array2: 8 MB

Size of Array3: 64 KB

Reads to Array1: 10

Writes to Array1: 10

Reads to Array2: 10

Writes to Array3: 10

Reads to Array3: 10

Writes to Array3: 10

An important note is that as we can see in the previous square the total number of writes to the Array1 per thread are 1000 operations (writes_to_A1 * loops). However, the number of generic invalidations can be bigger than this number due to the counter increments one when the core requests an invalidation but some cores that are requested to invalidate cannot do it because they are using these values, so the invalidation need to be repeated. The number of received invalidations is completely normal that can be bigger than 1000 (explained at chapter 4)

The number of cores is the same than the number of threads, as we explained before, to take advantage of the parallelism in the system.

To show the results we chose a ratio between the invalidations and the writes that shows us an important aspect: how many of the writes to the shared space memory provokes an invalidation. Therefore, this ratio is an indicator of the generation of invalidations at the systems related to the shared space memory. When the ratio arrives to 1 means that each write provokes one generic invalidation. This fact is critical in terms of performance of the entire system because the cost of an invalidation is around one order of magnitude bigger than a normal memory operation.

The size of Array1 is fixed to 512 KB in that simulation and we are going to modify the number of threads. This size is a small space of memory, but for the first case and the initial results is completely ok.

The statistics obtained with the previous parameters can be viewed at the following table:

Processor	Hits	Misses	Received invalidations	Generic invalidations
P0	117858	9453	283	217
P1	76952	5399	217	283

Table 2 – Results of the first simulation with 2-core

The average number of generic invalidations is 250

And the ratio between this average and the number of writes per thread:

$$250 / 1000 = \mathbf{0,25}$$

In that case, and as we mentioned before, it can be checked that the two counters have the cross values that indicates that the procedure followed in the programming of this changes is correct.

Now, the parameters are the same, we just modify the number of cores to 4:

Processor	Hits	Misses	Received invalidations	Generic invalidations
P0	99122	7340	473	519
P1	66704	5231	426	440
P2	138192	11615	810	477
P3	103992	6251	555	660

Table 3 - Results of the first simulation with 4-core

The average number of generic invalidations is 524

And the ratio between this average and the number of writes per thread:

524 / 1000 = **0,524**

Following the same procedure, the number of cores is updated to 8:

Processor	Hits	Misses	Received invalidations	Generic invalidations
P0	149344	9501	881	844
P1	70842	5311	886	611
P2	191946	16571	1544	704
P3	161072	8198	1063	1166
P4	78257	5975	1079	592
P5	69160	5333	732	739
P6	75304	5661	857	714
P7	72110	5689	545	811

Table 4 - Results of the first simulation with 8-core

The average number of generic invalidations is 772,36

And the ratio between this average and the number of writes per thread:

772,36 / 1000 = **0,77236**

Following the same procedure, the number of cores is updated to 16:

Processor	Hits	Misses	Received invalidations	Generic invalidations
P0	295701	14726	1440	1199
P1	75924	5465	1462	840
P2	325603	26899	2882	1036
P3	309666	13898	1709	1936
P4	84427	6118	1718	657
P5	72480	5477	843	1082

P6	80825	5819	1396	964
P7	78501	5906	950	1038
P8	88625	6602	1837	658
P9	75549	5593	1397	920
P10	82179	6015	1604	844
P11	76446	5624	1017	973
P12	88599	6451	1698	793
P13	76752	5644	1204	1097
P14	82906	6027	1313	1054
P15	77397	5638	624	1042

Table 5 - Results of the first simulation with 16-core

The average number of generic invalidations is 1008,31

And the ratio between this average and the number of writes per thread:

1008,31 / 1000 = **1,008**

Fig. 15 shows how the ratio scales as a function of the number of cores:

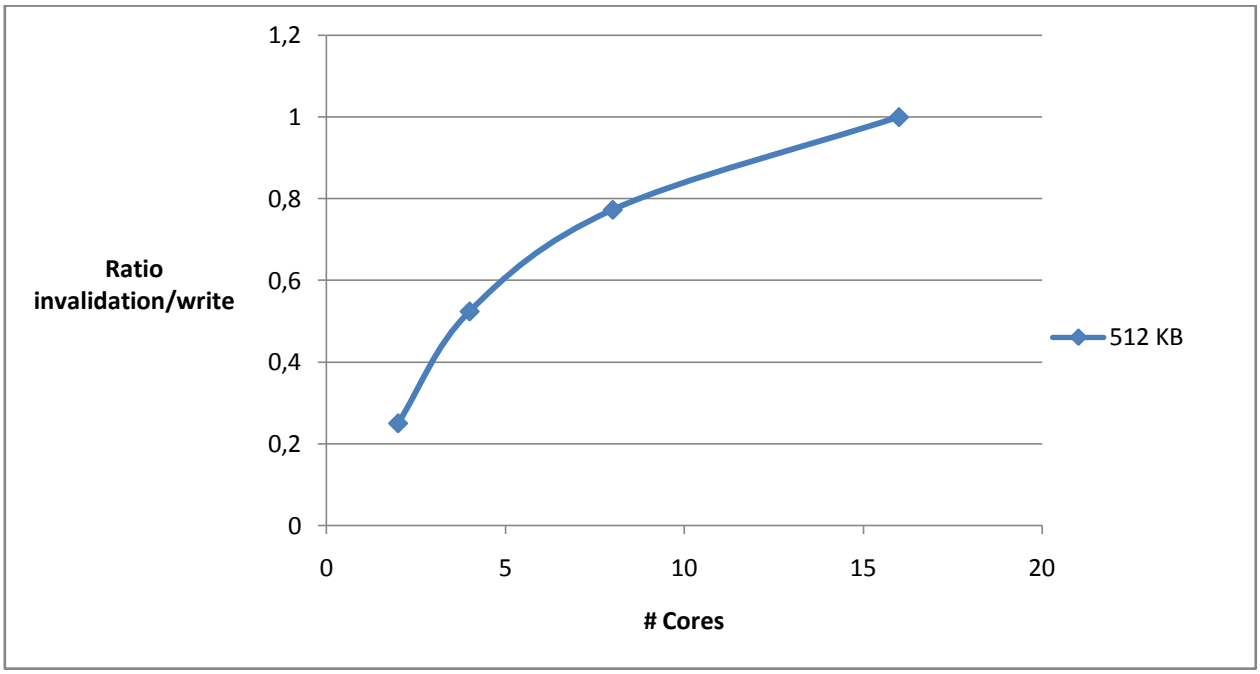

Figure 15 – Ratio invalidation/write respect to the number of cores (Sim_1)

In this simulation the space shared between all the cores/threads is small, so when the number of cores increases it is expected than the number of generic invalidations grew up fast. As seen in the Fig. 15. The rhythm of increment is really big as we can check with a 16-core CMP where the ratio is 1. So, each write operation provokes an invalidation. In this point, the performance of the system is really poor because the cost of these invalidations is really large, affecting to the entire system.

Then, the scalability for this simulated system is really bad because even with 16-core the performance is affected due to this reason, so just think that if we have 1000-core the system is not going to work with the actual solutions to this constraints.

5.2– Simulation 2

Parameters for the simulation 2

Number of threads: 2

Loops: 100

Size of Array1: *1 MB* (131072*64)

Size of Array2: 8 MB

Size of Array3: 64 KB

Reads to Array1: 10

Writes to Array1: 10

Reads to Array2: 10

Writes to Array3: 10

Reads to Array3: 10

Writes to Array3: 10

In this simulation the size of Array1 is fixed to 1 MB and we are going to repeat the same procedure than at the previous simulation and just modify the number of threads.

The statistics obtained with the previous parameters can be viewed at the following table:

Processor	Hits	Misses	Received invalidations	Generic invalidations
P0	117892	9486	224	164
P1	77005	5430	164	224

Table 6 - Results of the second simulation with 2-core

The average number of generic invalidations is 194

And the ratio between this average and the number of writes per thread:

194 / 1000 = **0,194**

Now, the parameters are the same, we just modify the number of cores to 4:

Processor	Hits	Misses	Received invalidations	Generic invalidations
P0	98570	7316	306	375
P1	66321	5212	278	313
P2	136453	11507	639	350
P3	102747	6194	392	487

Table 7 - Results of the second simulation with 4-core

The average number of generic invalidations is 381,25

And the ratio between this average and the number of writes per thread:

381,25 / 1000 = **0, 38125**

Following the same procedure, the number of cores is updated to 8:

Processor	Hits	Misses	Received invalidations	Generic invalidations
P0	149521	9560	620	648
P1	71235	5349	565	430
P2	191587	16618	1249	529
P3	161159	8262	822	928
P4	78394	6017	733	455
P5	69302	5413	491	529
P6	75538	5714	613	528
P7	72371	5758	356	586

Table 8 - Results of the second simulation with 8-core

The average number of generic invalidations is 579,125

And the ratio between this average and the number of writes per thread:

579,125 / 1000 = **0, 579125**

Following the same procedure, the number of cores is updated to 16:

Processor	Hits	Misses	Received invalidations	Generic invalidations
P0	295416	14714	1086	1019
P1	76401	5493	1031	621
P2	326581	26988	2444	858
P3	310239	13938	1491	1789
P4	81103	6097	1299	488
P5	75995	5550	606	923
P6	80424	5840	1015	744
P7	76268	5910	669	860
P8	91958	6634	1436	526
P9	76495	5650	994	704
P10	77766	5996	1157	639
P11	76884	5668	720	790
P12	83612	6388	1270	602
P13	76716	5671	854	877
P14	82551	6043	971	833
P15	77186	5719	450	889

Table 9 - Results of the second simulation with 16-core

The average number of generic invalidations is 805,556

And the ratio between this average and the number of writes per thread:

805,556 / 1000 = **0,805556**

At the Fig. 16 it is possible to observe how the ratio is increasing while the number of cores does it too.

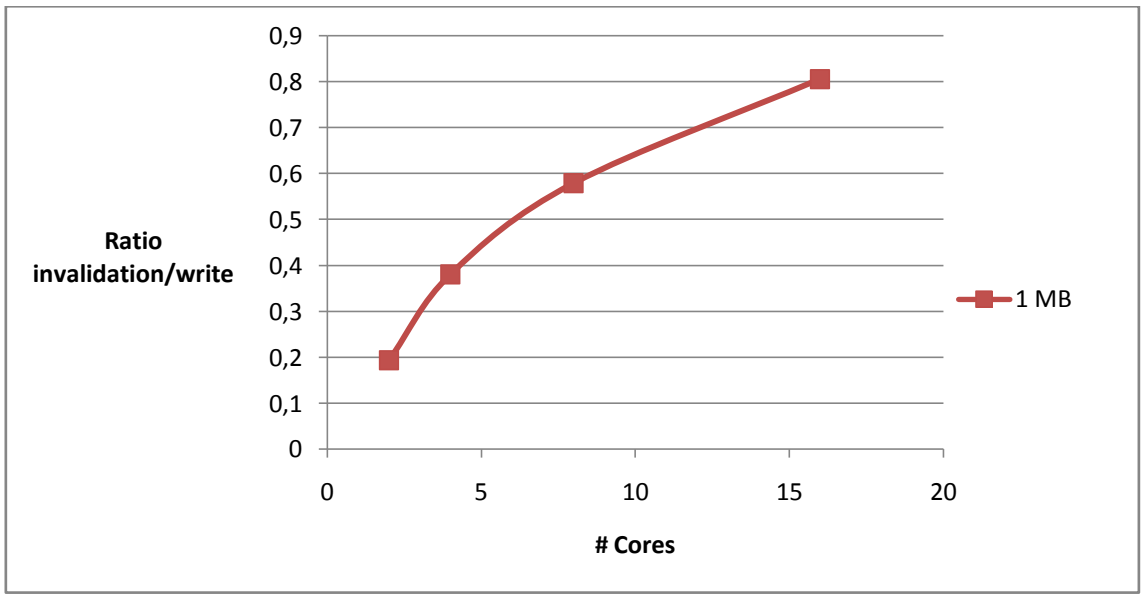

Figure 16 - Ratio invalidation/write respect to the number of cores (Sim_2)

In this simulation, with a bigger shared space between the cores the growth of the number of invalidations is slower due to the possibility of collision at the memory is lower. But even with that, with the 16-core CMP the ratio of invalidation/write is really near to 1. Therefore, the system is closely to get saturated due to the same reason than the simulation 1: the cost of the invalidations is really big and is increasing with the number of cores.

The evaluation of the scalability in that case gives the same result than the simulation 1. With a 16-core the system performance is really bad, when we use the actual protocols with a 1000-core the system is not going to work.

5.3 – Simulation 3

Parameters for the simulation 3

Number of threads: 2

Loops: 100

Size of Array1: *8 MB* (1048576*64)

Size of Array2: 8 MB

Size of Array3: 64 KB

Reads to Array1: 10

Writes to Array1: 10

Reads to Array2: 10

Writes to Array3: 10

Reads to Array3: 10

Writes to Array3: 10

In this simulation the size of Array1 is fixed to 8 MB and we are going to repeat the same procedure than at the previous simulation and just modify the number of threads.

The statistics obtained with the previous parameters can be viewed at the following table:

Processor	Hits	Misses	Received invalidations	Generic invalidations
P0	117805	9457	169	113
P1	76892	5469	113	169

Table 10 - Results of the third simulation with 2-core

The average number of generic invalidations is 141

And the ratio between this average and the number of writes per thread:

141 / 1000 = **0, 141**

Now, the parameters are the same, we just modify the number of cores to 4:

Processor	Hits	Misses	Received invalidations	Generic invalidations
P0	98791	7307	166	237
P1	66719	5261	127	172
P2	136368	11533	448	247
P3	102664	6225	274	295

Table 11 - Results of the third simulation with 4-core

The average number of generic invalidations is 237,75

And the ratio between this average and the number of writes per thread:

237,75 / 1000 = **0, 23775**

Following the same procedure, the number of cores is updated to 8:

Processor	Hits	Misses	Received invalidations	Generic invalidations
P0	149351	9492	289	357
P1	71062	5363	215	197
P2	191406	16619	819	331
P3	160939	8234	605	547
P4	78196	6036	327	272
P5	69026	5397	197	213
P6	74939	5710	270	259
P7	70434	5634	152	216

Table 12 - Results of the third simulation with 8-core

The average number of generic invalidations is 299

And the ratio between this average and the number of writes per thread:

299 / 1000 = **0, 299**

Following the same procedure, the number of cores is updated to 16:

Processor	Hits	Misses	Received invalidations	Generic invalidations
P0	295067	14723	495	559
P1	76444	5550	349	229
P2	325357	26951	1544	548
P3	310733	13975	1209	1156
P4	82115	6165	494	250
P5	74702	5541	258	307
P6	81459	5924	399	306
P7	75785	5578	264	291
P8	89413	6677	553	311
P9	76066	5698	333	254
P10	81668	6050	414	274
P11	76927	5710	294	250
P12	87740	6447	474	317
P13	76573	5687	290	291
P14	83549	6116	372	291
P15	75778	5648	210	294

Table 13 - Results of the third simulation with 16-core

The average number of generic invalidations is 370,5

And the ratio between this average and the number of writes per thread:

370,5 / 1000 = **0, 3705**

At the Fig. 17 it is possible to observe how the ratio is increasing while the number of cores does it too.

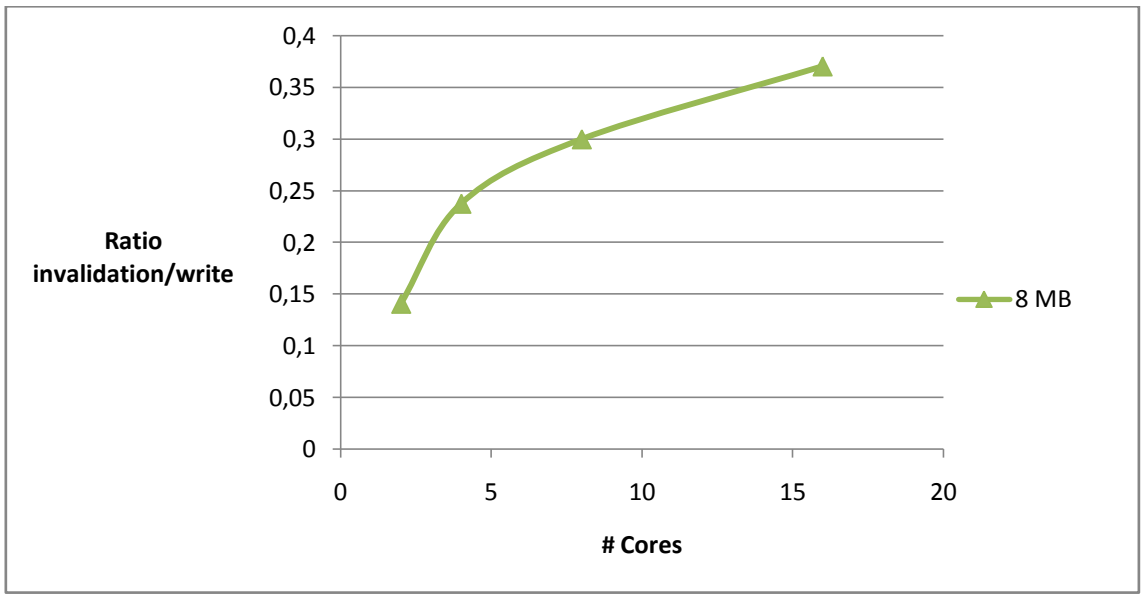

Figure 17 - Ratio invalidation/write respect to the number of cores (Sim_3)

In this case, we test a bigger size to see what happens in both faces of the coin. As it is expected, the ratio increases slower than the other two cases. This can be checker for the 16-core CMP than at the previous simulation is near to 1, while here is around 0,4. This is understandable due to the space is really big than the other cases, so the probability of collisions in this space is smaller than in the other cases.

Therefore, the scalability of this system allows operating with more cores until all the system gets saturated. Even with that, the system is going to be saturated before to arrive to the 1000-cores. Also note that the programs normally share some variables, not a so big space like in this simulation.

6. – Conclusions

The memory system and more concretely the coherence mechanisms and protocols that guarantee the correct work of all the system are completely necessary with the actual procedure in the field of microprocessor architecture, including the hardware and the software approach. But the problem is that these solutions are working and designed for a small number of cores, and when it wants to be applied to the many-core architectures it is not that they do not scale well, it is that they do not scale.

In this work we demonstrate that point and we extract the conclusions based on a simulator and a benchmark that are widely used by the research community and even the industry. Therefore, the coherence protocols are major constraint to the design of the many-core processors since it affects to the performance of all the system, and maintains the coherence is basic to guarantee the correct work of the computers.

But this is not the only constraint for the design of CMPs; there are other problems like the creation of good algorithms that really exploit the capacity of all the cores and threads of the systems, the synchronization, etc. But in this work we focus on this one because is the most obvious and the most feasible right now.

In conclusion, we cannot use the actual solutions for hundreds or thousands of cores and is completely necessary to redefine them according to new necessities and capabilities enabled by technology advances.

7. – Future Works

This final project career is the first step from the computer architecture point of view to the Graphene Wireless Network on Chip (GWNoC), explained in the chapter 1.2 of the present document.

In the GWNoC project there are three main partners involved: the *Barcelona Supercomputing Center – Centro Nacional de Supercomputación* (BSC-CNS), the *Universitat Politècnica de Catalunya* (UPC) and the *Kungliga Tekniska Högskolan* (KTH). The first institution works on the computer architecture, the second one works on the communication part and the graphene antenna and the third one works on the graphene as a material. This project represents a breakthrough respect to the actual technologies and procedures. A great factor is the verticality that presents because involve researchers at all the layers (architecture, communications and materials) and that is awesome due to we can work taking care of the necessities of the others parts and at the same time give our requests to the other members. It is like a retro-alimented system in order to improve the efficiency and the performance of all the parts individually and collectively.

To continue with the way opened for this final project career, I am planning to keep working on that topic with the new possibilities planted in this document. These possibilities are a more accurate study of the other bottlenecks that presents the design of the many-core systems, the development of analytical models to characterize the procedures of the processors, etc. All this work would be a Master Thesis. And the next step, which can be taken into account as a Ph. D., is the development of the novel architectures that really exploits the characteristics of the new ways of communication that the technology is enabling to improve the performance of the CMP with more than 1000-cores. Also, it is important to note that if these processors are application dependent the final performance can be better than a generic one. This also can include more Ph. D. students in the field of the algorithms that exploits the new characteristics to obtain a good mapping. With this combination, the hardware and the software, we will get optimal solutions for the next generation of multicore/multithreaded processors.

As a first step of this research plan, a paper will be submitted to the 1st International Workshop on Emerging Topics in NoC-aware Computer Architecture (ETNA) that it will be held in the 40th International Symposium on Computer Architecture (ISCA 2013, Tel Aviv, Israel). This paper, if it is accepted, is going to present the position and the future vision for the computer architectures enabled this new technology (the graphennas), which is much related with the first study realized in this final project career.

8. – References

[1] John L. Hennessy, David A. Patterson; *Computer Architecture: A quantitative approach 4th edition.* San Francisco: Morgan Kaufman Publishers, 2007.

[2] Natalie Enrigth Jerger, Li-Shiuan Peh; *On Chip Networks ;* Synthesis lectures on Computer Architecture #8. Morgan & Claypool Publishers

[3] Bruce Jacob; *The Memory System;* Synthesis lectures on Computer Architecture #7. Morgan & Claypool Publishers

[4] Tim Harris, James R. Larus, Ravi Rajwar; *Transactional Memory 2nd edition;* Synthesis lectures on Computer Architecture #11 Morgan & Claypool Publishers

[5] D. Dice, O. Shalev, and N. Shavit; *Transactional Locking II*. In Proceedings of the 20th International Symposium on Distributed Computing (DISC), 2006

[6] A. Dragovic, R. Guerraoui, and M. Kapalka. *Stretching Transactional Memory*. In Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), 2009

[7] John L. Hennessy, David A. Patterson; *Computer Organization and Design, The hardware/software interface.*

[8] David Culler, Jaswinder Pal Singh, Anoop Gupta; *Parallel Computer Architecture.*

[9] Binkert, N., Beckmann, B., Black, G., et al. *The gem5 simulator*. Computer Architecture News, 2011.

http://www.m5sim.org/Main_Page

[10] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. *EigenBench: A Simple Exploration Tool for Orthogonal TM Characteristics*. In IISWC '10: Proceedings of The IEEE International Symposium on Workload Characterization, December 2010

http://ppl.stanford.edu/eigenbench/

[11] Sergi Abadal, Albert Cabellos-Aparicio, Jose A. Lazaro, Eduard Alarcón, Josep Solé-Pareta; *Graphene wireless hybrid architectures for multiprocessors, bridging nanophotonics and nanoscale wireless communication.* In Proceedings of the 14th International Conference on Transparent Optical Networks (ICTON), 2012, pp. 1–4.

[12] Sergi Abadal, Albert Cabellos-Aparicio, Eduard Alarcón, Max C. Lemme, Mario Nemirovsky, Ian F. Akyildiz; Graphene-enabled Wireless Communication for Massive Multicore Architectures. Accepted for Publication in IEEE Communications Magazine, 2012.

[13] George Nychis, Chris Fallin, Thomas Moscibroda, Onur Mutlu, Srinivasan Seshan; *On-Chip Networks from a Networking Perspective, Congestion and Scalability in Many-Core Interconnects.* In SIGCOMM'12

[14] Stavros Volos, Ciprian Seiculescu, Boris Grot, Naser Khosr Pour, Babak Falsafi, Giovanni De Micheli; *CCNoC: Specializing On-Chip Interconnects for Energy Efficiency in Cache-Coherent Servers.* In Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS 2012)

[15] Ran Manevich, Isask'har Walter, Israel Cidon, and Avinoam Kolodny; *Best of Both Worlds: A Bus enhanced NoC (BENoC).* In 2010 IEEE 26-th Convention of Electrical and Electronics Engineers in Israel.

[16] Alberto Ros, Stefanos Kaxiras; *Complexity-Effective Multicore Coherence*

[17] Jaydeep Marathe, Frank Mueller, Bronis de Supinski; *A Hybrid Hardware/Software Approach to Eficiently Determine Cache Coherent Bottleneck.* In ICS'05, June 20-22, Boston, MA, USA.

[18] Alvin R. Lebeck and David A. Wood; *Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors.* In ISCA '95, Santa Margherita Ligure Italy

[19] Ismail Kadayif, Mahmut Kandemir; *Modelling and Improving Data Cache Reliability.* In *SIGMETRICS'07,* June 12–16, 2007, San Diego, California, USA.

[20] Joonwon Lee, Umakishore Ramachandran; *Synchronization with Multiprocessor Cache.*

[21] Partha Pratim Pande, Cristian Grecu, Michael Jones, André Ivanov, Resve Saleh; *Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnect Architectures.* In IEEE TRANSACTIONS ON COMPUTERS, VOL. 54, NO. 8, AUGUST 2005

[22] Umit Y. Ogras, Jingcao Hu, Radu Marculescu; *Key Research Problems in NoC Design: A Holistic Perspective.* In *CODES+ISSS'05,* Sept. 19-21, 2005, Jersey City, New Jersey, USA

[23] Luca Benini, Giovanni De Micheli; *Networks-on-Chips: A New SoC Paradigm.* In 2002 IEEE Transaction.

[24] A.K. Geim and K.S. Novoselov; *The rise of Graphene*

Index of tables

Index of figures

Acronyms

CMP	Chip Multiprocessor
MPI	Message Passing Interface
GWNoC	Graphene Wireless Network on Chip
MESI	Modified Exclusive Shared Invalidated
NoC	Network on Chip
SoC	System on Chip
ISA	Instruction Set Architecture
MIPS	Microprocessor without Interlocked Pipeline Stages
CPU	Central Processing Unit
TM	Transactional Memory
TL2	Transactional Locking 2
STM	Software Transactional Memory
ICT	Information and Communication Technologies
RF	Radio Frequency
API	Application Programming Interface

Appendix I - Gantt diagram