UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

On the Impact of Heterogeneous NoC Bandwidth Allocation in the WCET of Applications

Author:
Jordi CARDONA NADAL

Supervisors:
Dr. Carles HERNANDEZ LUZ
Dr. Jaume ABELLA FERRER
Dr. Francisco J CAZORLA
ALMEDIA

A thesis submitted in fulfillment of the requirements for the degree of Master in Innovation and Research

in the

Facultat d'Informàtica de Barcelona (FIB)
Departament d'Arquitectura de Computadors (DAC)

and the

Computer Arquitecture and Operating Systems (CAOS)
Barcelona Supercomputing Center (BSC-CNS)

April 16, 2018

Declaration of Authorship

I, Jordi CARDONA NADAL, declare that this master thesis titled, "On the Impact of Heterogeneous NoC Bandwidth Allocation in the WCET of Applications" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master degree at Universitat Politècnica de Catalunya .
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Universitat Politècnica de Catalunya or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

_____

Date:

_____

Abstract

Achieving high performance and time predictability in current manycore systems is still a challenge these days as the number of cores in manycore systems keeps increasing every day. Even though the memory speed has improved in the last decades, the speed gap between cores and memory is still relevant since part of the time that cores are stalled is due to waiting for data coming from memory. This performance bottleneck gets even worse when we increase the core count as we have more and more cores competing for the same memory bandwidth and for the same interconnection network to reach memory.

In order to mitigate this issue, in the last few years Networks-on-Chip (NoCs) such as meshes and trees, have been introduced in high-performance manycore processors due to their physical scalability and low cost. In general, these NoCs lead to heterogeneous latencies across cores to reach memory due to core location, which determines the distance between the core and the memory accessed (i.e. number of routers in between), the arbitration policy and the routing algorithm used, and the contention caused by other cores in the NoC. Furthermore, in the context of parallel applications running simultaneously in multiple cores, performance is determined by the slowest thread, which may change across different thread-to-core allocations.

In the context of Critical Real-Time Embedded Systems (CRTES), the use of manycores deploying wormhole NoCs complicates the analysis of application's timing behavior. In particular, in order to assess whether applications will run within their allocated time budget, we need to estimate their Worst-Case Execution Time (WCET) which, in turn, depends on the Worst Contention Delay (WCD) that each packet can experience to reach memory. In this thesis, we study the influence of core allocation on WCD and WCET for tree and mesh NoCs, and how specific modifications of the arbitration algorithm allow reducing the WCET by homogeneizing the memory latency across cores.

Acknowledgements

First of all I would like to deeply thank my master thesis advisors Carles Hernández, Jaume Abella and Francisco J Cazorla for their guidance, mentoring and for giving me the opportunity to undertake this master thesis in the Computer Architecture and Operating Systems' group (CAOS) at Barcelona Supercomputing Center (BSC-CNS). I also want to thank the rest of the members of the CAOS group as they have always been available to provide feedback on my work and give me advices to improve it.

Furthermore, I would like to acknowledge the BSC institution for financially support my master studies and these two others institutions as this work has been partially funded by the Spanish Ministry of Science and Innovation under grant TIN2015-65316-P and the HiPEAC Network of Excellence.

Finally, I would not forget to thank my entire family and specially my parents, my sister and my closest friends for their unconditional support during all these years, without you this master thesis would not be possible. My most heartfelt thanks for the great confidence you have shown in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computing systems require functional correctness, so that the outputs provided correspond to the system specification. A subset of the computing systems, known as real-time systems, also needs *timing correctness*. Timing correctness refers to the execution of the corresponding functionalities before specific deadlines. For instance, the braking system of a car needs to stop the car within limited time bounds. Analogously, video players need to process frames at a given speed.

Some real-time systems may afford missing some deadlines with certain frequency, since those *timing failures* only produce a lower quality output, which may not even be perceived if the deadline miss rate is sufficiently low. For instance, this is the case of many systems related to entertainment (e.g. music and video players). However, some other systems are intended not to miss any deadline, since a failure of those systems could lead to catastrophic consequences (e.g. the navigation system of a plane or the braking system of a car). These systems, often referred to as critical real-time embedded systems (CRTES), require a careful functional and timing verification to prove – qualitatively and quantitatively – that the risk of failure can be regarded as residual. In other words, the validation and verification (V&V) process provides evidence that all relevant scenarios have been considered and safety measures have been put in place to mitigate risks.

CRTES can be critical because of many reasons. For instance, safety-critical systems are those whose failure could cause casualties, injuries or severe damages to objects (including the system itself). Instead, mission-critical systems are those whose

failure may typically cause economical losses such as, for instance, systems controlling measurement instruments in a satellite. Even if those systems do not compromise the integrity of the satellite itself, they may lead to a failure of accomplishing the mission, which ultimately is a severe consequence. In this work, we target the design and timing verification of CRTES, regardless of their type of criticality.

Until recently, CRTES built upon relatively-simple software running on relatively low-performance (and low-complexity) hardware. For instance, many avionics systems still today build upon single-core processors with in-order execution and without cache memories. The advantage of those systems is that timing verification is relatively simple, since execution time variability is low. Hence, the Worst-Case Execution Time (WCET) can be estimated with affordable costs either by using timing models of the system or by collecting measurements and adding a safety margin over the highest execution time observed.

However, the increasing automation of systems first, and the trend towards fully-autonomous systems later, pushes CRTES industry for adopting hardware platforms delivering much higher performance to respond to the performance demands of complex functionalities. Multicore and manycore processors are one such type of hardware platform. They consist of a number of cores capable of executing software simultaneously, as well as an interconnection network to communicate cores among them and with neighbor devices (e.g. main memory).

1.1 Motivation

Multicore and manycore processors have already been considered for the execution of critical real-time software in experimental environments [24, 34, 39, 17, 3, 13] – although they have not been deployed yet. While communication buses have been proven effective for small multicores [33], they have been proven not to scale well to larger multicores and manycores (e.g. $\geq$ 8 cores) [37]. Hence, more complex Networks-on-Chip (NoCs) are used for the interconnection of the cores, as well as to reach any other device.

Different types of NoCs have been considered to satisfy the communication requirements of multi/manycores. We classify those works into two different categories: ad-hoc and commercial designs. The former category consists of those NoCs that are particularly designed to provide time predictability, so that the WCET of tasks can be easily estimated. In general, these designs trade average performance for guaranteed performance (WCET), which is against efficiency. Therefore, chip manufacturers are unlikely to adopt them. The latter category of NoC designs corresponds to Commercial Off-The-Shelf (COTS) designs [25, 9, 40]. In other words, it corresponds to those designs that can already be found in commercial processors (e.g. meshes, torus, etcetera). However, those designs often provide very high average performance at the expense of offering poor timing guarantees. Hence, they are not suitable, in general, for CRTES. Recently, this has been addressed and small modifications have been proposed on COTS NoC designs to make them amenable to (tight) WCET estimation while preserving high average performance [24].

Those multi/manycore processors equipped with powerful NoCs have been proven efficient for the execution of independent tasks in the different cores, or by running some simple deployments of parallel applications [24]. Also, it has been shown that almost-homogeneous bandwidth can be allocated to the different cores with appropriate hardware support on COTS NoCs, thus making core allocation irrelevant for WCET estimation, since all cores experience similar worst-case communication delays [22]. These NoCs are referred to as weighted NoCs. By using locally-unfair arbitration, those NoCs achieve globally-fair bandwidth allocation across cores. However, the benefits of weighted NoCs (and in particular meshes) for different degrees of traffic and application types have not been assessed. Also, the performance of NoCs for different types of parallel applications has not been assessed.

1.2 Contribution

This thesis studies whether it is possible (and to what extent) configuring powerful NoCs in multi/manycores to reduce the WCET of critical real-time parallel applications.

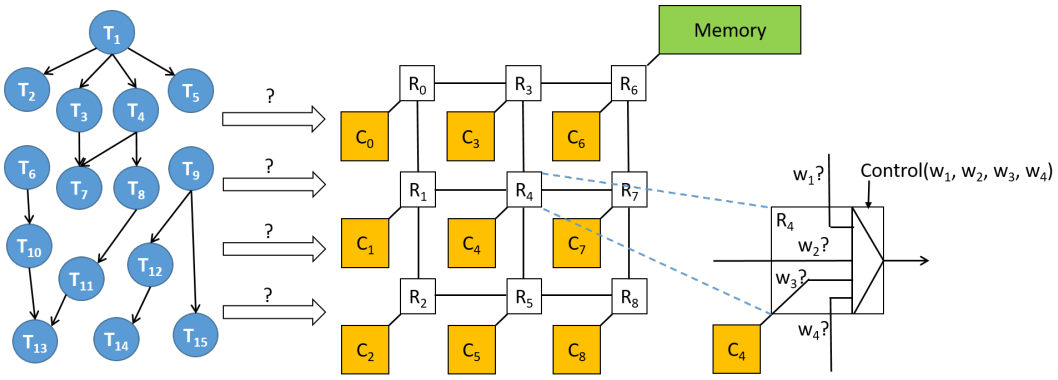


FIGURE 1.1: Schematic of the problem where this thesis contributes.

The magnitude of the problem is illustrated in Figure 1.1. As shown, mapping parallel applications to a multi/manycore where bandwidth per link can be adjusted is a very complex problem since both, thread mapping and weight allocated to each link, are not independent. In this thesis we aim at investigating the tradeoffs involved. Note that solving this problem is a complex research challenge much beyond the scope of this thesis. Instead, this thesis focuses on characterizing the problem and understand the tradeoffs involved.

In particular, the contributions of this work are as follows:

- We verify and adapt a NoC simulator to provide WCET bounds for requests.
- We implement a weighted mesh on that simulator where the bandwidth allocated to each core and flow can be flexibly adjusted. In this process, we define a number of parameters left open in the original description of the mechanism implemented.
- We assess quantitatively the potential gains that thread-to-core allocation and weight allocation can obtain by running experiments on a processor performance simulator where the NoC simulator is integrated. For the sake of this analysis, ad-hoc benchmarks have been developed, thus providing us with controllability on the experiments and a-priori knowledge of the behavior of applications. This is key for the verification of the design.

1.3 Structure of the Thesis

The rest of this document is organized as follows. Chapter 2 provides background on timing analysis for CRTES, NoC design and application imbalance. Chapter 3 provides some related work on NoC designs for CRTES. Chapter 4 introduces the evaluation framework used in this work. Chapter 5 presents weighted meshes including their implementation, WCD modelling and characterization. Chapter 6 provides the result of our analysis for parallel applications. Finally, Chapter 7 draws some conclusions and presents some future work.

Chapter 2

Background

2.1 Timing Analysis

Critical real-time tasks must complete their execution by a given deadline. In order to assess whether this will be the case during operation, a process called *timing analysis* is performed as part of the verification of the system to estimate the WCET of those tasks. This step is mandatory to schedule tasks such that they can complete their execution before a given deadline.

Two main timing analysis strands have been pursued in industry and academia: static timing analysis (STA) and measurement-based timing analysis (MBTA) [42].

STA builds a timing model of the hardware and on which it performs an abstract interpretation of the program under analysis, modelling the potential hardware states that can occur. Ideally, STA models all possible state transitions, thus obtaining each potential state of the hardware at each step (i.e. after each instruction or after each execution cycle). Finally, when all instructions of the program have been analyzed, the final state with a higher execution time determines the WCET. However, the number of potential states exploits exponentially due to potential outcomes of branches, uncertainty on the addresses accessed (and so on the hit/miss outcome of cache memories and the resulting cache state), and limitations of the timing model. The way STA addresses this is by reducing the number of states modelled making pessimistic assumptions. For instance, cache accesses that could hit or miss, may be assumed to miss, or some data that could reside in cache at some point, is assumed not to be in cache to allow merging several states. Overall, STA is

highly demanding on the amount of information needed to derive tight WCET estimates and it has been shown only suitable for simple programs running on simple hardware [4].

MBTA, instead, builds upon execution time measurements of the program collected on top of the target platform. This removes the need for a timing model and for any type of abstract interpretation. However, the challenge for MBTA resides in relating the scenarios evaluated during the test campaign with those that can occur during operation. Moreover, a number of features such as memory placement of objects (and so cache placement), contention in shared resources and values operated in variable-latency units are, often, beyond the control of the end user. Hence, the limited controllability together with the difficulties to assess the coverage of the analysis tests with respect to the scenarios during operation pose uncertainty on the WCET estimates obtained with MBTA. In fact, MBTA usually uses the maximum observed execution time (MOET) plus a safety margin (e.g. 20% in single-core processors [41]) as the WCET estimate. However, the confidence provided by the safety margin is unknown. Hence, MBTA is convenient when users are highly familiar with the software analyzed and the hardware platform, so that they can create relevant test cases and set appropriate safety margins [4].

Some hybrid timing analysis approaches exist that attempt to combine the advantages of both, STA and MBTA, by, for instance, collecting measurements at finer granularity than end-to-end program runs and applying some form of abstract analysis to estimate the execution time for paths that have not been measured [4]. While conceptually those methods may bring tighter estimates than STA and less uncertainty than MBTA, their tightness and confidence cannot be proven better than those of STA and MBTA.

Overall, different timing analysis families bring their pros and cons. In general, each approach is suitable for some specific systems, but all of them have been used even for systems with the highest criticality. In general, MBTA is the most used approach since it does not provide WCET estimates duly pessimistic, and its application is less cumbersome than that of STA. In this thesis, we use specific models to estimate the worst latency needed to traverse the NoC, as such latency can be obtained analytically, and MBTA to measure the execution time of the program and

to estimate the WCET. In particular, the WCET is obtained by measuring execution time in a contention-free scenario in the NoC, and then adding the Worst Contention Delay (WCD) to each memory request to account for the potential contention that packets could experience upon integration of the application with other software that, potentially, could cause such worst-case contention. This methodology introduces time composabilty as when we integrate for instance a new application in our system, we do not need to reverify the timing correctness of all the applications that are running but the new one. Time composability property is a highly sought property as it allows industry to optimize the V&V process (e.i. it can be done in an incremental manner) reducing the cost of the timing V&V part.

2.2 NoCs

The need for multicores and the fact that cores may need to communicate between them and with other devices (e.g. main memory, shared caches) imposes the need of setting up an interconnect network among cores and other devices. On small multicores, classic monolithic solutions such as buses have been shown to be effective [33]. However, as the number of cores in the processor grows, buses become easily a bottleneck. The main reason is that their latency increases due to their increased capacity and long distance, which can only be partially mitigated pipelining the bus. However, in general, the bus cannot serve more than one transfer simultaneously due to its monolithic nature. Hence, given that its latency grows with the number of cores, the relative bus occupancy per core increases, and the number of cores willing to use it also increases, thus making the bus being a potential bottleneck even with a few cores (e.g. > 4 cores).

To tackle this problem, NoCs have been proposed as the most convenient interconnection solution to connect cores and other devices in multicores [10]. NoCs rely on setting up point-to-point connections inside the chip to keep all components connected by means of switches and links. Hence, in a NoC one may not be able to reach each device directly, but messages can be routed through those links and switches to destination. Given that NoCs build upon many individual connections, they can afford transmitting multiple messages simultaneously as long as they can be managed

not to clash in the same resources. For instance, routing messages appropriately may avoid conflicts or, alternatively, setting up buffers may allow stalling messages when the resource required is busy.

The most common NoC topologies used in multicores include rings [31], trees [37] and meshes [33, 10]. For instance, some commercial processors build upon those networks. The Intel Nehalem [25] implements a bidirectional ring network. The Kalray MPPA-256 implements a tree network [6, 9]. Finally, several processors, such as the Tilera 100-core chip [40], the Polaris prototype [16] and the Single-chip Cloud Computing (SCC) prototype [30], implement a 2D mesh network.

2.2.1 Data organization and transmission

When two or more networked devices try to read from each other's memory, the unit of information sent or received is called *message*. The device that sends the petition, first has to compound this message and send it in the form of a *request*. This request contains inter alia, the address where the receiver will found the requested data. Then, after processing the request, the receiver will be able to send back a *reply* message containing the data also known as *payload*. Alternatively, a networked device may send a message containing data to a receiver, thus containing already the payload. The receiver, upon data reception may or may not send a reply message (without payload) to acknowledge the reception of the data, depending on the actual data management policy of the network implemented.

Usually, *network interfaces* together with some *direct memory access (DMA)* engines and link drivers are in charge of composing and processing messages before sending and receiving information. In some networks, there is a fixed amount of information that can be transferred so that network buffers can be sized appropriately. Messages longer than the maximum transfer unit are divided into smaller units, called *packets*. These packets are reassembled into messages at the destination end node before delivery to the application.

Packets normally contain, in addition to the payload, a header and a tail part. In Figure 2.1 it can be seen a representation of a possible packet format. Usually, the packet header contains the destination port, the message ID of the packet it belongs

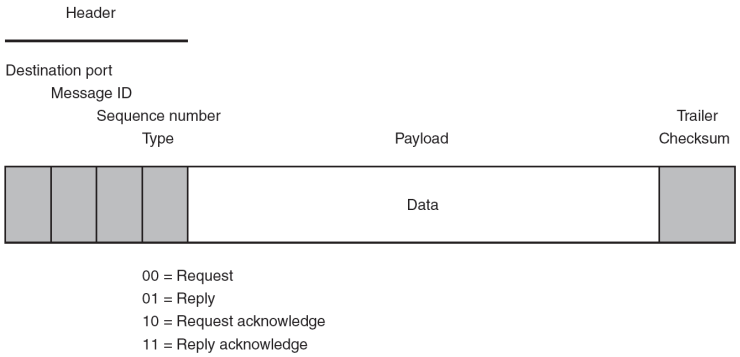


FIGURE 2.1: Packet format

to, a packet sequence number to enable reordering of the entire message in the destination device and packet type (request, reply, acknowledge,...). Each packet also uses to contain a checksum field that is used to check if the packet received has no corruption (re-calculating the checksum value in the receiver device and comparing it with the checksum field received in the tail part of the packet).

Packets may have sizes larger than the transmission bandwidth (links). For instance, messages may be divided into packets of up to 512 bits, and links to communicate networked interfaces may be of up to 128 bits. Hence, messages cannot be transmitted at once and may need to be divided into smaller network units called *flow control units (FLITS)*. FLITS, are the smallest data entity inside networks and they can be transmitted atomically. The set of FLITS forming a message is known as *flow*. All FLITS of a flow are transmitted atomically across several cycles between two nodes, so only the first flit needs to include information about the destination, whereas all FLITS contain a flow ID to determine when a packet has been completely received and hence, a different packet can be sent.

Once a packet is ready to be sent at its source, it is injected into the network by the network interface. The speed of the packet transmission will depend strongly on the media, distance and the form factor used. In order to ensure reliable delivery of packets, two main assumptions have to be taken into account: (1) The sender cannot send packets at a faster rate than they can be processed by the receiver, and (2) packets are correctly received (neither lost nor corrupted in transit). The most used strategy is called *flow control*, where the receiver notifies the sender to stop sending packets until the receiver has enough space in its input buffer or an emptiness threshold has been reached. The basic implementation consists of using a simple

handshaking protocol between the sender and the receiver. Two main different strategies exist to implement flow control:

- *Stop & Go*: The receiver notifies the sender to stop or to resume sending packets once high or low buffer occupancy levels are reached respectively.
- *Credit-based*: Every time a packet is transmitted, the sender decrements the credit counter. When the receiver processes it, it increments the sender's credit counter. The sender can send a new packet as long as its credit is above a given threshold.

The most spread policy is the Stop&Go one as it introduces lower traffic in the network (flow control messages are only sent when buffer capacity bounds are reached) than the Credit-based strategy (flow control messages are sent every time that a packet is processed by the receiver).

2.2.2 Network structure

A NoC consists of a number of physical components, including switches, links, network interfaces, etc. Network interfaces connect end nodes to switches (also referred to as routers). Then, routers are connected among them by means of links. Packet's flow is determined by routers, which implement a number of policies for switching, routing and arbitration in general. Instead, links are passive components, and network interfaces inject/eject packets with specific source and destination, but without influencing when and how those packets will traverse the NoC.

Figure 2.2 depicts a typical pipelined router. First, a set of input buffers for each port are used to store incoming packets. Second, a routing algorithm determines the next router where those packets need to be sent (or the network interface at destination). Third, an arbiter determines which input ports buffers grants access to each output port. Finally, a switch allocator matches the corresponding input port buffers with the output ports ones for the packets' transfer. Eventually, whenever input buffers are available at the destination router, packets leave the current router and are stored in the input buffers of the next router.

Input buffers for a given input port can be organized into multiple queues (virtual channels) to allow multiple flows being processed in parallel. This requires

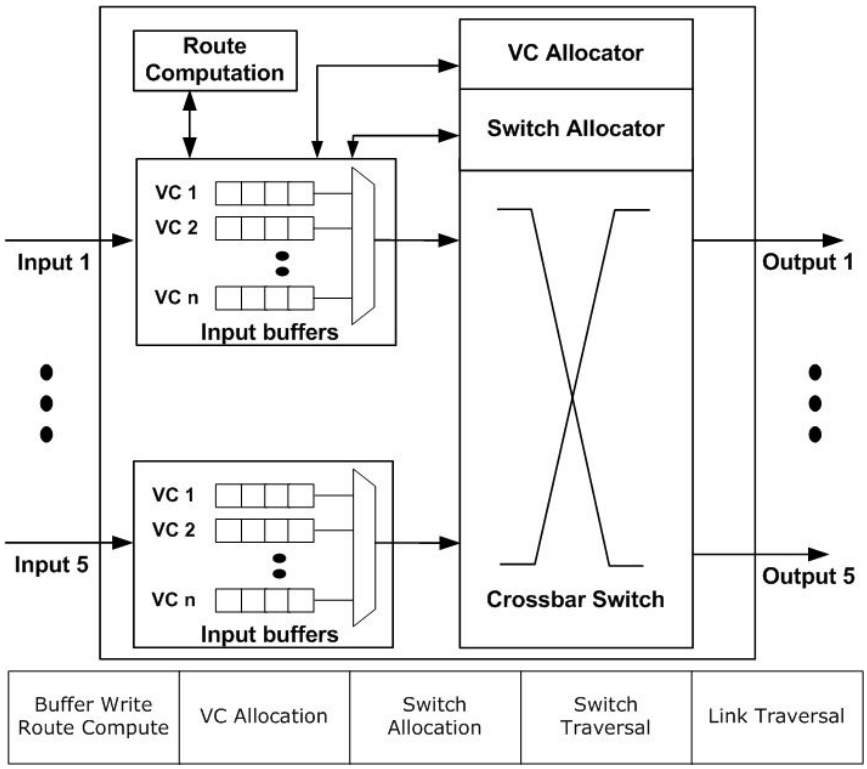


FIGURE 2.2: Pipelined router

additional queues per input port and a virtual channel allocator. On the other hand, however, virtual channels typically increase the average throughput of routers.

In many NoCs, multiple routes exist to forward a packet from its source to its destination. Different routing policies have been proposed to accomplish this goal, with some common goals, namely: (1) trying to minimize the number of hops from source to destination, (2) avoiding deadlocks and (3) avoiding livelocks. In particular, minimizing the number of hops is desired for efficiency reasons, and only some policies consider alternative routes to go around broken links or overly congested NoC regions. Deadlocks in NoCs occur whenever a set of packets use multiple resources (e.g. buffers) such that no packet can make forward progress because the resources needed are busy, and no resource can be released because packets cannot make any forward progress. Finally, livelocks occur when routing decisions may potentially make a packet not to reach its destination despite it may keep moving in the NoC (e.g. looping in a subset of routers). Routing policies implementing those goals may be dynamic or static, either taking routing decisions based on dynamic conditions or having predetermined routes. In the case of critical real-time tasks, static policies are normally used since they allow guaranteeing efficient routes, as well as

deadlock and livelock avoidance. Among those, the most popular routing policy is XY, where packets are forwarded in the X direction until they reach the Y-coordinate of their destination, and then they are forwarded in the Y direction until the destination node. Once a packet is forwarded in the Y direction, it cannot be further routed in the X direction. Such policy guarantees, by construction, minimal distance routes as well as deadlock/livelock avoidance. Moreover, it allows deriving tight bounds to the worst delay to forward a packet from any node to its destination. In critical real-time tasks we can also use other static routing policies that also belong to the dimension-order routing where it belongs XY routing (e.g. YX (2D) or XYZ and e-cube in 3D NoCs).

Switching algorithms determine how a packet is transferred from source to destination. The two most popular switching policies are *cut-through* and *wormhole*.
The former does not allow a packet make any forward progress until enough resources for the full packet cannot be allocated in the following router. Wormhole, instead, allows individual FLITS moving forward, but once a flit has been sent, only subsequent FLITS of the same packet are accepted in the destination buffer, thus avoiding the interleaving of packets in a single buffer.
Wormhole NoCs are regarded as more efficient than cut-through ones since they allow some forward progress even if resources at the destination router cannot store the full packet, and also allows arbitrarily long packets, as opposed to cut-through, which may need large buffers if long packets may be communicated. Hence, in the rest of this work we rely on wormhole switching, which has been proven to be the preferred choice for CRTES [23].

2.2.3 NoC concepts for CRTES

In the context of CRTES, the most critical parameter for timing analysis (and so for WCET estimation) is the amount of time a request can be delayed in the NoC due to contention. In particular, the delay a request (packet) takes to traverse the NoC consists of the minimum intrinsic delay to reach the destination in a congestion-free scenario, also known as *zero load latency* (*zll* for short), plus the delay due to contention. Since contention during operation cannot be forecasted tightly until late

design stages, but WCET estimates are needed much earlier in the design process, worst-case assumptions need to be made in terms of congestion.

Two main approaches have been devised to account for such worst-case congestion: Worst-Case Traversal Time (WCTT) and Worst Contention Delay (WCD).
The former [28, 19, 27, 8] accounts for the worst individual delay each request may suffer. However, adding such delay to each request for WCET estimation has been shown to be overly pessimistic because a large fraction of such delay can be overlapped for several requests being processed in parallel [23]. Hence, some authors propose using WCD instead, which only accounts for the additional delay caused by another request, discounting the delay that has already been accounted to other requests. This delivers reliable but much tighter WCET estimates.

Since worst-case assumptions need to be made to account for worst-case contention delay, it has been shown that several choices, despite not being the most efficient ones for average performance, deliver the lowest WCET estimates. Amongst those we build on the following ones [23]:

- **Packetization**. Since a request may have to wait for requests in the other ports to be processed first, and those requests (packets) could be arbitrarily long, packetization has been proposed, where all packets are made to have a single flit. Hence, each flit of the task under analysis has to compete with single flit packets from other flows. Otherwise, each packet, regardless its length (e.g. 4 FLITS), would have to compete against maximum-size packets (e.g. 16 FLITS) from other contenders, whose relative contention delay can be much higher (e.g. 4x that of single-flit packets).

- **Single Virtual Channel**. The existence of multiple virtual channels, while effective in the average case, has a multiplicative effect on the number of contenders that packets of the task under analysis will have to compete with in every router. Hence, given N virtual channels, worst-case contention grows by a factor of N w.r.t. a single virtual channel. Therefore, the best choice for CRTES is using NoCs with a single virtual channel.

Finally, performance guarantees have been shown to vary drastically across cores in meshes (one of the most commonly used NoC architectures) due to the varying bandwidth effectively allocated to each core and diverse latencies caused by non-homogeneous distances from cores to their target node (e.g. the one where main memory is attached) [22]. Heterogeneous bandwidth and latency is, in general, unwanted due to the fact that tasks running in some cores – those with lower bandwidth and higher latencies – can be severely penalized. This challenge has been addressed with the use of weighted meshes, where heterogeneous bandwidth allocation across links is given in the routers so that overall bandwidth can be homogenized across cores and, to some extent, performance across cores is homogenized [22]. Later in this thesis, we provide details on weighted meshes, as they are a key NoC design studied in the context of parallel applications as part of our work.

2.3 Parallel Applications

Most research on NoCs for CRTES has considered single-threaded applications, since they are current practice in the domain. However, parallel applications are needed for computing intensive tasks such as those related to autonomous driving and unmanned navigation. Hence, NoC design and analysis cannot focus on the behavior of each core in isolation only, but it also needs to account for the implications on parallel applications.

Existing work has only considered default NoCs devised for single-threaded applications with just one exception: the parMERASA architecture [26, 24]. In the context of parMERASA (an already finished FP7 programme project), investigation was done on how to map parallel applications to cores for an efficient use of resources building on some NoC designs including meshes. However, bandwidth and latency heterogeneity was not tackled and thread allocation was applied on those default architectures, which have been shown to offer highly unbalanced bandwidth and latency across cores.

High imbalance across cores may lead to scenarios where, despite smart thread-to-core mapping algorithms are used, threads experience highly diverse execution

times. In the context of single-threaded applications this can be mitigated by allocating additional tasks to the fastest cores. However, in the context of parallel applications, the slowest thread determines the full application execution time. Therefore, high imbalance penalizes performance (and consequently WCET) severely.

Therefore, weighted meshes offer a great opportunity to optimize weight allocation and thread-to-core mapping, either independently or coordinately, to optimize the performance of parallel applications in CRTES.

Chapter 3

Related Work

While there have been several proposals for real-time aware NoC designs, exploring to which extent high-performance (COTS) NoC designs can be used in the real-time domain is of paramount importance:
On the one hand, it is well accepted that the CRTES domain is a relatively small market in comparison with other domains such as mobile. Hence, customized NoCs specifically designed for real-time systems (e.g. time-triggered ones and those based on time division and multiplexed access (TDMA)), which may require high non-recurrent costs, are unlikely to be adopted in the context of industrial CRTES [39].
On the other hand, the big majority of the proposed manycore designs across all computing domains use high-performance wormhole NoCs (wNoCs) to perform the interconnection of cores and shared resources within the chip. This makes wNoCs accessible (at low cost) by the CRTES since they are implemented in a vast set of chips. In this paper we have focused on improving the performance guarantees achieved by wNoCs in terms of bandwidth and latency.

Several real-time specific NoCs have been proposed based on TDMA such as [34] and [12]. While TDMA-based NoCs deal with contention at transaction level (e.g. read and write memory operations), time-triggered architectures [21] increase the abstraction level by introducing a self-contained computational unit. In time-triggered architectures micro-components exchange messages in contention-free slots. However, event-triggered transactions, such as cache misses that access main memory through the NoC, may suffer contention delay which must be upper bounded. We refer to NoC designs with real-time guarantees and time-composable behavior as Guaranteed Service (GS) NoCs. Nostrum [20] and Aethereal [12] NoCs provide

GS using time-division multiplexing, and hence, time composable bounds.

Many studies have also been carried out with the purpose of providing realistic and feasible latency bounds for best-effort wNoC. Using prioritization on a per-virtual channel basis has proven being an effective means to achieve tight latency bounds in wNoCs [35]. However, the use of per-virtual channel prioritization becomes impractical when a significant number of flows exist in the network. To overcome this issue the impact of virtual channel sharing has been analyzed in [36] and [29]. However, while these approaches effectively reduce the number of virtual channels required, the timing guarantees obtained build upon a detailed knowledge of the characteristics of the software (applications and/or tasks) that will execute in the deployed system and hence, do not meet incremental qualification requirements. The work in [18] has similar pros and cons, since the proposed solution guarantees specific bandwidth allocation for GS connections per port, by splitting the bandwidth of output ports among best effort and guaranteed service connections.

Authors in [19] made one of the first studies that provided reliable contention bounds for wNoCs without building upon flit-level virtual channel preemption. Later, this analysis has been improved in [28] where tighter bounds are presented. The model in [28], as those mentioned above, also requires detailed information on all communication flows that will be finally deployed in the system to estimate reliable upperbounds. In other words, latency bounds provided in [28] are not time-composable. Some recent works that build upon wNoCs propose interference-free NoC designs [5, 14]. The solution in [5] has been proven to cause lower degradation on best-effort traffic than the one in [14]. The former achieves its goal by using specific ways to multiplex virtual channels. However, despite its improved performance, the performance degradation caused on best-effort traffic is still large.

Recently, as explained before, authors in [22] have proposed an alternative approach to meet CRTES requirements. In particular, that work proposes specific ways to derive time-composable worst contention delay bounds without sacrificing average performance and by allocating weights to arbiters so that fair bandwidth allocation is achieved across cores. In our work we analyze the use of wNoCs for parallel applications and the impact of using weighted meshes in that context.

Chapter 4

Evaluation Framework

In this section we describe the details of the evaluation framework we have used in this thesis. In particular, we provide details about both the simulation platform and the type of benchmarks/workloads we have employed.

4.1 Simulation Platforms

Our simulation platform is based on an enhanced version of the SoClib simulator [38] that has been developed at the CAOS group of the Barcelona Supercomputing Center. This simulator platform models with high detail (cycle accurate) the pipeline of the processors and the cache hierarchy. This simulator has been validated against real boards showing that performance deviations of this simulation platform are below 3% when modeling the NGMP multicore processor [7].

4.1.1 Processor details

We use Soclib to model a multicore/manycore processor. Cores used the PowerPC architecture [15] since this architecture is one of the most interesting ones for avionics platforms. Cores employed in these architecture are simple to ease its timing analyzable. In particular, we use in-order cores with 5-stage pipeline, single issue, and floating-point support. Cores also comprise separate data and instruction L1 caches. Data caches employ write-back write policy to reduce the amount of requests to the shared resources. The exact details of the core are given in Table 4.1.

	Core Features
Pipeline	32 bit Sparc PowerPC In-order, single-issue 5-stage pipeline FPU
Cache	L1private 4-way 16KB Instruction 4-way 16KB Data LRU replacement Modulo placement

TABLE 4.1: Processor Configuration

4.1.2 Network-on-chip Simulator

To model the behavior of NoCs we have attached the gNoCsim simulator to the SoClib platform [2]. The gNoCsim simulator is a cycle accurate simulator of wormhole networks developed by Universitat Politècnica de València under the scope of the FP7 NanoC project. The gNoCsim simulator supports several topologies like tree, mesh, and torus and several different deterministic routing algorithms like XY, dimension order routing, and logic-based distributed routing. In this thesis, we only use the mesh with XY routing and the tree configurations. In a similar way, even gNoCsim implements different flow control algorithms, in this thesis we only use Stop&Go already explained in the Background section. Meshes in gNoCsim can be configured to support different router architectures while trees do only support the router architecture presented in [37]. In this thesis, for meshes we have chosen the canonical router architecture while we have used the default router architecture provided for the trees.

The gNoCsim simulator can work in both master and slave modes. In slave mode gNoCsim simulator simulates the requests produced by the SoClib simulator that go through the network. Typically, these are the core to memory petitions and the corresponding memory responses. However, gNoCsim simulator can also work in master mode using different synthetic traffic patterns to characterize the network behavior under different stressing situations. In this thesis, we have used both simulation modes provided by the gNoCsim simulator.

4.2 Workload

4.2.1 Synthetic Traffic

In order to analyze the behavior of the different network configurations analyzed, namely trees and meshes, using round-robin (RR) or weighted round-robin (WRR) arbitration, we have used synthetic traffic generation in gNoCsim simulator. The gNoCsim simulator by itself offers the possibility to generate different types of traffic in the NoC configuring some parameters such as:

- Size of short and long messages.
- Size of the links, packets, input link buffers.
- Percentage of short and long messages.
- Insertion rate in each core of the NoC.

Since the focus of this thesis is analyzing the worst-case impact of network contention, we have focused on the all-to-one traffic pattern since it is the one creating the worst-contention in a specific target, and allows modeling the case where all cores attempt to access a shared memory.

4.2.2 Real Traffic

4.2.2.1 Resource Stressing Kernels

As benchmarks we have used and created specific resource stressing kernels using C language and PowerPC assembly. The idea was to create workloads for which we can specify the exact fraction of instructions of each type and distribute them randomly. Additionally, for these benchmarks we can control at very fine granularity the number of requests going to memory by enforcing a specific number of hits and misses for the load and store instructions in each kernel.

For each of the stressing benchmarks generated, we execute 1 million instructions. Table 4.2 summarizes the properties of the different benchmarks employed in the evaluation.

Note that we have not used real applications because we want to have exact control of the amount of traffic generated by each benchmark since this is the most

relevant parameter in the context of the worst-case network performance. Real applications are required to accurately characterize average performance. However, resource stressing kernels are the preferred solution to characterize the impact of contention on the different processor shared resources.

Benchmarks	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
% Local Op	80	50	50	60	95	87.5	87.5	90
% LD Op	10	10	40	20	2.5	2.5	10	5
% ST Op	10	40	10	20	2.5	10	2.5	5

TABLE 4.2: Single-threaded benchmarks

4.2.2.2 Spinlock benchmarks properties

In order to analyze the spinlock effect, we have created 4 different benchmarks all of them with 5% of memory instructions and 95% of integer instructions but with different number of instructions each one, as shown in Table 4.3. These benchmarks are intended to evaluate the impact of spinlocks for synchronization purposes whenever some threads of a parallel application finish their execution and others still run.

Spinlock benchmarks	**Bench0**	**Bench1**	**Bench2**	**Bench3**
% INT Op	95	95	95	95
% LD Op	5	0	0	0
% ST Op	0	5	5	5
# instructions	1.100.000	100.000	300.000	500.000

TABLE 4.3: Spinlock Benchmarks Properties

Chapter 5

Controlling Bandwidth Allocation in NoCs

In this thesis we analyze the potential of using a Flexible Bandwidth Allocation (FBA) scheme to improve WCET estimates. In particular, we explore how assigning each core with a given fraction of the available shared resources bandwidth impacts the execution time of parallel applications. The idea behind FBA is that by allowing a fine-grain allocation of bandwidth, we can better exploit the computing resources of manycore systems by enabling a balanced execution time of the different tasks executing on the processor. The goal of the FBA is maximizing the overall performance guarantees of the system.

Applications parallelization is a well known topic in the high performance computing domain. In CRTES domain, parallelization of the application also has to preserve timing guarantees. This introduces an additional set of challenges: *(i)* time-predictable parallel software patterns and algorithms in order to facilitate the parallelization of legacy code as well as the development of new applications, *(ii)* time-predictable low-level synchronization primitives and *(iii)* hardware support for "fair load balancing". It is in this last aspect in which we focus our attention.

In parallel programming, a significant effort is devoted to balancing the workload of application threads i.e. sizing the computation chunk of each thread evenly. However, regardless of the programmer's effort, load imbalance can occur, either *intrinsic* (e.g. due to a specific input set) or *extrinsic* (due to, from an application perspective, external factors such as the hardware or the operating system). We focus on the hardware part.

Non-uniform memory access (NUMA) architectures are commonly employed in systems with a high number of cores. Processors including a high number of cores are typically interconnected using a NoC that interconnects the cores, memory resources, and processor peripherals using a given topology (e.g. mesh, torus, tree, etc.) that determines the latency in the communication flows (e.g. from cores to memory). On NUMA systems, the latency which a given task experiences to access memory depends on the actual location where this task is mapped to. While this uneven latency distribution affects the task's execution time, its impact is in general reduced since, in terms of average performance, caches help to minimize the number of accesses to the different shared resources and this makes at the same time the contention experienced by the requests to be low. However, since in general, deriving WCET estimates requires accounting for the worst potential contention, the impact of NUMA is not negligible in this case. In fact, to calculate the WCET in a correct way, we need to be conservative and assume the worst-case scenario in every core-memory request. That means that for every core request, we need to assume the highest contention from the other cores (the core request will be the last one to receive the arbitration grant in the routers in its path).

A FBA scheme can be implemented at the different shared resources arbiters in the processor. However, in this thesis we focus on the impact of bandwidth allocation in the interconnection architecture i.e. the connection between the cores and the shared memories or caches. For instance, in a multicore that uses an on-chip bus to connect the cores to a shared memory controller, FBA requires modifying the arbiter to allow distributing the bandwidth from cores to memory. Assuming a N-core system, a plain bandwidth allocation scheme assigns $1/N$ of the bandwidth to each of the cores, whereas FBA assigns each core the fraction of the bandwidth that maximizes **guaranteed performance**, where such fraction may not be homogeneous across cores.

5.1 Arbiter Design

We implement FBA by leveraging a weighted arbitration. The proposed weighed arbitration is implemented on top of a round-robin arbiter and is based on using weights for the different communication flows to allocate the available bandwidth to the different cores on a flexible manner.

We base our design on a binary tree comparator structure [1] that easily extends to every other NoC topology. In a binary tree comparator, the global arbitration decision is split into multiple simpler arbitration decisions where each decision involves two contenders at most (see Figure 5.1). This kind of structures provides a fast response, but they are not suitable when the number of bits for encoding the priority increases [43].

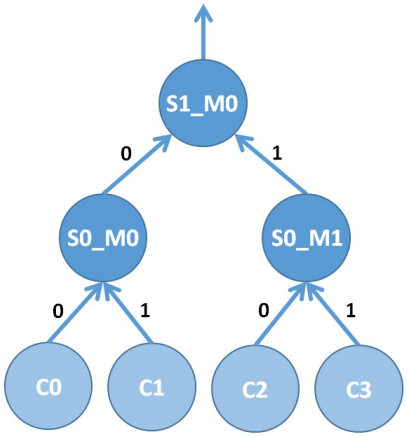


FIGURE 5.1: Binary Tree NoC structure

Each 2to1 arbiter has to make a decision between two contenders. The winner is forwarded to the next 2to1 arbiter until no competitors exist. In a 2to1 arbiter, only those inputs with an active request can be forwarded. When both inputs request the output port, that input with higher bandwidth allocated – weight – wins. If both inputs have the same bandwidth allocated, a fair round-robin (RR) policy is implemented.

We can have different implementations of the weighted round-robin (WRR) policy. The one explained before is one of the more straightforward ones (i.e. the implementation does not take into account which was the port that won the arbitration the previous times). This means that, for similar weights, we can have a good arbitration alternation on the input ports that compete for an output port (both applications can

make progress alternatively). However, whenever we have very different counters, we will have the case where one of the ports always wins the arbitration until the counters equalize. For instance, if port 0 has 5 time slots and port 1 has only 2, the first 3 arbitrations will be granted to port 0 (decreasing the corresponding counter on every arbitration). Then, both counters will indicate 2 time slots for each port, so RR arbitration will occur. Overall, the sequence of arbitration grants will be either 0000101 or 0001010.

Hereafter, we propose two possible implementations of the WRR arbiter policy:

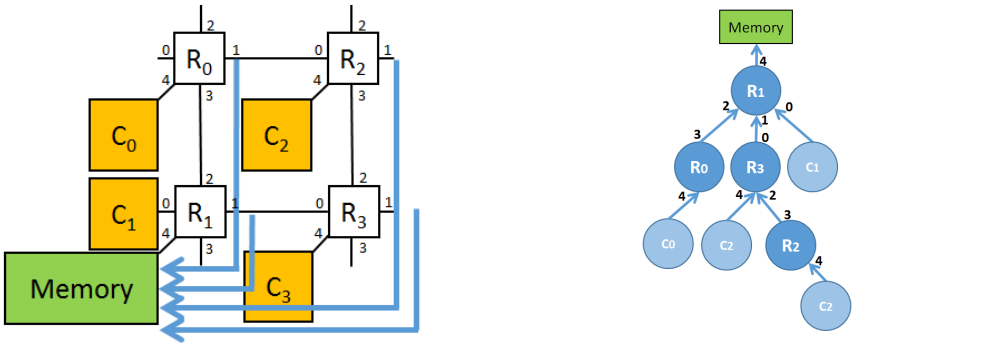
- **Weight counters' implementation**: the first solution is the implementation explained in the 2to1 arbiter. So as to implement this option, each router input port has to store for each output port its current weight and the maximum weight that can have. The arbitration policy grants the arbitration to the input port that has the highest weight that contend for the same output port. When an input port wins the arbitration, its current weight is decreased by one unit. When both input ports that contend for the same output port have their current weights equal to zero, all the input weights are set again to the maximum weight they can have. If the prioritized input port to win the arbitration does not have a ready petition, the arbitration will give the priority to the next input port that has higher current weight counter (e.i the other input port). The maximum weight values are set at boot time and re-set when all input port weights are equal to zero.

  Make notice that every time that weights are set to their maximum value, the order in which the arbitration policy grants the arbitration to input ports can potentially vary not only because of the RR part when all input ports have the same weight value, but also if one input port does not have a petition ready to be sent.

- **Arbitration window implementation**: the second option is to explicitly specify the arbitration window that we want the arbitration to follow per each router output port. This solution incurs in higher hardware overhead for the implementation part (we need to identify each input port, a pointer to traverse the arbitration window,...) but allows to have precise control on the arbitration

that each output port is using. Every time that the window is traversed the same pattern is followed. We can also set at boot time the arbitration windows and all the values we need. In this case the port that has priority is the one pointed by the window arbitration pointer. If this port has a ready petition it wins the arbitration, otherwise we advance the pointer and we look for the next input port that has a petition. We can implement the arbitration window as a circular buffer and whenever the pointer reaches the last window slot, it is automatically set to the first arbitration window position again.

As said before, the binary tree implementations can easily be extended to a generic topology since, in the end, every NoC topology can be described as a tree structure for a given target node, in which each tree node has as many leaves as the total number of incoming links each router has.



(A) 2x2Mesh example with traffic flows

(B) 2x2Mesh tree representation of flows

FIGURE 5.2: 2x2 Mesh representation in tree

In Figure 5.2a we can observe a 2x2 Mesh with 4 routers represented as R_0 to R_3 and 4 cores represented as C_0 to C_3. We can also appreciate that all the routers of the mesh have 5 ports in spite of some of them are not being traversed by the traffic flows shown in the image. All the flows of the figure have as a source point one of the cores and as a target the memory placed at port 4 of R_1. As previously mentioned, every NoC topology can be represented as a tree structure. Figure 5.2b shows how the mesh topology shown in Figure 5.2a can be interpreted as a tree. The tree structure only shows the input and output ports that are traversed by a given set of flows even though we can have potentially a $(P-1)ary$ tree where P is the number of ports that each router has (e.g. we can only have 4 contenders to the same output port as the same output port cannot be competing as an input port).

5.1.1 Implementation cost of a weighted round-robin (WRR) arbiter

In order to assess the cost of achieving a programmable arbitration in the NoC routers, we analyze the cost of the two main alternative implementations explained in the previous section. In this section we account for implementation costs of WRR as the heterogeneous BW distribution to achieve homogeneous BW allocation and we also commend how to be extended to any type of FBA.

Weight counters' implementation. For each output port, we will need 1 counter and 1 register per input port (with similar cost). The counter tracks the pending arbitrations (weight) that decreases along the execution and the register keeps the maximum value to set as initial value whenever all counters in the port reach zero. That means that for a router with P_{ports} for each potential output port (P_{ports}) we need as many counters as potential input ports for each output port ($P_{ports} - 1$) per output port multiplied by 2 (the counter and the register). So, for each of the P_{ports} per router in each of the $R_{routers}$ in the NoC we will need $2 \cdot (P_{ports} - 1)$ weights. Each such weight has as many bits as needed to codify one flow for each core in the system $\lceil log_2(C_{cores}) \rceil$.

$$ArbCost_{imp1} = R_{routers} \cdot P_{ports} \cdot 2 \cdot (P_{ports} - 1) \cdot \lceil log_2(C_{cores}) \rceil \tag{5.1}$$

For instance, for Figure 5.1, the cost of implementing weight counters would be $3 \cdot 1 \cdot 2 \cdot 2 \cdot 2 = 24$ bits, so 3 bytes only. That is, we have 3 routers and in each router has 1 potential output port and 2 potential input ports per this output port. Each potential input port has to keep information about the current counter (counter) and the maximum value that the counter will have when it will be set (register).

The previous formula easily extends for other topologies like meshes. In the case of meshes, $R_{routers} = (NxM)$ where N and M are the X and Y dimensions of the mesh respectively. For example, for a 4x4 2D mesh with 16 cores, the cost of the implementation would be $16 \cdot 5 \cdot 2 \cdot 4 \cdot 4 = 2560$ bits, so 320 bytes for the whole NoC (20 bytes per router). That is, for each router (16) we have 5 potential output ports and 4 potential input ports for each of this potential output ports. Each potential input port has to keep information about the current counter (counter) and the maximum value that the counter will have when it will be set (register). The

more cores in the mesh we have, the more bits we need to codify the weights to achieve homogeneous BW allocation (e.i number of cores in the mesh grows).

Arbitration Window implementation: we need an arbitration window for all potential output ports (P_{ports}) that each router has. Each arbitration window needs to have as many entries as number of cores in the NoC (C_{cores}) to allow homogeneous BW allocation in all NoC cores. In each of the entries of the arbitration window, we need to codify the potential values of input ports per each output port that we can have ($P_{ports} - 1$). Thus, we need $\lceil \log_2(P_{port} - 1) \rceil$. This values can be X+,X-,Y+,Y- and Processor/Memory Element (PME) indicating the input port incoming direction. This design is sketched in Figure 5.3). Then, each arbiter also needs a $\lceil log_2(C_{cores}) \rceil$ bit counter pointing to the next entry in the window along with an incrementer for that counter. Alternatively, one could use shift registers with wrap-up for the window and use always the value at a given position (e.g. first position) to determine what port is granted access next.

$$ArbCost_{imp2} = R_{routers} \cdot P_{ports} \cdot \left(C_{cores} \cdot \lceil log_2(P_{ports} - 1) \rceil + \lceil log_2(C_{cores}) \rceil\right) \quad (5.2)$$

For instance, for Figure 5.1 binary tree, the cost of implementing the window arbiter would be $3 \cdot 1 \cdot (4 \cdot 1 + 2) = 18$ bits, so 2,25 bytes only. Or for example, for a 4x4 2D mesh with 16 cores, the cost of the implementation would be $16 \cdot 5 \cdot (16 \cdot 2 + 4) = 2880$ bits, so 360 bytes for the whole NoC (22,5 bytes per router).

If we want to implement FBA not only to homogenize the BW allocation in all the interconnect, the implementation will be more costly. When applying WRR arbitration to homogenize BW it is enough to have counters of $\lceil log_2(C_{cores})) \rceil$ bits or arbitration windows of C_{core} size. However, if we want a desired BW distribution (not following the RR and homogenizing WRR), we will have to change the C_{core} value for the *maximum weight value*.

Some NoC implementations favor efficiency in front of flexibility and arbitration choices are hardwired. Adapting such a NoC to implement the weighted arbitration would require, at most, hardwiring different choices in the arbitration windows, thus not increasing the hardware cost.

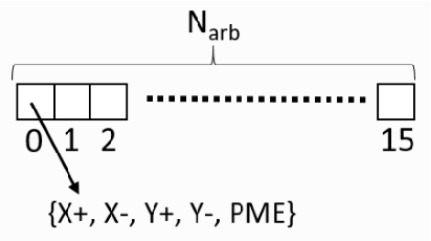


FIGURE 5.3: Window arbitration implementation.

Overall, hardware modifications would have limited impact on the overall cost of the NoC, which is mostly dominated by the buffering required at input ports.

5.1.2 Adapting arbitration weights

In weighted round-robin arbitration (WRR) to achieve homogeneous core BW allocation, weights can be used to determine the frequency at which a given master gets access to a given shared resource. In a NoC router with $N_{IP_{contenders}}$ input ports contending for the access to a output port (OP) different arbitration weights can be employed for each of the input ports (IP_{weight}) provided that the following conditions are meet:

$$OP_{BW} = \sum IP_{weight} / N_{IP_{contenders}} \tag{5.3}$$

The equation above, simply illustrates formally that the total bandwidth (BW) of the output port has to be shared by the different input ports. For instance, for a plain round-robin arbitration, the weights are all 1 since all input ports are allocated the same bandwidth. However, round-robin arbitration does not distribute the bandwidth fairly in the context of NUMA-based network topologies like the mesh. Weighted arbitration can be employed to homogeneously allocate the BW. To do so, weights can be computed using the following expression:

$$w(I_{dir}, O_{dir}) = I_{dir} / O_{dir} \tag{5.4}$$

where I_{dir} represents the number of communication flows traversing the dir_i input port of a given router being dir any of the possible mesh router port directions. Similarly, O_{dir} is the number of flows traversing the dir output port of the same router.

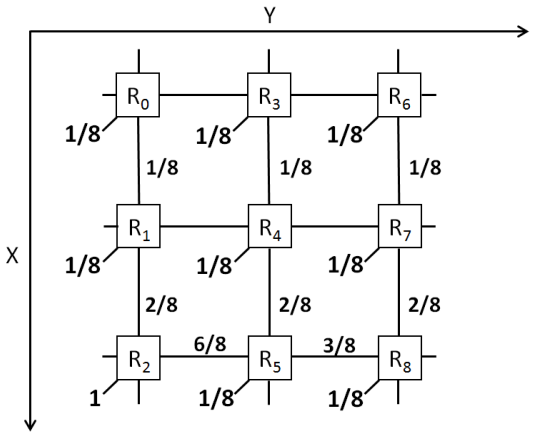


FIGURE 5.4: 3x3 mesh weight using XY routing and WRR arbitration

We can configure our mesh input ports' weight for each output port as we show in Figure 5.4, which in practice means:

- In both WRR arbiter implementations, we can set in R_2 for the memory output port, the weights 1 and 3 for ports that come from R_1 $(X+)$ and R_5 $(Y-)$ respectively and weight 0 to all other input ports. Analogously for R_5, having as an output port the port that goes to R_2, we can assign the weights 1, 3 and 2 to input ports that come from C_5 mapped in R_5 as a Processor/Memory Element (PME), R_8 $(Y-)$ and R_4 $(X+)$ respectively.

- In the particular case of the arbitration window implementation, in the memory output port of R_2 we can use the window $X+, Y-, Y-, Y-$ being $X+$ the input port that comes from R_1 and $Y-$ the input port that comes from R_5. Analogously, for the output port that goes to R_2 router in R_5, we can set as arbitration window the combination $PME, Y-, X+, Y-, X+, Y-$ being PME the input port that comes from C_5 mapped in R_5, and $Y-$ and $X+$ the input ports that come from R_8 and R_4 respectively.

Router id	**X+**	**X-**	**Y+**	**Y-**	**PME**	**Router id**	**X+**	**X-**	**Y+**	**Y-**	**PME**
*R*0	0	0	0	0	1	*R*5	2	0	0	3	1
*R*1	1	0	0	0	1	*R*6	0	0	0	0	1
*R*2	1	0	0	3	0	*R*7	1	0	0	0	1
*R*3	0	0	0	0	1	*R*8	2	0	0	0	1
*R*4	1	0	0	0	1	-	-	-	-	-	-

TABLE 5.1: Weights per input port direction for a 3x3 Mesh with WRR

5.2 On-Chip Interconnection Architectures

We apply the FBA scheme to two of the common NoCs topologies implemented in current multicore processors like the tree and the mesh. For each topology, we analyze its particular behavior and propose solutions on how to take advantage of using a FBA scheme.

5.2.1 Tree

We consider a tree NoC topology as the one proposed in [32]. This topology is also implemented in real processors like the P2012 [6] and it can serve as the basis to efficiently implement crossbar topologies as shown in [32]. In tree NoC topologies all the cores are at the same distance from the memory. Thus, we have intrinsically unified memory access (UMA) in all the cores. Having UMA in the NoC, allows basic RR arbitration to homogeneously allocate BW across all the cores.

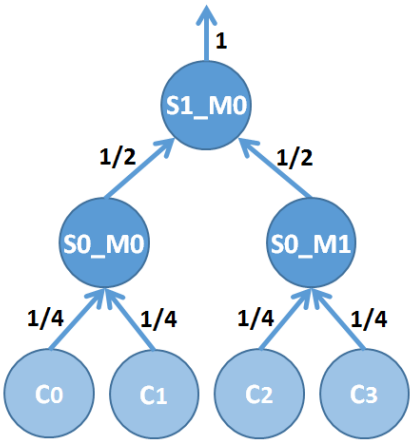


FIGURE 5.5: Binary tree with RR arbiter

When we apply homogeneous BW distribution in each router (i.e. using RR arbitration), we observe homogeneous BW allocation as shown in Figure 5.5. In each one of the routers, the RR arbitration policy divides by two the output BW to the input ports that are contending for this output port. Applying this BW distribution recursively, we end up with the weights shown in Figure 5.5. In this case, as we have 4 cores, each core receives $1/4$ of the available BW. We can generalize this BW distribution in UMA NoCs in the following way: if we have N cores, when we follow the RR policy to distribute the BW along the tree, we end up with $1/N$ of the initial BW per core.

5.2.2 Meshes

In this section we analyze the mesh topology. Meshes, unlike trees, have intrinsically a non-unified memory access (NUMA) since, depending on the core location in the mesh, it takes more or less time to access memory. On the one hand, we have different latencies, which translates to different execution times from the same application depending in which core we run the application even in isolation (i.e. the execution time of the application does not depend on other contenders). On the other hand, in the presence of contention, the fact that RR arbitration is applied in every router along the path from source to destination, causes an uneven distribution of bandwidth. This phenomena is illustrated in Figures 5.6 (flows) and 5.7 (weights).

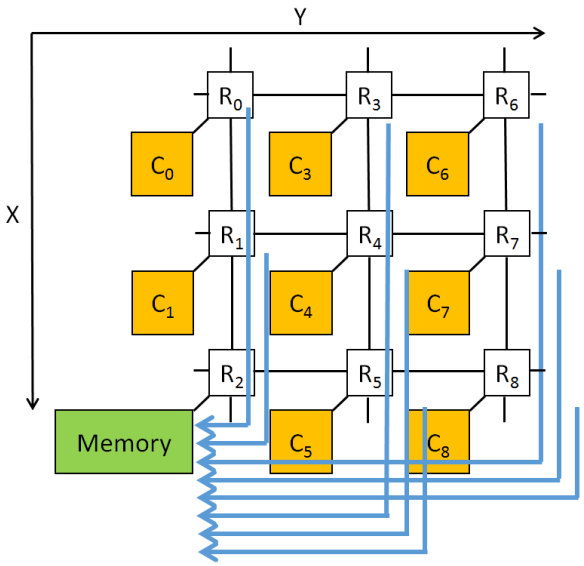


FIGURE 5.6: 3x3 mesh flows using XY routing algorithm

In Figure 5.6, we can see which are the traffic flows when each core injects messages to the 3x3 mesh targeting the memory controller located in R_2. We show how flows are mapped in the mesh following XY routing algorithm which, as we have already explained in section 2.2.2, is one of the most used routing algorithms since it is easy to implement in hardware and it has deadlock/livelock free properties.

As said before, when we apply homogeneous BW distribution in each mesh router (i.e. using RR arbitration), we observe globally heterogeneous BW allocation. We can observe this phenomena in Figure 5.7. Note that the example we propose in this figure shows a 3x3 mesh NoC with 8 cores as we connect a memory module in R_2 instead of C_2. With this RR BW distribution, we have C_0 and C_1 with 1/4 of

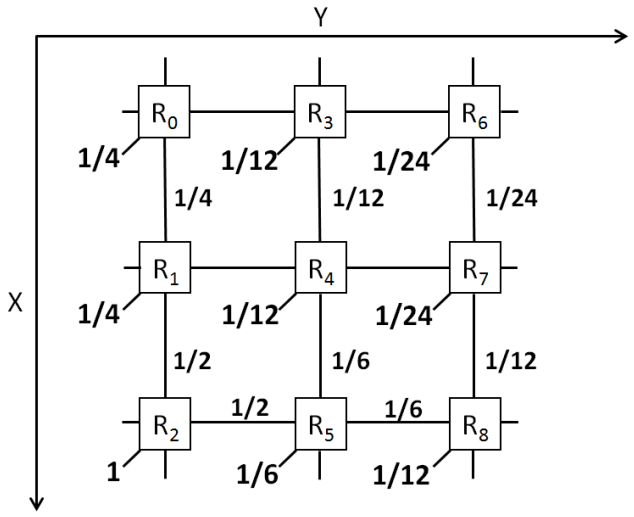


FIGURE 5.7: 3x3 mesh weights using XY routing and RR arbitration

the BW, C_5 with 1/6, C_3, C_4 and C_8 with 1/12, and C_6 and C_7 with 1/24 of the BW. As we can notice, this BW distribution is highly related with the distance from each core to the memory (placed in this case in R_2) as the farther cores from memory (C_6 and C_7) tend to have less BW, even though it does not follow this trend perfectly (C_6 and C_7 receive the same amount of BW but C_6 is farther away in number of hops from memory as it has 4 hops whereas C_7 it has 3).

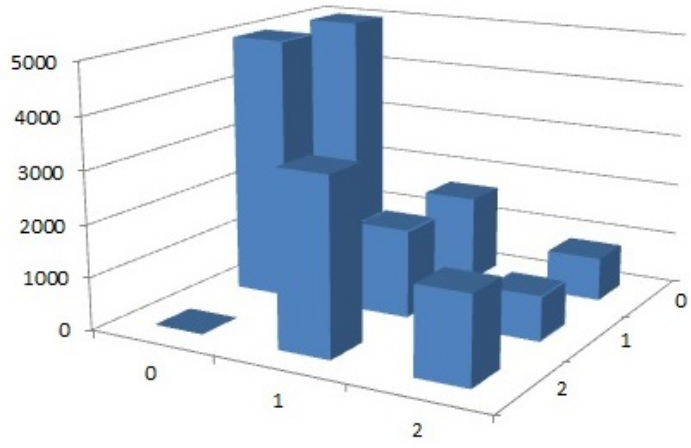


FIGURE 5.8: Messages sent per NIC in a 3x3mesh XY routing and RR arbitration

In Figure 5.8, we can observe the imbalance between messages sent per core which is strongly related to the imbalance that the mesh topology brings intrinsically when using RR arbitration. As we have explained, cores that are closer to the memory location have higher BW (i.e. C_0 and C_1) than the ones that are farther away (i.e.

C_6), which allows them to be granted more arbitration rounds than the others. This plot also shows that, when we send 20000 messages along the mesh, C_0 and C_1 are able to send 5000 messages each (25% of the BW),C_3 3333 messages (33% of the BW), C_3, C_4 and C_8 1667 messages (8,33% of the BW) and C_6 and C_7 833 messages (4,1% of the BW), which matches perfectly with the heterogeneous distribution showed in Figure 5.7.

When we deal with a 2D mesh as a NoC, we have also to take into account the real latency distribution, since messages sent from cores farther away from memory will take more time to arrive to memory than the messages sent from the closer cores. In Figure 5.9 we show the latency (time between a message is being sent and it arrives to the memory) that messages have experienced depending on from which core have been sent.

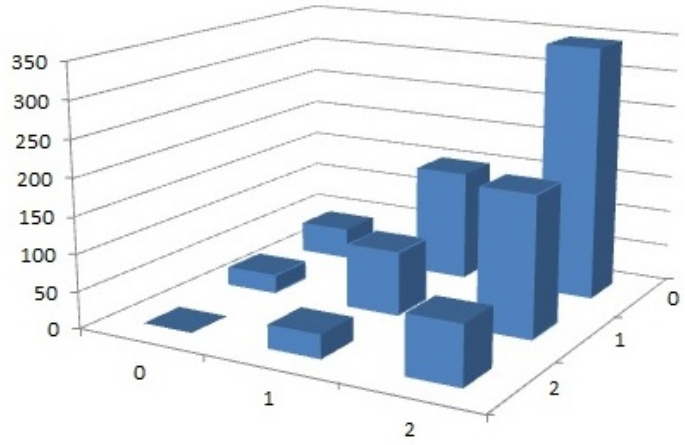


FIGURE 5.9: Messages latency

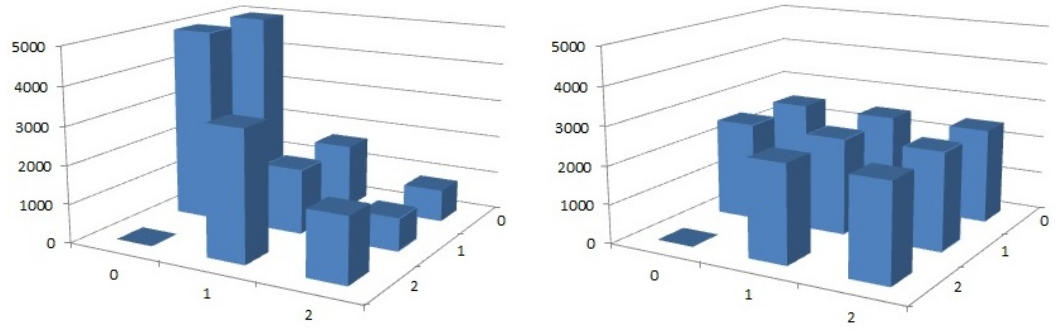
As said before, figure 5.9 shows the average latency that messages sent from all cores have experienced after reaching memory when all the cores are injecting at a 1.0 rate (1 message/cycle). The results shown are quite intuitive as C_6, that is the farthest core in the system, is the one that has more latency reaching the memory. After C_6, that has 4 hops until memory, we have C_7 followed by C_3. Even though C_7 and C_3 are at the same distance from memory (3 hops), C_3 has lower average latency because it has twice the BW allocated than C_7 (1/12 of the BW against 1/24). Observing that phenomena, we can say that, in non-uniform NoC topologies, the latency that messages experience from source to destination nodes does not only depends on the distance in hops between source and destination, but also on the

amount of BW that the source has been allocated.

Having the latency and BW properties of 2D meshes, we can take advantage of this and, for example, place the less memory demanding applications to cores far from memory with low BW allocated, and the more memory demanding ones in cores closer to memory, so with higher allocated BW.

However, in case of having a well-balanced parallel application, or applications with threads having similar memory requirements properties, the task running for instance in C_1 (messages with average latency of 24.1 cycles) and the same task in C_6 (messages with average latency of 345.1 cycles) execution will be 14,3x faster in C_1 than in C_6. This BW and latency imbalance in 2D meshes, when applying homogeneous BW distribution locally at each router, can cause a big drop in performance of well-balanced parallel applications where the execution time of the entire application is determined by the thread that needs more time to complete its execution.

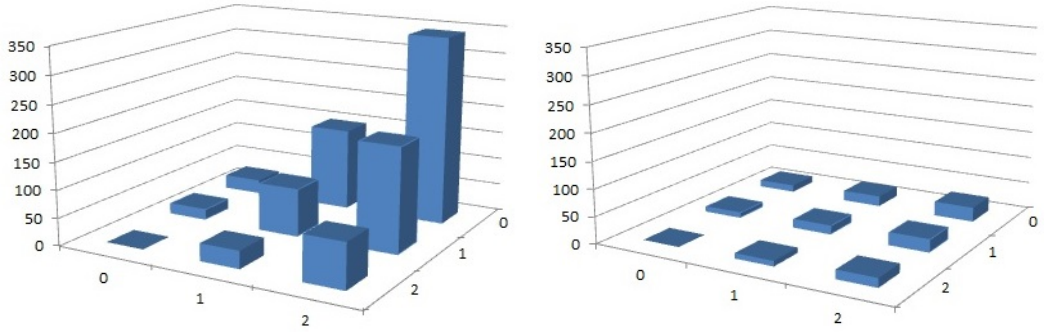
This worst-imbalance case between latencies of different threads only happens when all the cores or NICs in the mesh work at injection rates greater than 0.2 (1 message inserted every 5 cycles). As we know, arbitration BW distribution effectiveness is strongly related to the NoC utilization. Thus, when we decrease the injection rate in cores' NICs under 0.2, RR arbitration policy loses effectiveness very fast. However, computing WCET estimates requires making pessimistic assumptions on the load other contenders can put in the network making the quality of these estimates to be heavily degraded as a consequence of this uneven distribution of bandwidth, even if, in practice, real contention is lower.



(A) Injection rate 0.2 per core (B) Injection rate 0.1 per core

FIGURE 5.10: Messages sent per core in a RR 3x3Mesh varying the IR

In Figure 5.10 we can observe this arbitration effectiveness drop when reducing from 0.2 to 0.1 the injection rate (IR) in all 8 cores in the mesh. In Figure 5.10a we can observe that cores have the same BW distribution as in Figure 5.8, when the IR was 1.0 (NICs are able to send messages accordingly to their BW allocation). However, in Figure 5.10b, when we have an IR of 0.1 (1 message every 10 cycles), the BW distribution is homogenized as the mesh NoC is no longer the bottleneck of the system (messages can flow without waiting in the NIC when they are injected). In Figure 5.10b, we observe that all cores send more or less the same number of messages (around 2500 messages), which correspond to 1/8 of the BW, so it matches with an homogeneous BW allocation.



(A) Injection rate 0.2 per core (B) Injection rate 0.1 per core

FIGURE 5.11: Messages latency per core in a RR 3x3Mesh varying IR

We can also observe this effectiveness loss in the arbitration when we analyze the latency values evolution when passing from 0.2 to 0.1 cores' NICs IR. In Figure 5.11, we can see, similarly as in Figure 5.10, a drop in the latency in all cores. The most important change, as expected, is observed in the farthest memory core of the mesh (i.e. R_6 where C_6 is placed) where we move from having a latency of 342.7 cycles with a 0.2 IR to having 26.9 cycles with 0.1 IR. In other words, reducing the injection from 0.2 to 0.1 involves a message latency reduction in the farthest core of the mesh of 12.75x. Note that, in such a NoC, whenever the joint IR of the NoC is above 1 (e.g. $0.2 \cdot 8 = 1.6$ for the 8 cores), the NoC saturates and contention is constantly the maximum. However, when the overall IR is below 1 (e.g. $0.1 \cdot 8 = 0.8$ for the 8 cores), the NoC is able to eject packets faster than they are injected, so interaction among packets in routers is negligible – if any.

5.3 A model for computing worst-case delay (WCD) from the allocated bandwidth

WCET estimation in manycores needs bounding access times to shared hardware resources [26, 11]. In the case of NoCs, this translates into i) bounded WCD such that every request sent to the NoC has a service time, or traversal time, boundable at analysis time; and ii) time-composable WCD such that the bound to the traversal time derived for the request of a task does not depend on the load put by other co-running tasks on the NoC. Low WCD translates into tighter WCET estimates, which allows increasing the guaranteed performance that the manycore chip can provide.

5.3.1 Baseline NoC

We model a canonical 2D wormhole mesh router comprising five input ports that have queues to store packet FLITS. The router arbiter grants an output port to a given input flow. To be able to have time-composable WCD estimates, no prioritization mechanism is used in the router, and arbitration decisions to select the flow accessing the requested output port are taken using a time-analyzable arbitration policy, e.g. round-robin.

We consider a *NxM* mesh NoC configuration as depicted in Figure 5.7, in which each node can be identified using (x,y) coordinates. Each node comprises the router that communicates the node to the mesh and a *PME* (Processor/Memory element). The PME can be either a processor core, a cache memory, main memory, I/O, etc. In the network, several traffic flows may exist. A traffic-flow (F_i) is a packet stream that traverses the same *H*-node route from a source to a destination node and requires the same grade of service along the path.

We use deterministic *XY* routing as explained before. It further enables identifying routers in a given path as R^j where j is the hop number of the path (e.g. R^1 is the source node). With XY routing packets are forced to use the X dimension first. In the X dimension the position of the target node with respect to the source node determines whether to go right (X+) or left (X-) direction. The same approach is used for the Y dimension. Once packets are routed using the Y dimension they cannot be forwarded to the X dimension. Note that the opposite port is represented as $\bar{Y}$ and

$\bar{X}$. For instance the opposite port of $Y+$ is $Y-$. Routing restrictions help determining the exact number of requests (P_i^j) that might contend at router R^j for the same output port as F_i, in the worst-case situation. P_i^j values can be determined as follows:

$$P_i^j = \begin{cases} 2 & if\ destination\ is\ X+\ or\ X- \\ 4 & if\ destination\ is\ Y+,\ Y-\ or\ PME \end{cases}$$

5.3.2 Accounting for the Impact of Bandwidth Allocation in WCD

WCD values can be derived for regular NoC designs following the expressions given in [28, 23]. In this section, we provide expressions to compute WCD bounds that are also suitable for NoCs using weighted round-robin arbitration. The expressions given in this section are based on the concept of worst-case ejection rate (ER_i^j). We define ER_i^j as the rate at which FLITS of flow F_i can be ejected from router R^j to the corresponding port when the next router (R^{j+1}) is accepting incoming packets (i.e. it is not stalling R^j packet transmission). We also extend the concept of worst-case network ejection rate to model the rate at which FLITS can be ejected from a given router port when the network is fully congested. To do so, we define propagated worst-case ejection rate PER^{wc} as the minimum rate at which FLITS of F_i can be ejected from R^j in the worst-case situation. ER_i^j values can be computed by considering the maximum number of flows P_i^j contending at R_i^j for the same output port as F_i as shown in Equation 5.5.

$$ER_i^j = \frac{1}{P_i^j} \tag{5.5}$$

Then, $PER_i(R^j)$ is computed by multiplying ER_i^j factors from the current router R_i^j to the target router R_i^H as presented next:

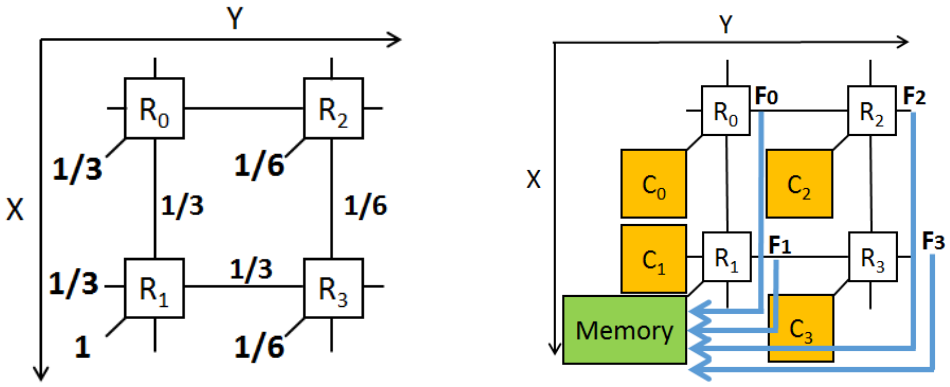
$$PER_i^j = \prod_{k=j}^{H} \frac{1}{P_i^k} \tag{5.6}$$

Let D_i^j be the time that a packet of flow F_i requires to go from the input port of R^j to its destination node. D_i^j can be computed recursively by considering the time required to reach R^{j+1} ($1/PER_{fx\{i\}}^j$) plus the time required to reach its destination once

at R^{j+1}. $fx\{i\}$ represents the index of the flow that causes the worst possible blocking in F_i. Note that a $F_{fx\{i\}}$ packet stalled in a subsequent router of the path followed by F_i might cause F_i to suffer worst contention than one following exactly the same path. In the same way $PER^j_{fx\{i\}}$ represents the worst ejection rate for F_i packets. To determine the flow causing the worst contention, PER values for all routers and all flows have to be computed in advance, and for any particular flow and router we choose the worst $PER^j_{fx\{i\}}$. Equation 5.7 shows the recursive definition of D^j_i.

$$D^j_i = \frac{1}{PER^j_{fx\{i\}}} + D^{j+1}_i \tag{5.7}$$

The WCD for flow F_i, given by D^1_i, is the time required to reach its destination ($j = H$) from the source node ($j = 1$).



(A) 2x2Mesh BW distribution with RR

(B) 2x2Mesh flows

FIGURE 5.12: 2x2 Mesh with 4 cores

We illustrate how to compute WCD using Equations above with the example presented in Figure 5.12 and considering round-robin arbitration first. We aim at computing the WCD of packets with source node C_2 in router R_2 and destination memory in router R_1. First, we compute PER^j_i as the product of the ER^j_i coefficients of all the routers that F_i ($i = 2$) traverses. Later, we start from the last hop ($j = 3$) and compute all D^j_i values.

$$D_2^3 = \frac{1}{1/3} = 3$$

$$D_2^2 = \frac{1}{1/6} + D_2^3 = 9 \qquad D_2^1 = \frac{1}{1/6} + D_2^2 = 15$$

Table 5.2 provides the D_i^j values for all flows in a 2x2 mesh with RR and WRR arbitration respectively.

	Round-Robin				**Weighted Round-Robin**			
	F_2	F_3	F_0	F_1	F_2	F_3	F_0	F_1
D_i^3	3L	-	-	-	2L	-	-	-
D_i^2	9L	3L	3L	-	6L	2L	4L	-
$D_i^1(WCD)$	15L	9L	6L	3L	10L	6L	8L	4L

TABLE 5.2: WCD values for L-flit packets, where the maximum allowed packet size is L.

5.4 Computing WCET in NoC-based processors

Once we have the worst contention (WCD) that each of the requests in the NoC is exposed to, we can compute the WCET using the following expression:

$$WCET = OET + WCD \cdot N_{req} \tag{5.8}$$

where OET stands for the observed execution time and N_{req} is the number of requests of the particular task traversing the NoC. The OET is computed by executing applications in isolation (i.e. in the absence of contention). N_{req} can be determined by the statistics collected when tasks are executed in isolation. If there is no specific counter for the number of NoC requests, this number can be obtained from the number of misses and evictions in the different caches and/or the number of write operations depending on the employed cache write policy.

Chapter 6

Evaluation Results

In this chapter we show the results and performance characterization of applying FBA in tree and mesh topologies for both single-threaded and multi-threaded workloads. We also want to mention that we have used the window arbitration implementation of FBA as it allows to have precise control on the arbitration that each output port that applies WRR arbitration policy is using.

6.1 Performance characterization of workloads

In this section we analyze the impact of executing applications in a 4-core tree based architecture. To do so, we have used the synthetic benchmarks mentioned in Table 4.2. Results shown in this section have been obtained using Soclib and the gNoCsim simulator with the NGMP configuration described in chapter 4. The details of this processor architecture were given in Table 4.1.

6.1.1 Execution time in Isolation

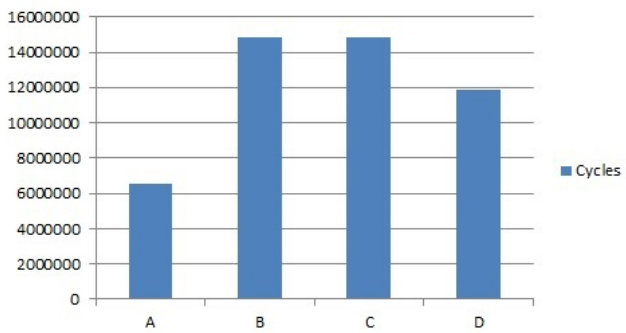


FIGURE 6.1: Tree isolation benchmarks results

Figure 6.1 shows the execution time of the generated benchmarks in isolation. We focus first in the first 4 benchmarks (A, B, C and D), which are the ones imposing higher contention in the NoC. Later in this chapter we also analyze the remaining ones. As expected, since all benchmark have the same amount of instructions, their execution time is deeply related to the fraction of memory accesses (LD and ST) that each benchmark has. For this reason, A benchmark is the fastest benchmark executing in 6.559.367 cycles (20% of memory accesses) followed by the D benchmark (60_20_20) with (40% of memory accesses) executed in 11.882.442 cycles and in the last position benchmarks B (50_10_40) and C (50_40_10) with the same number of memory accesses (50%) executed in 14.872.529 and 14.852.995 cycles respectively.

6.1.2 Homogeneous executions

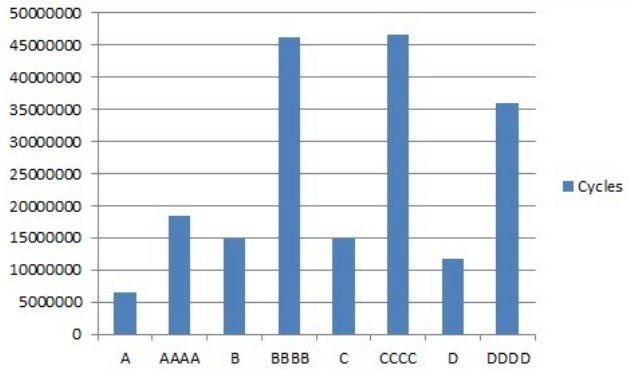


FIGURE 6.2: Comparison between multicore homogeneous and isolation benchmarks results

In Figure 6.2 we compare the execution time (in cycles) between A, B, C, D benchmarks executed in isolation in the tree and the same benchmarks when they are executed concurrently in each of the cores of the system simultaneously. We refer to this concurrently executed workloads as AAAA, BBBB, CCCC and DDDD. As it is expected, the execution time of multi-thread executions are higher than the isolations executions because there are collisions between the cores in the interconnection and the memory. Note that to avoid inter-task conflicts in the shared L2 we have used way partitioning, so that each core has its own L2 cache space and no mutual evictions can occur. In the plot, we can also see that the increment in the parallel

execution time also depends on the fraction of memory accesses that A, B, C and D benchmarks have. We can also observe this timing increase in Table 6.1.

Benchmark	Execution time	Growth
A	6,559,367	
AAAA	18,540,331	2.83x
B	14,872,529	
BBBB	46,125,611	3.10x
C	14,852,995	
CCCC	46,585,611	3.14x
D	11,882,442	
DDDD	36,016,375	3.03x

TABLE 6.1: Growth between isolation and homogeneous execution

6.1.3 Heterogeneous executions

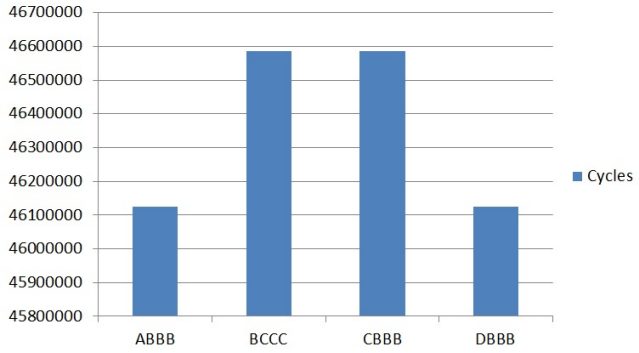


FIGURE 6.3: Heterogeneous benchmarks results

In this section we analyze the impact of having concurrent heterogeneous workloads. These results are shown in Figure 6.3 (note the narrow y-axis scale). As it can be seen, the execution times obtained in heterogeneous executions are closer between them than the ones obtained in isolation and homogeneous executions (we analyze this in detail in Figure 6.4). That happens in this case, because all the execution combinations have B benchmark inside (the longest one) and the total execution time of each combination is the time that needs the longest benchmark, B, in all combinations. In addition, the time that the benchmark needs to finish it is related to the amount of contention that the other benchmarks running in other cores cause.

A desirable situation would be that, whenever a thread in a parallel application completes its execution, it waits for the others to end without interfering with the

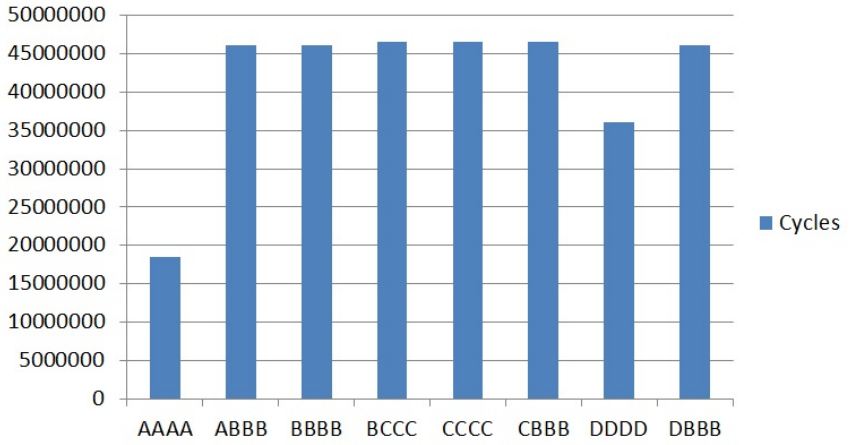


FIGURE 6.4: Comparison between multicore heterogeneous and homogeneous benchmarks results

other cores execution (releasing all the NoC bandwidth required by this core so that it can be used by the other cores). If we were in this case, the performance of the other cores would increase (i.e. every time that one core of the system ends its execution the remaining cores executing have lower contention in the shared resources) but we have observed this is not the case (see the spinlock analysis in Appendix A, section 6.4). In general, the implementation of the barrier and other synchronization calls depends on the runtime and the hardware support that each processor and operating system provides. In our case, the combination of the NGMP atomic operations and the kernel lib that was developed in the context of the parMERASA project [26] created this inefficient synchronization mechanisms based on spinlocks. So, what we observe executing these heterogeneous benchmarks is the following: Let's suppose that parallel execution benchmarks $XYZT$ are mapped X to C_0, Y to C_1, Z to C_2 and T to C_3, where X, Y, X and T benchmarks refer to one of the A, B, C or D defined benchmarks. Whenever C_i ends its execution, it waits making a spinlock loop checking if the other cores have already finished. The spinlock can be translated as a loop of loads that directly go to memory as all the cores are checking for the same shared variable (i.e. we have memory contention). This spinlock situation in C_i causes a performance loss in the other cores execution as C_i behaves as a benchmark all whose instructions are loads from memory. To avoid this undesired behavior, we have modified the way the spinlock is implemented as described in Appendix A (see in Section 6.4).

6.2 Tree-NoC arbitration and bandwidth allocation analysis

In this section we evaluate the effectiveness of the FBA scheme in tree NoC topologies. First, we corroborate the tree topology provides a fair BW distribution in all cores. To do so, we have used synthetic traffic to simulate the bandwidth and the latency of the packets in the network when using different types of messages.

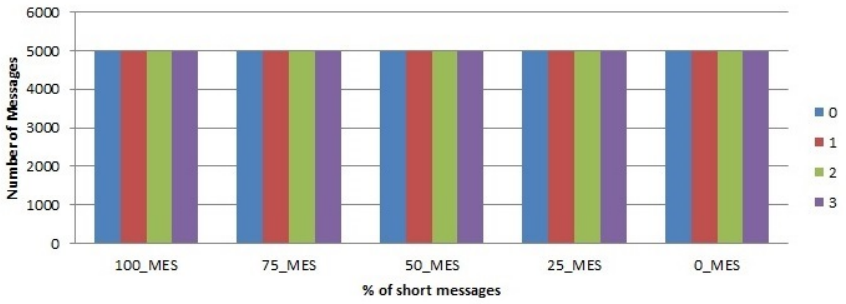


FIGURE 6.5: Messages sent per NIC with 1.0 of IR and RR arbitration

Figure 6.5 shows the number of messages that each core NIC is able to send. For this experiment we have used the same IR for all the cores in the tree and messages of two different sizes: 4 FLITS for the long messages and 1 FLIT for the short ones. All the experiments have been obtained injecting 20000 messages uniformly in all the NICs in the Tree in order to observe the impact of BW distribution. We have varied the injection rate (IR) and the proportion of short and long messages to observe if there is any impact of this factors in the BW distribution. As we can observe in Figure 6.5, independently of the percentage of short and long messages we are sending (100_MES means 100% of short messages), cores 0 to 3 are sending around 5000 messages each (every core is sending 1/4 of all the messages injected in the NoC). In Figure 6.6 we can observe the amount of short and long messages that are sent as long as we vary the percentage of short and long messages. For example, when we have 100% of short messages we can see that all cores send 20000 short messages (5000 messages each core) whereas when we have 75% of short messages all the cores send 15000 short messages and 5000 large messages uniformly distributed along the four cores. Therefore, our experiments confirm that trees intrinsically have homogeneous BW allocation when we apply an homogeneous BW distribution technique like RR arbitration policy.

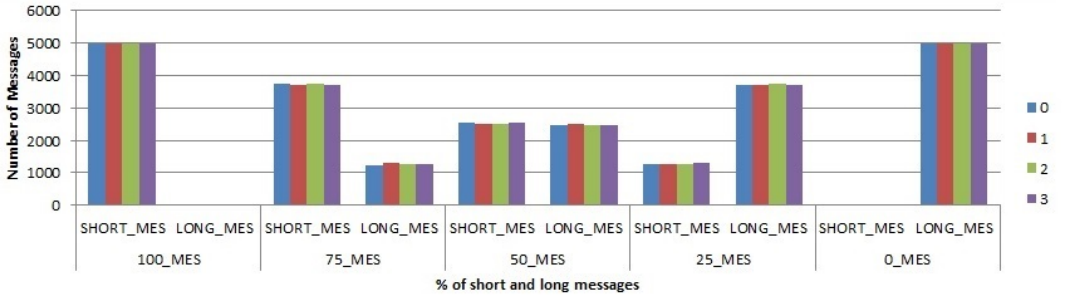


FIGURE 6.6: Number of short and long messages sent per NIC with 1.0 of IR with RR arbitration

Homogeneous BW allocation is specially indicated when the system executes well balanced parallel applications or single-thread applications that have very similar memory requirements at the same time in separate cores. In that case, the system is able to achieve good performance. On the contrary, when we deal with unbalanced parallel applications or single-thread applications with very different memory requirements that run simultaneously in different cores of the Tree, homogeneous BW allocation is not a good option anymore as the threads that have more memory access requirements will run slower meanwhile the threads with lower memory access requirements will not be using the bandwidth that they have assigned. To solve these issues, we can use the FBA based on weighted arbitration that we propose in Chapter 5.

To illustrate this with an example, let us imagine that we want to run at the same time in our 4-core Tree Noc system 4 different applications that can be split in two different types of applications: one group has 10% of memory accesses and the other group has 40% of memory accesses. Based on this we can tune our weighted arbitration values in the Tree routers in order to allocate more BW to the cores where we will map the more memory demanding applications and less BW to the cores stressing less the memory.

Figure 6.7 shows the BW distribution across cores tuning the weights of those input ports that contend for the same upper output port. Given that in this case we have 2 applications of each type, we can modify the BW distribution only changing the weights of $S1\_M0$ input ports (i.e. put a 2 in input port 0 and an 8 in input port 1). By doing that, we can give more BW to the right branch of the tree (80%) and less to the left one (20%). The arbitration weights of $S0\_M0$ and $S0\_M1$ do not need to

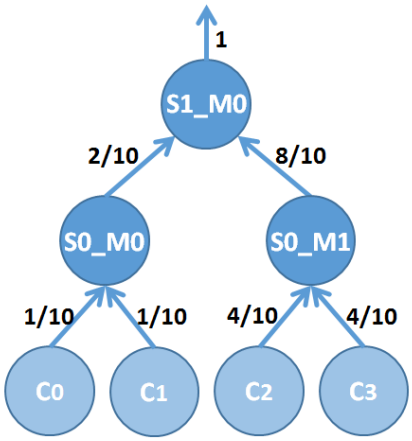


FIGURE 6.7: Binary tree with WRR arbiter

be modified as they keep distributing the received BW in equal parts to their input ports (50% each input port). Applying this changes, we end up with a tree NoC where C0 and C1 have 10% and 10% and C2 and C3 have 40% and 40% of the BW.

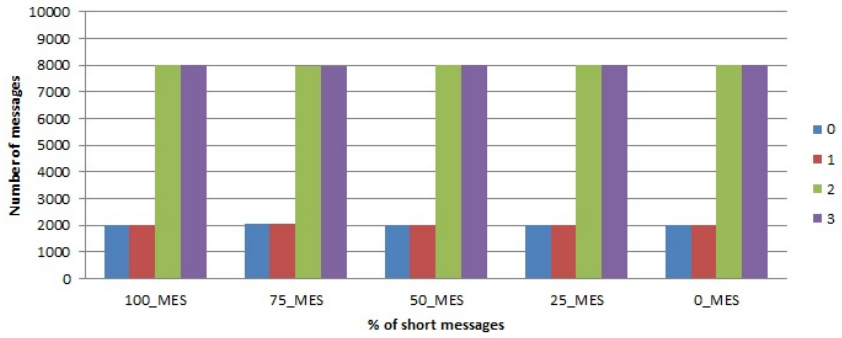


FIGURE 6.8: Messages sent per NIC with 1.0 of IR with WRR arbitration

In Figure 6.8 we can observe a heterogeneous BW distribution in which core 0 and 1 branch has 20% of the BW and core 2 and 3 branch the remaining 80% of the BW distributed (10%, 10%, 40% and 40% from core 0 to core 3). We can observe in Figure 6.8 how cores 0 and 1 send around 2000 messages and cores 2 and 3 around 8000 matching with the the BW in the arbitration windows.

In Figure 6.9 we can also see that, independently of the percentage of short and long messages that a given core injects in the tree, they continue being uniformly distributed inside the BW allocated in each branch.

In this case, as we are using heterogeneous BW allocation (FBA), we know that the effectiveness of this BW distribution using weighted arbitration is strongly related to the IR that NICs have. As we have explained in the Arbiter Design section

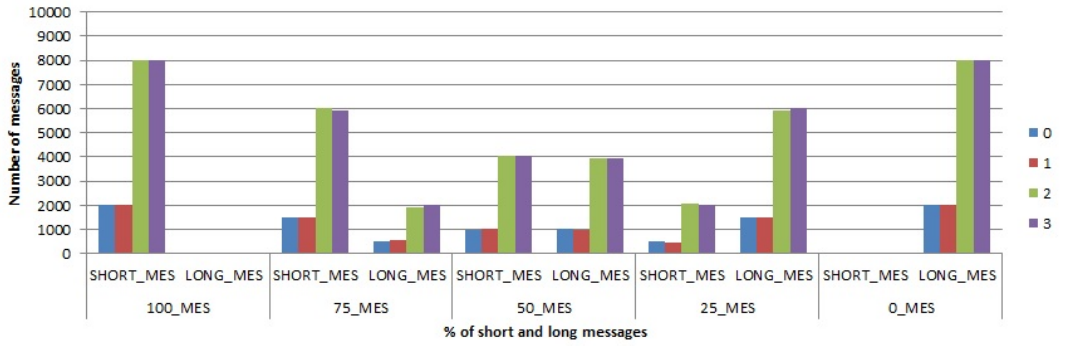


FIGURE 6.9: Number of short and long messages sent per NIC with 1.0 of IR and WRR arbitration

(see Section 5.1), when the prior input port has no request to send, the following input port with more priority wins the arbiter. That means that as long as we have lower injection rates in the system NICs, the effectiveness of the weighted arbitration decreases. To corroborate this hypothesis, we have run the same experiments as before but reducing the NICs IR (all the NICs have the same IR).

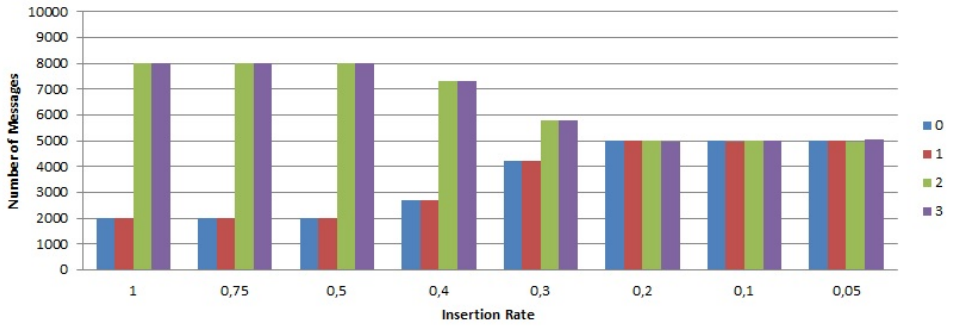


FIGURE 6.10: WRR arbitration impact varying the tree NoC IR

In Figure 6.10 we can clearly observe that when cores decrease below 0.5 their message IRs, the weighted arbitration starts having less impact. We see this phenomena between 0.5 and 0.2 rates. With 0.4 we start observing that cores 3 and 4 send less than 8000 messages, with rate 0.3 cores 3 and 4 only manage to send close to 6000 messages and, finally, with rate 0.2 all cores send the same number of messages (5000 messages each), which is the BW intrinsic distribution of trees. As explained before, arbitration has negligible effects whenever the global IR is below the ejection rate. Hence, IR values below 0.25 do not saturate the NoC and hence, packets progress with virtually no contention at all.

6.3 WRR arbitration analysis

In this section we evaluate the FBA impact in 2D meshes. In order to prove the effectiveness of FBA we use synthetic traffic experiments in gNoCsim simulator using different mesh sizes, injection rates, and message sizes.

We start with a 2x2 2D Mesh with 3 cores that send packets to a memory module. Figure 6.11 shows the amount of packets that each of the different cores is able to send to the destination module. As shown in the plot, applying WRR arbitration we achieve a perfectly homogeneous BW distribution.

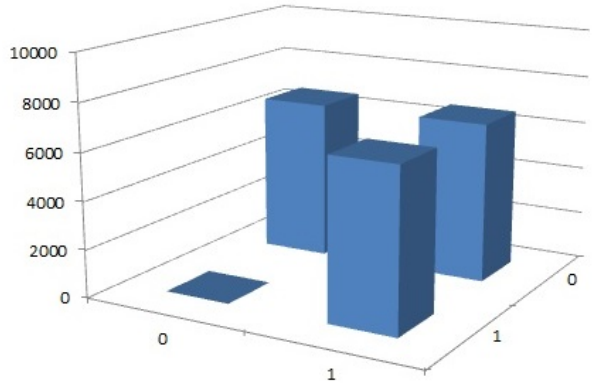
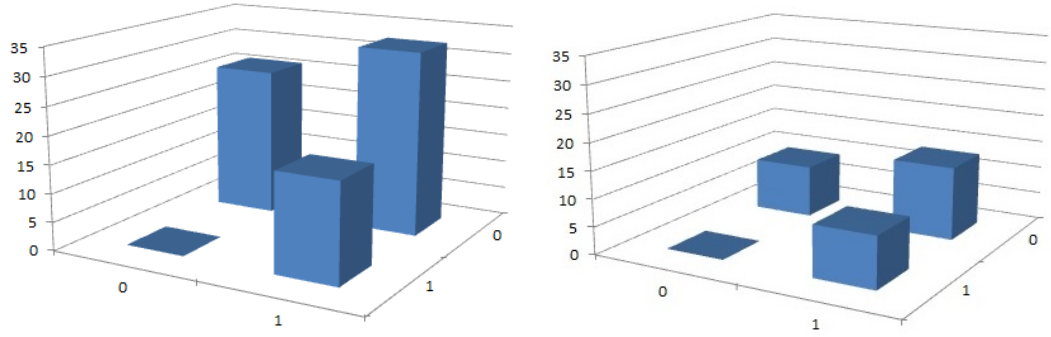


FIGURE 6.11: Messages sent per NIC in a 2x2mesh XY routing 1.0 IR and WRR arbitration

We see that the WRR arbitration, similarly as it was observed for the RR arbitration (see Section 5.2.2), losses effectiveness when the IR of messages in the NoC is reduced. However, since in WRR we already achieve homogeneous BW allocation, the impact of the IR on the number of packets sent per core is roughly null when the IR is reduced in all cores from 1.0 to 0.01. In all tested cases all cores are able to send around 6666 messages, which actualy matches with 1/3 of the BW when we inject uniformly 20000 messages in the 2x2 2D mesh. This loss of effectiveness can be seen between IR 0.3 and 0.2 if we pay attention to the latency of the messages instead. This is shown in Figure 6.12. This effect can be explained by the fact that all the cores can send the packets without suffering congestion anymore, as explained before.



(A) 0.3 injection rate (B) 0.2 injection rate

FIGURE 6.12: Latency evolution in a 2x2 2D Mesh reducing IR

We have also analyzed the behavior of WRR arbitration for bigger network sizes. In particular, we analyze a 4x4 2D Mesh where 15 cores send requests to the same memory device.

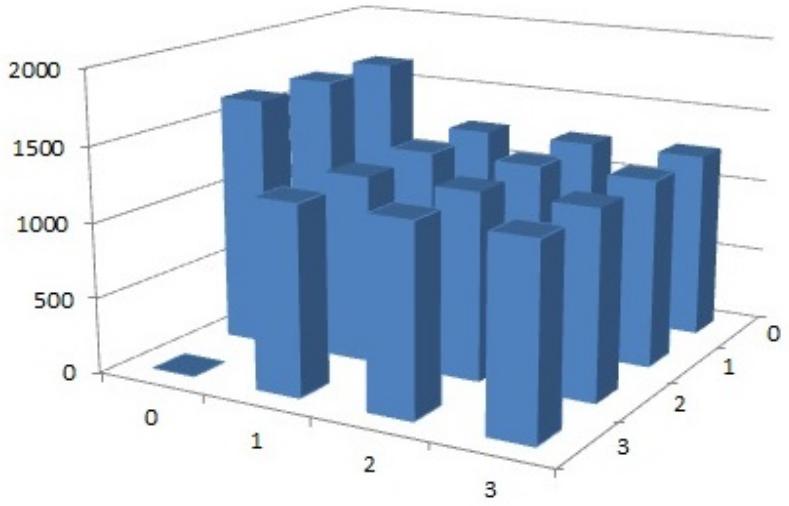


FIGURE 6.13: BW distribution in a 4x4 2D mesh with 1 FLIT messages

In Figure 6.13 we observe that, although WRR is able to balance the traffic much better than RR arbitration, we still observe imbalance between the numer of messages from C_0, C_1 and C_2 compared to the number of messages sent by the rest of the cores. Indeed, the imbalance takes place between the Y+ port (that forwards C_0, C_1 and C_2 flows) and X- port (that forwards the flows of the 12 remaining cores). Whereas we expect to observe near to 1333 messages sent from each core (when sending 20000 messages in a 15 core 4x4 mesh), we observe that C_0, C_1 and C_2 are able to send 1667 messages each one (more than 1333 expected messages) and cores from C_4 to C_{15} are able to send 1250 messages (less than the messages expected).

This phenomena can be explained by the impact that the presence of bubbles can have w.r.t. to the alignment of the arbitration window.

During executions, bubbles can happen because of different reasons:

- Low core injection rate: If one or more than one core have a flow that traverses a certain router with low IR, in some cycles the router will not have FLITS to arbitrate in one or more input ports.

- Flow Control: If there is congestion in a certain path between a core and memory and the FLITS cannot make progress, the Flow Control will be in charge of not allowing FLITS to overflow routers' NICS and the Stop signal will be propagated from the router that is full to the core (ending with a core stall). When this stalled router resumes (FLITS can make progress) the Go signal will be propagated again from the stalled router to the stalled core (or the last stalled router). Traversing all the path until reaching the last stalled router or stalled core can take many cycles and that can introduce bubbles in some pipelines like in the previous situation exposed (the core IR has been 0 during a time period).

The presence of bubbles challenge the efficiency of the WRR arbitration since their presence can cause particular bad alignments w.r.t. the arbitration window, leading to not perfect BW distribution. We elaborate more on this pathological cases in Appendix B (see in Section 6.4).

In our case this phenomena occurs due to flow control in the router microarchitecture (that has 5 stages) since, when we increase the message size from 1 FLIT to 5 FLITS, we are able to obtain a fair BW distribution as the one we were expecting (see Figure 6.14).

Figure 6.14 shows the homogeneous bandwidth expected (1/15 of the BW in each core) when increasing the message size (flow control is no longer producing bubbles in the NoC).

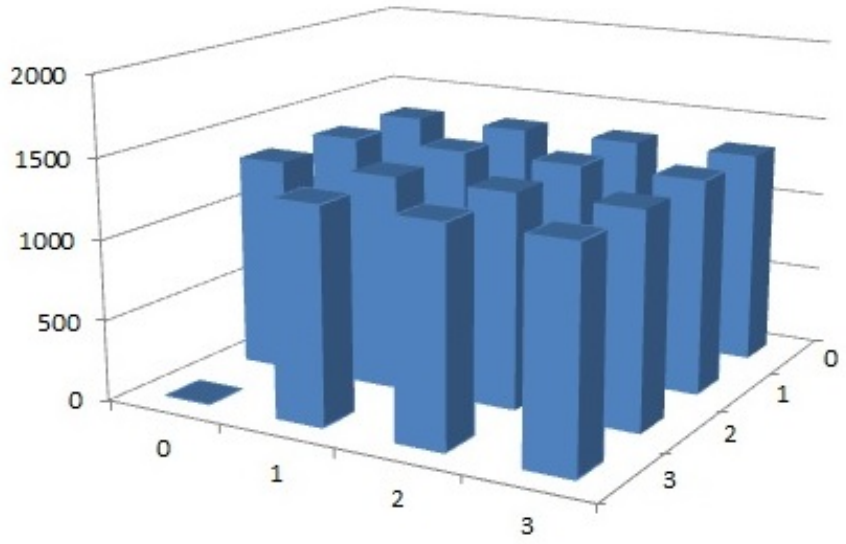


FIGURE 6.14: BW distribution in a 4x4 2D mesh with 5 FLIT messages

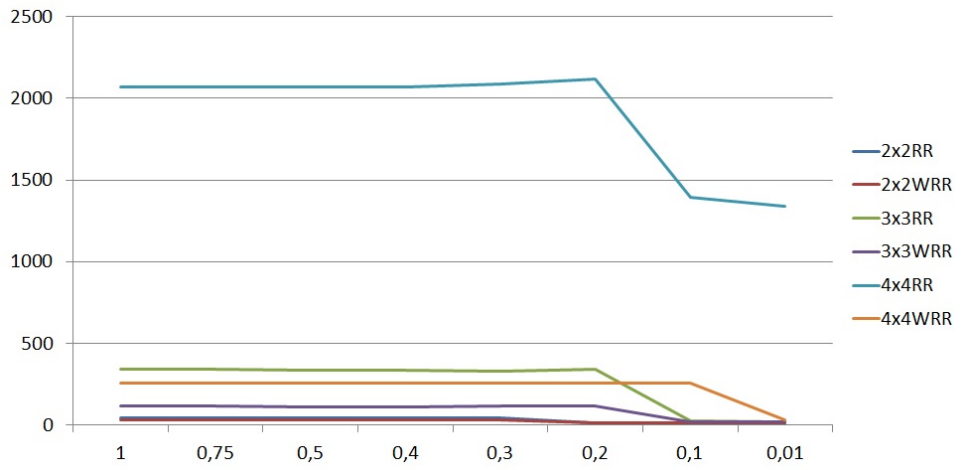


FIGURE 6.15: Message latency comparison with different 2D mesh sizes

Figure 6.15 shows the message latency from the farthest core from memory in 2x2, 3x3 and 4x4 mesh sizes. Looking at the figure, we can underline three particular observations:

- The latency of the farthest core packets grows steadily with the mesh size. This happens because of two quite trivial things. First, when we increase the size of the mesh, it also increases the distance in hops between the farthest core and the memory. Second, independently of the arbitration used, WRR or RR, when we increase the number of cores in the mesh, the same amount of BW has to be split across more cores. So, for example in Figure 6.15 for 2x2, 3x3 and

4x4 meshes with RR arbitration, we observe an exponential growth of message latency from the farthest core from memory. We are reducing the BW in this node and at the same time increasing its distance to memory.

- The distance between RR latency and WRR latency increases with the mesh size. We observe that latencies for 2x2 RR and WRR are practically overlapped, with 3x3 the difference between RR and WRR is more or less 230 cycles, and with 4x4 the difference is around 1,818 cycles. This can be explained by the fact that, whenever we use RR arbitration policy, the BW is homogeneously distributed along the mesh (e.g. every time we split the BW that an output port has among the input ports that contend for this output port in equal parts). When we have a bigger mesh, as the distance between the farthest core from memory and the memory increases, the BW that this core receives is every time smaller and smaller. The bigger the mesh size is, the most unfair BW distribution we have using RR. Still, when we use WRR distribution, the BW is always homogeneously allocated. It is true that increasing the mesh size we reduce the BW per core and increase the distance between memory and the farthest core from memory but, in any case, the BW that the farthest core will receive using WRR will be $1/N$ being N the number of cores in the mesh.

- Small meshes start losing arbitration's effectiveness earlier than big meshes. We observe that in a 2x2 mesh, RR and WRR arbitration start losing effectiveness between 0.3 and 0.2 IR, whereas in 3x3 that happens between 0.2 and 01, and in 4x4 between 0.1 and 0.01. The bigger the mesh we have, the more the BW is distributed along the mesh. With big meshes, the BW is distributed in smaller fractions, which has a good impact in the BW distribution as the interconnection keeps being congested even having less traffic. That is good because the NoC is more sensitive to the weights that we configure and the arbitration keeps distributing the BW perfectly even when we have less traffic inside. In general, given a mesh with N cores, whenever IR is below $1/N$, WRR arbitration loses effectiveness.

6.4 WCET analysis in RR and WRR NoCs

In this section we show the impact that WRR arbitration has in WCET. We only show results for the Mesh NoC topologies since in this topology the effect of FBA is significant regardless of the actual balancing of the parallel applications. Results for the tree follow the same trend when having parallel applications using tasks with different computational requirements. For WCET computation, we have executed the threads in isolation and used the expressions provided in Chapter 5. For the analysis, we use a 4x4 2D Mesh with 16 cores XY routing algorithm.

First of all, we have derived the WCD for each workload. To do so, we have used the model explained in 5.3 and we have applied the method for each benchmark and core using RR and WRR arbitration policies. In other words, we have calculated the WCD for each benchmark in each one of the possible placements (e.g. Benchmark A mapped in C_0 to C_{15}, Benchmark B mapped in C_0 to C_{15}, and so on and so forth) for both arbitration policies. We have executed all the workloads in isolation in each one of mesh placements in order to derive the observed execution time (OET) and number of memory petitions (N_req). In order to emulate the WCD we have chosen the worst possible placement of benchmarks (that is in C_{12} for all the benchmarks, where the WCD is the highest, in RR and also using WRR). Then applying the WCET equation (see in 5.8) we are able to obtain the WCET values for RR and WRR arbitration policies).

Table 6.2 shows for each workload the number of requests going to the network, the execution time in isolation (ET), the WCD of farthest node for both RR and WRR, and the corresponding WCET. As shown in the table, WRR arbitration achieves a significant improvement in WCET (gains between 73% and 83% when using WRR instead of RR). This can also be seen graphically in Figure 6.16 that shows the WCET reduction achieved by WRR w.r.t RR. The main reason for this improvement is caused by the huge penalization incurred by the farthest node with RR arbitration since this is the one that fill require more time to complete the execution and thus, the one determining the WCET of the application. In the same plot, we can also appreciate that of course highest reductions occur in benchmarks with more accesses to memory as even they have the same WCD they have less memory

access and faster observed execution time (EOT)

			Round-Robin		Weighted	
Benchmark	*N_req*	*OET*	*WCD*	*WCET*	*WCD*	*WCET*
A	204.108	9.892.993	417	95.006,029	36.67	17.376,953
B	504.108	22.592.930	417	232.805,966	36.67	41.076,890
C	504.108	22.582.871	417	232.795,907	36.67	41.066,831
D	394.108	17.936.458	417	182.279,494	36.67	32.387,085
E	58.207	5.887.606	417	30.159,925	36.67	8.021.863
F	133.207	12.126.203	417	67.673,522	36.67	17.010.460
G	133.207	9.063.806	417	64.611.125	36.67	13.948.063
H	105.707	8.820.795	417	52.900.614	36.67	12.696.719

TABLE 6.2: WCET of applications running in a 4x4 2D Mesh

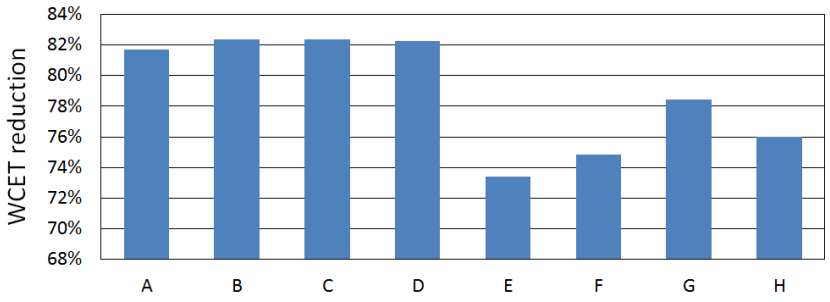


FIGURE 6.16: WCET reduction along all the benchmarks

WRR allocates the bandwidth in a fair way. This makes that the farthest nodes get more BW than with RR, but at the expense of a reduction in the BW of the nodes that are closer to the destination node. This effect is illustrated in Figure 6.17 that shows how with WRR the threads mapped to the nodes closer to memory achieve WCET values that are worse than the ones achieved with RR. However, the opposite trend is observed for the farthest nodes but with a more noticeable difference. The reason for this is the the pessimism that needs to be considered for farthest flows with RR is much higher making WCD values for these flows to grow exponentially. Interestingly, in the context of parallel applications, the execution time is determined by the slowest threads, making WRR to achieve significantly better performance for these applications.

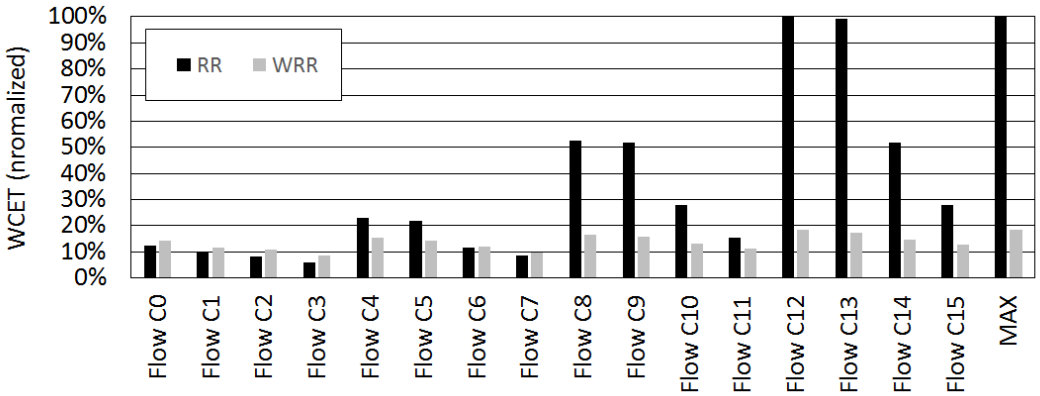


FIGURE 6.17: WCET normalized in a 4x4 mesh of benchmark A

Chapter 7

Conclusions and Future Work

In this thesis we have analyzed the potential of using heterogeneous bandwidth allocation to improve the WCET of applications executed in NoC-based multi- and manycore processors. In particular, we have analyzed how weighted arbitration schemes can be very useful to maximize the efficient utilization of shared resources. We have analyzed the impact of controlling the bandwidth allocation in two different scenarios. In the first scenario, we have shown that in processors with uniform access to memory, heterogeneous bandwidth allocation results useful to maximize the guaranteed throughput of parallel applications with heterogeneous threads. In a second scenario, we have shown that using weighted arbitration becomes crucial to equalize the uneven distribution of bandwidth in systems with non-uniform access to memory like processors using NoCs implementing a mesh topology.

In particular, our results show that weighted arbitration (WRR) provides bandwidth malleability with low cost and high flexibility. However, in the case of meshes, while weighted meshes can homogenize bandwidth allocation, they cannot mitigate the intrinsically variable core-to-memory latency since the distance from cores to memory determines a minimum access latency.

We also show that flexible bandwidth allocation (FBA) is beneficial, not only in equalizing the BW among all cores in non-uniform access to memory systems, but also in distributing the BW in the most convenient way regardless of whether the interconnection has unified memory access distance to memory or not.

In this thesis we have also analyzed the impact of weighted arbitration in BW allocation and latency in different scenarios and have shown how BW distribution effectiveness depends on parameters like the amount of traffic we have in the NoC,

the interconnection geometry, and how the different flows are routed in the topology, which are some of the most relevant parameters. We have identified and analyzed some FBA limitations caused by bubbles in the NoC pipeline that prevent WRR from achieving fully-balanced BW allocation in some cases (e.g. medium size and large meshes).

We have also proven that using WRR is specially beneficial to achieve homogeneous BW allocation in critical real time applications as we are able to achieve WCET estimates reductions between 73% and 83% when using WRR arbitration instead of RR.

Part of our future work consists in exploring the capabilities of heterogeneous bandwidth allocation by allowing the task scheduler decide how weights should be assigned according to the different requirements and features of each of the threads that are to be scheduled concurrently. Similarly, we foresee that integrating heterogeneous bandwidth allocation in the timing analysis can lead to significantly better WCET estimates, especially for parallel applications, since the thread-to-core allocation can be optimized together with the bandwidth allocation to each core so that the maximum execution time across threads is minimized.

Appendix A: Spinlock analysis

To analyze the impact of spinlock in the performance of parallel applications we have used specific PowerPC benchmarks with the characteristics shown in Table 4.3. It is important to underline that Benchmarks Bench1, Bench2 and Bench3 have the same properties (same amount of computation, load and store instructions) and only varies the size of each benchmark. In this part, we analyze the performance impact that have applications when finish their execution in the remaining execution of others. For this reason, we have implemented benchmarks with similar properties but different instruction counts. Bench0 is the longest benchmark that always remains executing during the experiments. We assume that Bench0 is mapped to C_0, Bench1 to C_1 and so on and so forth, and the underlying NoC is a tree, hence with balanced BW across cores.

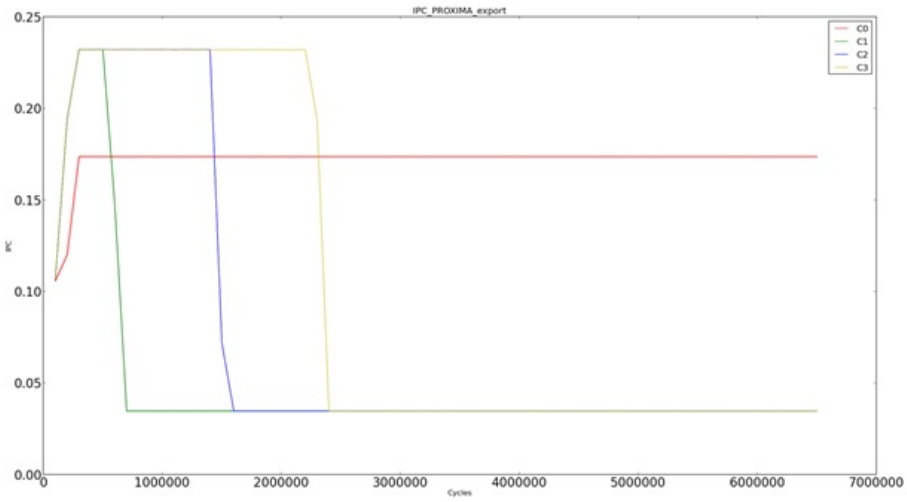


FIGURE 7.1: IPC evolution along time results

Figure 7.1 shows in the x-axis the Instructions Per Cycle (IPC) of the 4 cores during their execution. It can be observed that, initially, all cores have a growing slope which corresponds to the initialization part common in all cores. After that, C_1

(green) finishes and immediately starts executing the spinlocks part (very low IPC as the spinlock is equivalent as a 100% load miss benchmark). The same behavior can be observed in C_2 and C_3 some time later. On the other hand, C_0 IPC remains unaffected by the performance drop of other cores. The IPC of C_0 is worse than that of the other cores because C_1, C_2 and C_3 are executing stores, and the write buffer can mitigate the latency to serve stores. That is a better situation than the one is facing C_0, since Bench0 in C_0 executes loads that stall its pipeline on every access (i.e. the pipeline is stalled until the loaded data is received). One would expect that, upon the finalization of some benchmarks in some of the cores, the IPC of C_0 increased. However, main memory has a relatively large latency so, even though the benchmarks only have 5% of store instructions (write through policy), this access rate is enough to saturate the memory queue. That means that, whenever a core wants to access memory, even if it experiences no NoC contention, the core request gets stalled in the memory queue because it is full. C_0 can only send the next memory request when the previous one has been served and, since they get stalled in memory, it has to wait long until it is served. Whenever a core ends its execution (C_1, C_2 or C_3), it stops generating store requests and, instead, generates load requests due to spinlock. Again, these requests do not alter the behavior of C_0, which keeps finding the memory queue full, thus experiencing the same (very high) contention in the memory access.

In order to reduce the amount of times that a core checks the spinlock variable when finishes its execution, thus reducing the memory contention caused, we have explored a simple solution: a loop with division operations has been added in the spinlock loop.

In Figure 7.2 we see the effect that produces a core when it ends its execution using 10 divisions between spinlock checks. For instance, when C_1 ends the execution (green), its IPC decreases a little because starts checking the spinlock variable, but thanks to the division loop, the IPC is higher than in the baseline spinlock situation shown in Figure 7.1 when the IPC of C_1, C_2 and C_3 was under 0.05. The same behavior can be observed when C_2 and C_3 end the execution. When one of these cores ends its execution, the IPC of the remaining cores grow (the other cores now have more available memory bandwidth than before). This behavior is also shown in the

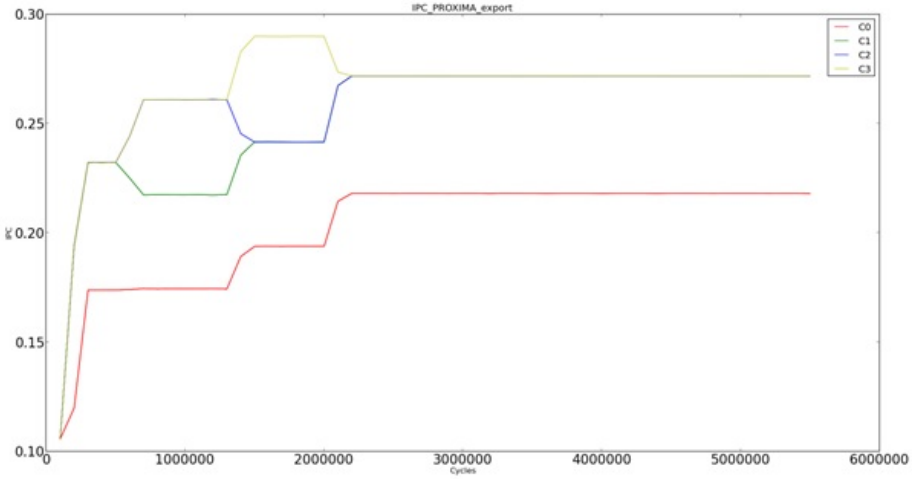


FIGURE 7.2: IPC evolution along time with spinlock reduction of x10

IPC of C_0 (red) that grows in three steps. Although the first one is pretty small due to the still high contention in memory, the other steps are more noticeable.

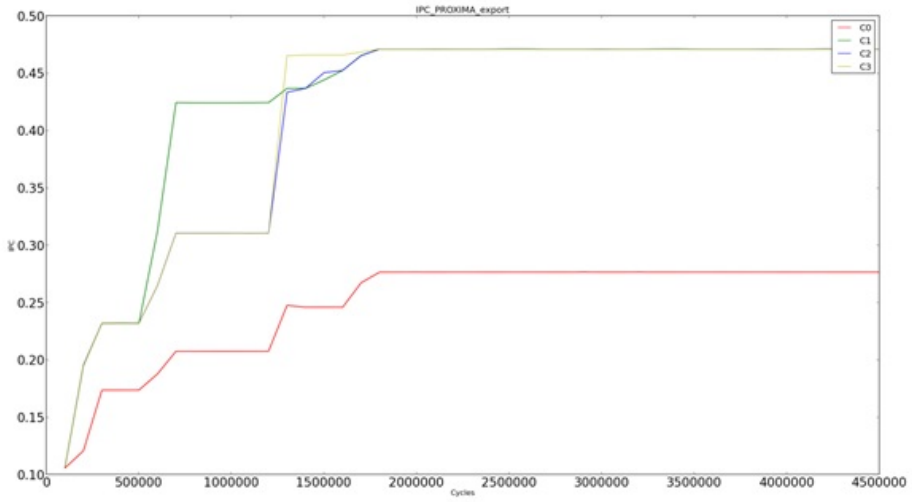


FIGURE 7.3: IPC evolution along time with spinlock reduction of x100

Figure 7.3 shows, as the previous plot, the IPC improvement in the cores that continue executing tasks when one of the other cores ends, but this time using 100 divisions in-between spinlocks instead of 10. The IPC impact in other cores is higher upon the finalization of any core due to the increased time in-between spinlock checks (the core that ends accesses fewer times the spinlock variable).

In Figure 7.4 we show the same IPC evolution when different cores end their execution using 1000 divisions instead of 100. Again, reducing the number of spinlock

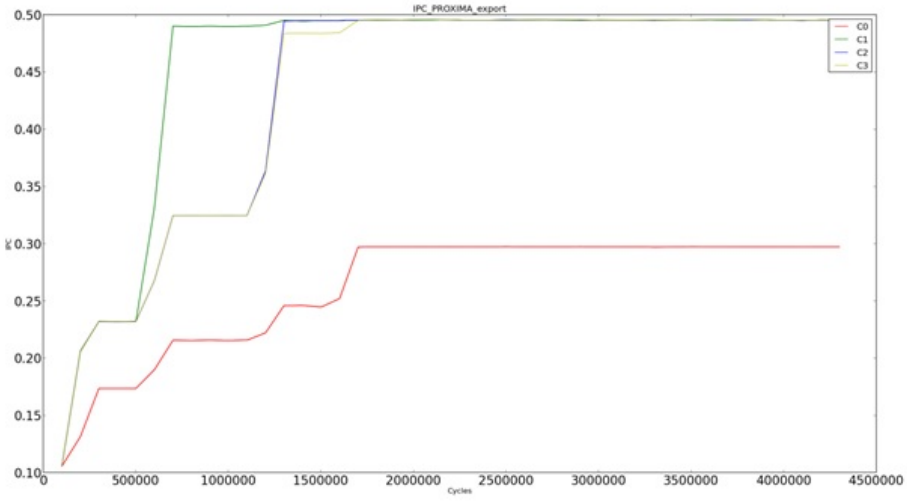


FIGURE 7.4: IPC evolution along time with spinlock reduction of x1000

checks, reduces memory contention and C_0 speedup is higher upon the finalization of any other core than in the cases with higher frequency spinlock checks (100 divisions, 10 divisions, or no divisions).

Appendix B: Arbitration pathological cases

For some settings we have observed some pathological arbitration scenarios caused by the systematic bad alignment of request w.r.t. the arbitration window. Figure 7.5 shows, for a 3x3 NoC, a pathological behavior that occurred even when we used the correct weights to obtain homogeneous BW distribution (e.g. 1/8 of the BW in each core). We observe imbalance between messages sent from C_0 and C_1 compared with the rest of cores in the 3x3 mesh. Instead of sending 2500 messages each core, we see that C_0 and C_1 send 2942 messages (more than expected) and cores from C_3 to C_8 send 2353 messages (less than expected). That means that we have an imbalance problem in R_2, as the imbalance is between 2 groups: flows from cores that come from X+ port (C_0 and C_1) and flows from cores that com from Y- port (C_3 to C_8).

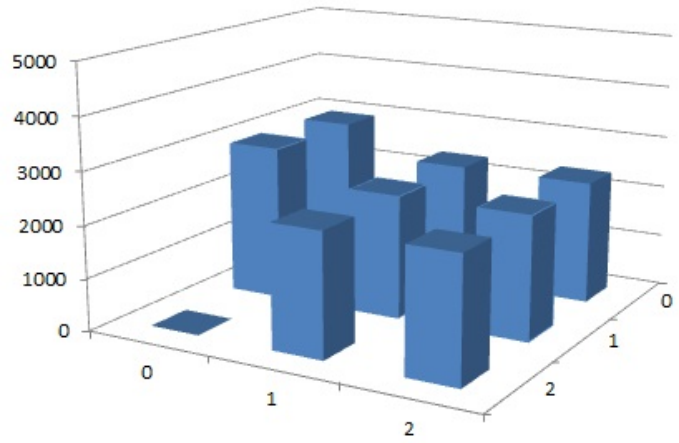
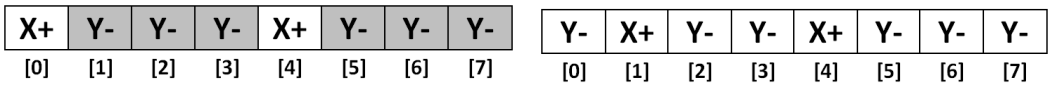


FIGURE 7.5: Messages sent per NIC in a 3x3mesh XY routing 1.0 IR and WRR arbitration

We have observed that the root of such imbalance is at the implementation of WRR. In this case, we were using for the memory output port in R_2 and X+ port (that traverse C_0 and C_1 flows) and Y- port (that traverse the remaining core flows).



(A) Original arbitration window (B) Permuted arbitration window

FIGURE 7.6: Arbitration windows in R_2

If we analyze in detail the case of R_2 arbitration, we see that, to provide a weighted arbitration window with homogeneous BW allocation, we have to give 2/8 of the BW to X+ port and 6/8 to Y-. So one could initially think that the arbitration window that provides the most uniform distance between arbitrations is the one shown in Figure 7.6a. After the execution, we have realized that this window, instead of providing 5000 messages to X+ port (2/8 = 25%) and 15000 messages to Y- (6/8 = 75%) was providing 5882 messages (≈2/7) and 14118 messages (≈5/7) to each port respectively.

Observing the real arbitration that the arbitration window is performing, we have identified a systematic pathological behavior occurring. In particular, a relevant number of times (once in every full arbitration window), arbitration is not given to Y- even if it has priority since there is no ready petition from Y-. Instead, it is given to the next X+ port (that is the following port that has a ready petition as X+). This is caused by how the arbitration window distributes the ports priority. Two consecutive requests of X+ have to wait in the worst case 3 arbitrations of Y- port. Instead, Y- may be granted the arbitration consecutively several times (up to 3 consecutive times) and 1 every 3 Y- requests has to wait for one arbitration round of X+. This means that Y- is more sensitive to routing pipeline bubbles. In particular, there is a bubble that makes that the 6 requests of the Y- port arrive in 7 cycles, but having the bubble exactly after 2 requests. Hence, in one of the two groups of 3 consecutive Y- arbitrations, the third arbitration is lost. Therefore, every 8 arbitrations (a full window), 2 are effectively given to X+, 5 are effectively given to Y- and 1 is lost (the grant is given to the other port, X+ in our case).

We have also observed that this imbalance keeps appearing independently of the injection rates employed, as long as they are sufficient to saturate the NoC, hence above $1/N$ per core.

In order to improve BW distribution with WRR with high IR, we have modified the permutations in the arbitration window that we were using (see Figure 7.6b). We have observed that, for example, for the new arbitration window shown in Figure 7.6b, we achieve the homogeneous BW allocation in cores that we were expecting. In particular, such new permutation allows aligning the bubble in Y- with an arbitration grant to X+, so that no Y- arbitration is lost.

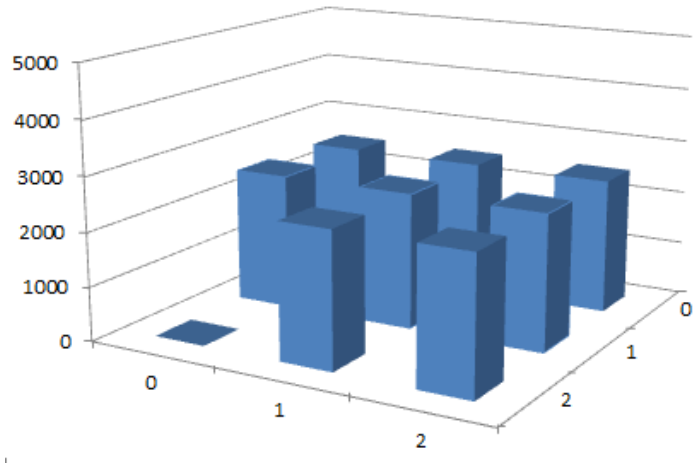
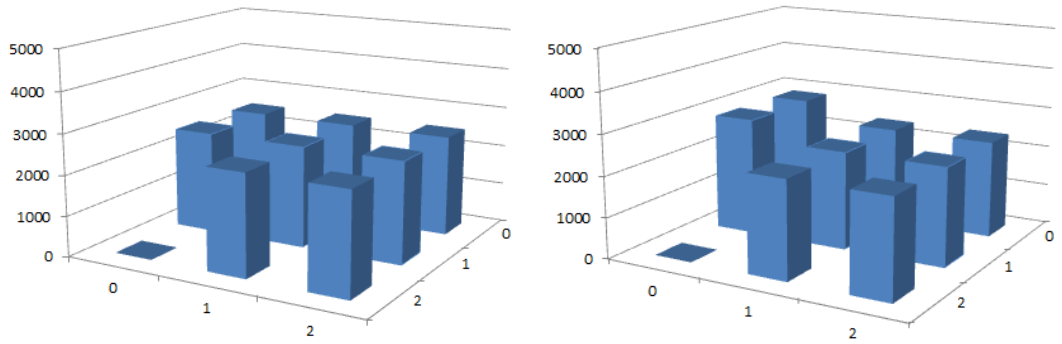


FIGURE 7.7: Messages sent per NIC in a 3x3mesh XY routing 1.0 IR with Figure 7.6b arbitration window

Once adopted the new permuted window, we do not observe any imbalance on the number of messages sent (see Figure 7.7) when we have IR greater than 0.4. However, with this new permuted window we observe that it starts losing effectiveness earlier than the original window that provided imbalance in all cases. This effectiveness loss of the arbiter occurs due to the reduction in the traffic in the mesh, which increases the likelihood of missing arbitration rounds.



(A) Injection Rate 0.4 (B) Injection Rate 0.3

FIGURE 7.8: Arbiter effectiveness lost due to window alignment

Figure 7.8 shows the permuted window arbitration effectiveness loss between 0.4 and 0.3 IR. As indicated, for lower IR, some imbalance occurs because Y- port is more likely not to have any pending request in some arbitration rounds, which ultimately leads to a non-fully-balanced BW allocation across cores.

List of Abbreviations

BW	**B**and**W**idth
COTS	**C**omertial **O**ff **T**he **S**helf
CRTES	**C**ritical **R**eal **T**ime **E**mbedded **S**ystems
DMA	**D**irect **M**emory **A**ccess
FBA	**F**lexible **B**andwidth **A**llocation
FLITS	**FL**ow control un**IT**s
IR	**I**njection **R**ate
MBTA	**M**easurement-**B**ased **T**iming **A**nalysis
NIC	**N**etwork **I**nterface **C**ontroller
NUMA	**N**on-**U**niform **M**emory **A**ccess
NoC	**N**etwork **o**n **C**hip
PME	**P**rocessor or **M**emory **E**lement
RR	**R**ound **R**obin
STA	**S**tatic **T**iming**A**nalysis
TDMA	**T**ime **D**ivision and **M**ultiplexed **A**ccess
UMA	**U**niform **M**emory **A**ccess
V&V	**V**alidation and and **V**erification
WCD	**W**orst **C**ontention **D**elay
WCET	**W**orst **C**ase **E**xecution **T**ime
WCTT	**W**orst **C**ase **T**raversal **T**ime
wNoC	**w**ormhole **N**etwork **o**n **C**hip
WRR	**W**eighted **R**ound **R**obin
ZLL	**Z**ero **L**oad **L**atency

Bibliography

[1] *6th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2000), 2-6 April 2000, Eilat, Israel*. IEEE Computer Society.

[2] *NanoC: NaNoC design platform. http://www.nanoc-project.eu*.

[3] Precision Timed Machines. http://chess.eecs.berkeley.edu/pret.

[4] J. Abella et al. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *SIES*, 2015.

[5] A.Psarras, et. al. Phase-noc: Tdm scheduling at the virtual-channel level for efficient network traffic isolation. DATE 2015.

[6] L. Benini et al. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *DATE*, 2012.

[7] Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor - LEON4-NGMP-DRAFT - Data Sheet and Users Manual*, 2011.

[8] D. Dasari, et al. Noc contention analysis using a branch-and-prune algorithm. *ACM Trans. Embed. Comput. Syst.*, 13(3s), March 2014.

[9] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654 – 1663, 2013. 2013 International Conference on Computational Science.

[10] José Flich and Davide Bertozzi, editors. *Designing network on-chip architectures in the nanoscale era*. Chapman & Hall/CRC computational science series. Chapman and Hall/CRC, 2011.

[11] GENESYS. GENeric Embedded SYStem Platform. http://www.genesys-platform.eu.

[12] K. Goossens, et al. Aethereal network on chip: concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 2005.

[13] K. Goossens, et. al. Virtual execution platforms for mixed-time-criticality systems: The compsoc architecture and design flow. *SIGBED Rev.*, 10(3):23–34, October 2013.

[14] H. M. G. Wassel, et. al. Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip. *SIGARCH Comput. Archit. News*, 41(3):583–594, June 2013.

[15] IBM. Power isa version 2.07, 2013.

[16] Intel Corporation. *Intel's Teraflops Research Chip. Advancing multi-core technology into the tera-scale era*.

[17] H. Kopetz and G. Bauer. The time-triggered architecture. *Proc. of the IEEE*, 91(1):112–126, 2003.

[18] T. Kranich and M. Berekovic. Noc switch with credit based guaranteed service support qualified for GALS systems. In *DSD*, 2010.

[19] Sunggu Lee. Real-time wormhole channels. *Journal Of Parallel And Distributed Computing*, 63:299–311, 2003.

[20] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch. The nostrum backbone-a communication protocol stack for networks on chip. In *IEEE VLSI Design*, pages 693–696, 2004.

[21] R. Obermaisser, et al. The time-triggered system-on-a-chip architecture. In *ISIE*, 2008.

[22] M. Panic, C. Hernandez, J. Abella, A. Roca, E. Quiñones, and F. J. Cazorla. Improving performance guarantees in wormhole mesh noc designs. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1485–1488, March 2016.

[23] M. Panic, C. Hernandez, E. Quinones, J. Abella, and F. J. Cazorla. Modeling high-performance wormhole nocs for critical real-time embedded systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016.

[24] Miloš Panić, Eduardo Quiñones, Pavel G. Zaykov, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Parallel many-core avionics systems. In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, pages 26:1–26:10, New York, NY, USA, 2014. ACM.

[25] C. Park, R. Badeau, L. Biro, J. Chang, T. Singh, J. Vash, B. Wang, and T. Wang. A 1.2 tb/s on-chip ring interconnect for 45nm 8-core enterprise xeon processor. In *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*, pages 180–181, Feb 2010.

[26] parMERASA. *EU-FP7 Project:http://www.parmerasa.eu/*.

[27] Y. Qian, Z. Lu, and W. Dou. Analysis of worst-case delay bounds for best-effort communication in wormhole networks on chip. In *IEEE/ACM NoCS*, 2009.

[28] D. Rahmati, et al. Computing accurate performance bounds for best effort networks-on-chip. *IEEE Transactions on Computers*, 2013.

[29] E. A. Rambo and R. Ernst. Worst-case communication time analysis of networks-on-chip with shared virtual channels. DATE, 2015.

[30] J. Rattner. *Single-chip Cloud Computer: An experimental many-core processor from Intel Labs*.

[31] Govindan Ravindran and Michael Stumm. A performance comparison of hierarchical ring- and mesh- connected multiprocessor networks. *HPCA*, 1997.

[32] A. Roca, et al. Enabling high-performance crossbars through a floorplan-aware design. In *ICPP*, 2012.

[33] Erno Salminen, Tero Kangas, Vesa Lahtinen, Jouni Riihimäki, Kimmo Kusilinna, and Timo D. Hämäläinen. Benchmarking mesh and hierarchical bus networks in system-on-chip context. *Journal of Systems Architecture*, 2007.

[34] M. Schoeberl et al. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *IEEE/ACM NoCS*, 2012.

[35] Zheng Shi and A. Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *NoCS*, 2008.

[36] Zheng Shi and A. Burns. Real-time communication analysis with a priority share policy in on-chip networks. In *ECRTS*, 2009.

[37] M. Slijepcevic, M. Fernadez, C. Hernandez, J. Abella, E. Quiñones, and F. J. Cazorla. ptnoc: Probabilistically time-analyzable tree-based noc for mixed-criticality systems. In *2016 Euromicro Conference on Digital System Design (DSD)*, pages 404–412, Aug 2016.

[38] SoCLib. -, 2003-2012. http://www.soclib.fr/trac/dev.

[39] J. Sparsoe. Design of networks-on-chip for real-time multi-processor systems-on-chip. In *ACSD*, 2012.

[40] Tilera. *TILE-Gx Processors Family http://www.tilera.com/products/TILE-Gx.php*.

[41] F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quiñones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, T. Vardanega, and F. J. Cazorla. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 241–248, June 2013.

[42] R. Wilhelm et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7:1–53, May 2008.

[43] Mei Yang, S. Q. Zheng, B. Bhagyavati, and S. Kurkovskyt. Programmable weighted arbiters for constructing switch schedulers. In *High Performance Switching and Routing (HPSR). Workshop on*, pages 203–206, 2004.