Hardware Extensions for a Timing-Predictable Many-Core Processor

Dissertation

for the degree
Doctor of Engineering (Dr.-Ing.)

submitted to the
Department of Computer Science

University of Augsburg

by

Martin Frieb

September 13th, 2019

Thesis titled: Hardware Extensions for a Timing-Predictable Many-Core Processor

Supervisor: **Prof. Dr. rer. nat. Theo Ungerer**, Department of Computer Science
University of Augsburg, Germany

Advisor: **Prof. Dr.-Ing. Rudi Knorr**, Department of Computer Science
University of Augsburg, Germany

Date of defense: November 19th, 2019

Abstract

The requirements for today's embedded hard real-time systems are high: They should deliver high performance, be energy-efficient and always react in time. This leads to the use of processors with several cores. However, when the cores are connected via a shared memory, static timing analysis suffers from high pessimism. We see distributed memory many-core processors as a solution where cores communicate via messages. One of them is the *Reduced Complexity Many-Core* (RC/MC) architecture [MFSU17]. It was developed with the goal of high timing predictability.

In our thesis, we present an approach to estimate the *Worst-Case Execution Time* (WCET) of programs running on this platform. Furthermore, we extend the RC/MC to improve its timing predictability and its worst-case performance. Our first step is the introduction of `ready` synchronization, which avoids buffer overflows. Second, we design hardware support for broadcasts and multicasts. Third, the RC/MC is extended with hardware supported barriers.

Each of these techniques is evaluated for its impact. We carry out timing analyses of the hardware operations for broadcasts/multicasts and barriers and compare them with their variants without hardware support. Finally, we present three case studies, where we analyze benchmarks taken from the NAS parallel benchmark suite to evaluate the worst-case performance of our extensions in the context of real use cases.

Never give up your dream
– Timo's life motto

Acknowledgements

This thesis would not have been possible without the support of many people:

First and foremost, I would like to express my sincere gratitude to my supervisor **Prof. Dr. Theo Ungerer**, professor of the *Chair of Systems and Networking*. He has been very supportive at all times, I thank him for encouraging my research and for allowing me to grow as a research scientist. I have learned a lot from him, his constructive comments and suggestions were always a good guidance for me. I am also very grateful that he made a research trip to Toulouse possible, where I integrated RISC-V support into the timing analysis tool OTAWA.

I also thank my advisor **Prof. Dr. Rudi Knorr** and **Prof. Dr. Sebastian Altmeyer** for accepting to be an examiner for my thesis.

My sincere thanks go to **all my colleagues** at the University of Augsburg, for many valuable discussions, their support and comments on my work, especially **Alexander Stegmeier** and **Dr. Jörg Mische**. Furthermore, I thank **my students** who helped with the implementation of my ideas and the timing analysis during their bachelor, master, practical or project modules. Their studies are referenced throughout the thesis.

Special thanks go to **Prof. Dr. Hugues Cassé**, **Dr. Wei-Tsun "Willie" Sun** and **Dr. Haluk Ozaktas** from the *Institut de Recherche en Informatique de Toulouse* (IRIT) at the Université Paul Sabattier who develop the timing analysis tool OTAWA and gave me support at using and extending it. More special thanks go to all members of the **TACLe (Timing Analysis on Code Level) network** for inspiring me on their summer school, PhD meeting and other meetings.

Last but not least I thank my parents **Roswitha** and **Erich** for their education and support during my studies as well as my sister **Birgit** and my brother **Rainer**. Furthermore, I thank my **friends** for their continuous support, especially **Dr. Ingo Blechschmidt**, **Dr. Michael Hartmann** and **Paul Colin Hennig**.

Martin Frieb Augsburg, September 13th, 2019

In beloved memory of

Timo Collenberg

⋆ September 21st, 1990

† April 21st, 2011

Contents

1 Introduction

Embedded real-time systems have become increasingly important devices. They are omnipresent in cars, trains or medical devices. They have to respect real-time constrains, i.e. computation results have to be available in time. On the one hand, considering *soft* real-time systems, a result gradually looses its relevance the later it becomes available, e.g. satellite-based navigation will still be a good guide when the result is one second too late, but not after five minutes. On the other hand, at *hard* real-time systems late results could immediately cause major harm or even death of involved people, e.g. an airbag not responding in time or an airplane crash due to late response of the autopoilot system [WEA+08]. Therefore, the *worst-case execution time (WCET)* of the involved programs has to be estimated to be sure that deadlines will not be missed. However, techniques in modern processors like caches, branch prediction or out-of-order execution make it hard to impossible to carry out a WCET analysis.

Moreover, just like with all other computing devices, there is an ever-growing demand for more computational power at embedded real-time systems. In the field of desktop and server computers and even smartphones, this demand is satisfied by adding more cores [Sut05]. This *multi-core revolution* slowly reaches the embedded real-time field, e.g. in the form of multi-core *Electronic Control Units (ECUs)* in cars [AUT14, KQnBS15]. Thereby, typically more cores are added, but shared memory is kept, because software engineers are used to develop shared memory programs. However, this way is very challenging, especially when timing constraints have to be met: The presence of additional cores itself increases WCET estimates, because it has to be assumed that other cores access shared memory

before the own memory access is processed [RBS+10, FJO+16]. Further problems are high costs for cache coherence, sequentialization when accessing shared resources and waiting times for critical sections [ADG16]. This leads to a loss of parallel computation power and decreasing efficiency per core for increasing core numbers.

Most of these problems also have to be handled in high-performance computing. There, performance is gained by employing many cores and connecting them via a *network-on-chip (NoC)* [HJK+00, KJS+02], e.g. in the Intel Xeon Phi [SGC+16] co-processor. A NoC enables that several components (e.g. cores or memory) on the chip can interact with each other simultaneously by applying proven concepts from large networks on the processor. They have several advantages like good scalability and high energy efficiency [BM06, AIS09]. One of the key factors when employing NoCs is that components on the chip have less side effects on each other. This is also beneficial in the real-time field, especially for timing analyzability.

Therefore, we also follow this route with the *Reduced Complexity Many-Core (RC/MC)* architecture [MFSU17]. It does not have any shared memory and its programming model is completely different to shared memory multi-cores. The basic idea of the RC/MC is that cores work in total isolation. Thereby, they communicate via explicit messages over a predictable NoC. Due to the high degree of isolation and simplicity of cores, a WCET analysis of the code executed on the cores can be carried out utilizing the established methods [WEA+08]. For the total WCET estimation of a parallel application, the outcome has to be combined with a WCET analysis of the network traffic. When more cores are added, only the latencies of the NoC increase a little – there is no effect on the WCET estimates of the cores. Therefore, we see this approach as a scalable solution, appropriate for real-time many-core processors and delivering the computational power and timing analyzability needed for future embedded real-time systems.

However, while the basic ideas are trendsetting, the RC/MC as presented by Mische et al. [MFSU17] requires some extensions to improve its timing predictability and worst-case performance. We see our focus at improving its *network interface (NI)*, which connects the cores with the NoC and contains the message send and receive buffers. In our thesis, we will integrate several hardware extensions at the NI and evaluate how they influence timing predictability and worst-case performance.

1.1. Contribution and Structure of this Thesis

The concepts elaborated in our thesis are generally platform independent. For their realization, we take the RC/MC architecture due to its simple design and because it was originally designed to be utilized for real-time applications [MMU11, MFSU17]. We will describe its background in Chapter 3 after having a look on related work in Chapter 2. Chapter 3 also includes our concept for the timing

analysis of parallel applications on many-core architectures. Afterwards, we will introduce our real-time flow control mechanism called `ready` synchronization in Chapter 4. It avoids buffer overflows while various cores communicate with each other. The next two Chapters 5 and 6 will focus on improvements of the One-to-All schedule, which manages the conflict-free and timing-predictable delivery of messages in the NoC. However, it has some drawbacks. To overcome them, we will elaborate and implement a hardware-supported broadcast/multicast operation in Chapter 5 and hardware-supported barriers in Chapter 6. Thereby, a timing analysis of our new and established broadcast/multicast and barrier operations will be carried out at the evaluation of the corresponding chapters. Then, we evaluate the impact of our hardware extensions in three case studies in Chapter 7. Finally, we conclude our thesis in Chapter 8 and give an outlook to future work.

2
Related Work

We will present related work on each hardware extension in the corresponding Sections 4.2, 5.2 and 6.2. Thus, our focus here is on other many-core architectures: First, we have a look on many-core architectures with directly connected cores in Section 2.1. They provide an own NI for each core and they are designed in a highly homogeneous way: All cores behave the same way and have the same latencies when communicating with other cores. Second, there are many-core architectures where two or more cores share one NI and sometimes even a small shared memory installed between them. It allows them to exchange data within the group very fast, but makes the architecture more complex. Moreover, communication latencies vary, depending where the communication partner is installed. We will present some architectures with cores organized in groups in Section 2.2.

2.1. Many-core Architectures with Directly Connected Cores

Generally, today's many-core architectures may be seen as successors of the transputer [WS85]. Its basic idea is to have *very powerful computer systems containing many processing elements* [WS85]. Whitby-Strevens suggests a workstation with transputers each for user interaction, execution of applications and controlling disks and further devices. Another idea is a board with 128 transputers for high performance database searching. From the architectural view, a transputer is what we call core in a many-core processor today. It consists of a processing element, memory and communication links and is programmed in Occam [May83]. A system is under-

stood as collection of transputers that are connected with each other. The concept to use this transputer system is to carry out local computations on local data and processes can communicate via message passing. Thereby, *channels* are the concept for communication provided by Occam. They are working in a synchronized way, i.e. transputers have to wait for each other when transmitting data. This avoids data loss and allows to have only very small buffers. On the hardware level, two transputers are connected via two one-directional signal lines. Since several links are available, several transputers may be connected with each other. This implies even heterogeneous transputers – e.g. different word lengths were considered throughout the design of the transputer concept. Its designers were also already aware of the bottleneck at off-chip communication. Therefore, only 25% of the chip area of a transputer are for the processing element and most of rest for on-chip memory. Many of the transputer ideas have found their way into modern many-core designs. Instead of channels, we have one or several NoCs today. They not only connect specific cores with each other, but allow communication between arbitrary cores. However, the basic idea to have simple cores and gain performance through high parallelism can be seen throughout the many-core architectures presented in this chapter.

One of them is the Epiphany many-core architecture, which was developed by Olofsson [ONUA14, Olo16, Olo17]. The Epiphany-IV processor features 64 cores with 32 KB of local scratchpad memory each [ONUA14]. They are directly connected via three independent 2D scalable mesh NoCs. Instead of explicit messages distributed shared memory is utilized for communication between the cores. Its successor Epiphany-V is a large many-core processor with 1024 cores and 64 MB of distributed on-chip memory, i.e. 62,5 KB per core [Olo16, Olo17]. The Epiphany processors target smartphone applications, supercomputers and floating-point acceleration in embedded systems. However, the caches do not have an explicit hierarchy and loads and stores are allowed to access data inter-core and even off-chip. Furthermore, there are high variations at traversal times when sending messages between cores. Altogether, this makes timing analysis impossible and this architecture unsuitable for real-time systems.

A homogeneous approach targeting embedded real-time systems is the T-CREST architecture [SAA+15]. Basically, it consists of a lot of Patmos cores [SSP+11] connected to two NoCs: One for data exchange between the cores and a *time-division multiplexing (TDM)* tree to connect the cores to a global shared memory. The Patmos core is a dual-issue 5-stage pipelined VLIW processor with caches (stack cache, data cache, method cache) and a scratchpad memory. For data exchange between the cores implicit message passing in the form of *direct memory access (DMA)* driven block transfers is carried out via the Argo NoC [KS14, KSS+16]. Thereby, the All-to-All schedule [SBSK12] is employed, which we explain in Subsection 3.2.2 and will

be used for comparison in the evaluation Sections 5.8 and 6.8 and at the case studies in Chapter 7. For the connection of the cores to the shared memory arbitration takes place in the NoC and in the memory controller.

Altogether, the T-CREST architecture is comparable to the RC/MC architecture. However, we use different cores without caches, only having a scratchpad memory. Only one NoC is present at the RC/MC and we employ PaterNoster [MU12, MU14] instead of Argo. Moreover, we do not have a shared memory or DMA.

The TILE64 processor developed by Tilera consists of 8x8 = 64 *tiles* [ABB+07, BEA+08]. Each tile consists of a core, several caches and a network switch connecting the tile to the different NoCs. A tile core is 3-way VLIW and has 8 KB of L1 data cache and another 8 KB of L1 instruction cache. The 2-way 64 KB L2 cache may be shared among tiles to provide up to 4 MB of L3 cache. All tiles are connected via five 2D mesh NoCs called iMesh: A static network for low-latency communication and four dynamic networks (dynamic network , user dynamic network, memory dynamic network, I/O dynamic network). As their names suggest, each of these NoCs is intended for a specific function. Moreover, adjacent tiles can communicate on register-level within a two-cycle latency. The TILE64 processor focuses on high performance in networking and digital media (e.g. video encoding). It is not capable to be used for hard real-time applications.

2.2. Many-core Architectures with Cores organized in Groups

At another class of many-core processors not every core is connected to the NoC. Instead, several cores form a group sharing several ressources (e.g. the NI or some on-chip memory).

An early example is the Intel *Single-chip Cloud Computer (SCC)* [Rat10]. It consists of 24 tiles with two Pentium cores each and a small *message passing buffer* of 16 KB. The Pentium cores have 32 KB L1 cache (16 KB data cache and 16 KB instruction cache) and 256 KB unified L2 cache [vdWMH11]. Together, the tiles form a non-coherent distributed shared memory. Although the programming model relies on message passing, no explicit messages are utilized for communication. The commercial successor of the Intel SCC is the Xeon Phi [SGC+16] co-processor. It features 36 tiles with 2 Atom cores, 2 vision processing units and 1 MB L2 cache, which is kept coherent via the MESIF protocol [GH04]. The Atom cores are out-of-order with 4-way simultaneous multithreading. A 2D mesh NoC connects the tiles. Both the Intel SCC and the Intel Xeon Phi are targeting high-performance computing and are not suitable for real-time systems.

A positive example for an architecture with good timing predictability is the *Composable and Predictable Multi-Processor System on Chip (CoMPSoC)* [HGBH09,

GKN+17]. At its development, predictability and composability were in the focus. Small processor tiles with several cores are connected via the real-time NoC dAElite [SMG14]. Predictability is achieved by utilizing real-time components everywhere, composability through strict isolation of applications. Both intentions are also found in the RC/MC processor. However, we use another NoC and *instruction set architecture (ISA)* as well as strict message passing, while CoMPSoC is based on shared memory organized in tiles.

Another platform targeting embedded systems is the Kalray *Massively Parallel Processor Array (MPPA)* [dDAB+13]. At the first MPPA *Andey*, 16 clusters with 16 processing cores each are installed. Thereby, each core has its own L1 cache and has access to a cluster-wide shared memory, which it uses together with the other 15 cores on this cluster. This cluster-wide shared memory may also be configured to be used as private scratchpads for the 16 cores on the cluster. Inter-cluster communication is organized via a dual 2D torus NoC [dD15]. It guarantees predictable latencies for all data transfers. A timing analysis for the Kalray NoC architecture was carried out by Ayed et al. [AESF16]. For communication between the cores/clusters, there is no explicit message passing – the Kalray MPPA can be programmed like a shared memory architecture; implicit message passing is possible via remote DMA to other clusters' memory [dD16, VLKJ17]. Furthermore, a high level programming model is available in the form of the data flow language ΣC [GSLD11]. Thereby, programmers only express data dependencies and do not need to care about communication [VLKJ17]. Later Kalray MPPA processors *Bostan* and *Coolidge* always follow the same basic principles, with varying numbers of clusters and cores [dD15]. Because of the complex architecture with caches and clusters, a static timing analysis is challenging. Moreover, due to the cluster concept, this architecture is highly placement dependent, while the RC/MC allows placement of tasks at any core.

3

The RC/MC Processor Architecture

Overview. After giving an overview on related work in the previous chapter, we now describe the basic concepts of the RC/MC in Section 3.1. Furthermore, we give an overview how communication within the NoC is organized (Section 3.2) and present the programming model for our platform (Section 3.3). Then, we illustrate the underlying instruction set architecture and how we extended it in Section 3.4. In Section 3.5, we introduce the hardware model of our platform and the simulators we utilize for our experiments. Finally, in Section 3.6, we show how timing analysis for our platform works, integrating the worst-case times of local computation as well as communication on the chip.

3.1. Basic Concept

Our processor is based on four assumptions for a hard real-time capable many-core processor by Metzlaff et al. [MMU11]:

1. *A small and simple core is the best building block for a real-time capable many-core.*
2. *Distributed memory with the option to access the memory of other nodes provides a predictable memory access and enough bandwidth for embedded applications.*
3. *A statically scheduled mesh network for hard real-time tasks targets a predictable timing and a high utilization.*

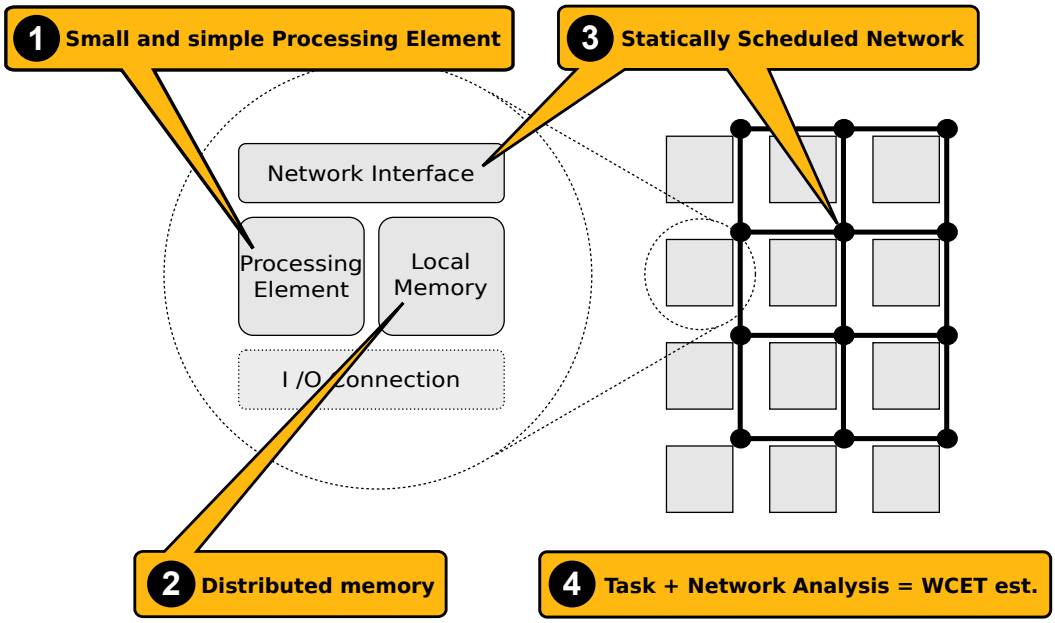


Figure 3.1.: The ideas behind the RC/MC are small and simple processing elements (1) with distributed memory (2), which are connected via a statically scheduled NoC (3). This allows to estimate a WCET by combining a task analysis with a network analysis (4) [MMU11, MFSU17].

4. *A tight WCET analysis of the system can be reduced to an independent task analysis per core and an analysis of the network communication.*

We actualized a processor following these assumptions, the *Reduced Complexity Many-Core* (RC/MC) processor [MFSU17]. It is a many-core processor for hard real-time systems, which delivers good timing predictability due to *reduced complexity (RC)* and high performance via *many cores (MC)*. In Figure 3.1, we give an overview over our RC/MC processor:

Following the first assumption, it employs simple *processing elements (PEs)* on each node[1] – each of them implements a classical single-issue in-order 5-stage pipeline utilizing the RISC-V 64 bit ISA [WAE17]. Thereby, no speculative components are installed at the nodes – neither caches nor branch prediction. All nodes are single-threaded and intended to execute only one software thread. I/O or additional DRAM is connected to single nodes. When something is needed from or to be sent there, a request to the corresponding "DRAM node" has to be sent. Interrupts are also handled by dedicated nodes. Besides external components/events, the architecture is considered to be completely homogeneous: All nodes have the same

[1] Because we consider a many-core architecture, we avoid the term *core*. Instead, the term *processing element (PE)* refers to each 5-stage pipeline including register set. The PE together with its local memory (scratchpad) and network interface (and I/O connection) is referred to as *node*.

structure and capabilities as well as the same access and waiting times. This is beneficial for timing analyzability, because no special cases have to be considered.

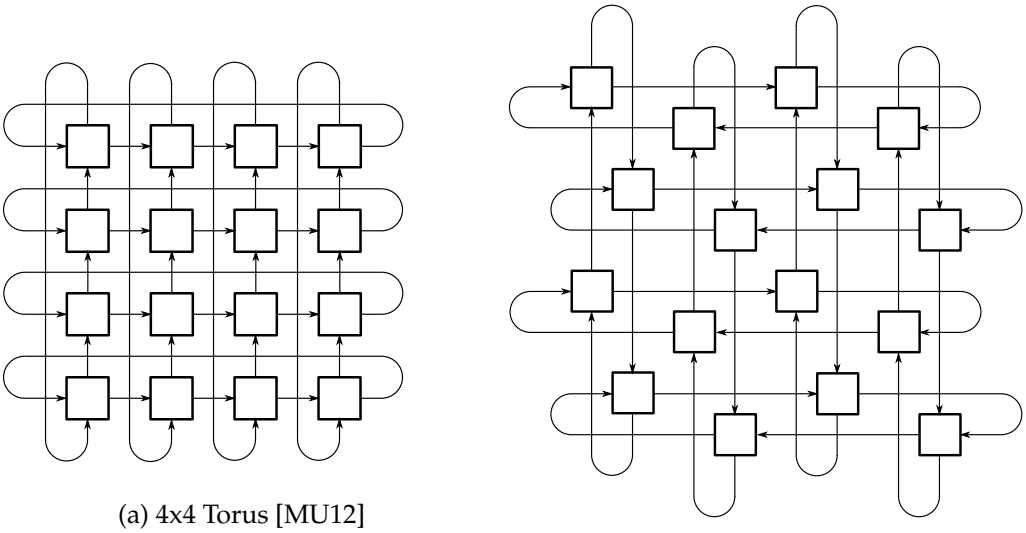
As proposed by the second assumption, there is no global shared memory, only distributed core-local memory in the form of scratchpads. In our prototype, each scratchpad is connected to its PE like a L1 cache, having a memory access time of only one cycle – there is no latency. Thus, load/store instructions pass the pipeline within five cycles (1 cycle for each stage). However, the downside is that each node only has 128 KB of memory. Due to the distributed organization of memory, nodes have to send a message to the node whose memory is to be accessed. It then responds with the requested data. All communication between nodes is carried out via explicit messages.

Although the third assumption postulates a mesh network, we employ a torus network (like in Figure 3.2) due to its regular structure and not having "border" nodes. It has a completely timing-predictable behavior and is described in the following Section 3.2. Due to the NoC being the only way for the nodes to communicate with each other, they work completely isolated except when executing communication operations. This eases timing analysis, because each node can be analyzed without the need to respect other nodes.

The fourth assumption is the consequence of the others: A timing analysis of the code running on the nodes gives us information when each node wants to send or receive data. Thereby, data is transported via the statically scheduled NoC. This allows us to determine when data arrives at a target node. Thus, we can combine the WCET estimation of core-local code with the latest points in time when messages are to be sent or received. Altogether, we get the total WCET estimation for the distributed application running on the RC/MC processor. Considerations for the analysis process are given in Section 3.3 and an example is illustrated in Section 3.6.

3.2. Details on Network Communication

For communication between the nodes, *flits* are sent from one node to another over the NoC. A flit is the smallest unit to be sent via a NoC [BM06], in our case it has a payload of 64 bits (corresponding to the size of a register in the register set). In many NoCs, messages are composed of a header flit containing routing and further information and data flits following it [BM06, AIS09]. At the RC/MC, we go a different way by supporting only single flit messages. Headers are transmitted via additional signal lines next to the payload signal lines [MU12]. For example in a 4x4 NoC, we have links between the nodes with a width of 73 bits: 64 bits for the payload of each flit, 4 bits for addressing the target node, another 4 bits for specifying the sender node and 1 bit to indicate if the current flit is valid (i.e. if a flit is transmitted or data on the lines is to be ignored).



(a) 4x4 Torus [MU12]

(b) Folded 4x4 Torus [MU12]

Figure 3.2.: Torus and folded torus for a 4x4 NoC.

Nodes in the NoC are connected via *routers* [AIS09]. One NoC router is installed at each node. At the RC/MC, we employ a lightweight NoC router called Pater-Noster [MU12]. It connects each node with its neighbours and manages when and how flits are moved forward or stored in buffers. To have short distances between nodes, they are arranged as quadratic 2D torus like in Figure 3.2a. Because the wrap arounds may lead to long connection lines with increased latencies, it may also be realized as folded torus like in Figure 3.2b. Following the definition of Dally et al. [DT04], we define n as dimension of the NoC. Thus, we have $n \cdot n = N$ nodes on the chip, e.g. in Figure 3.2 $n = 4$ and $N = 16$. Due to the regular and symmetric structure of the torus it looks same from the perspective of any node. As a result, tasks are placement-independent.

The torus consists of unidirectional x-rings (horizontal) and y-rings (vertical). Flits travel along the rings following the concept of xy-routing (see e.g. [NM93]): First, they are sent in x-direction until they reach their target column. Thereby, intermediate nodes instantly forward flits, so that no buffers are needed. In the target column, the flit is stored in the *corner buffer*. Finally, the flit takes the ring in y-direction until it reaches its target node. Because all nodes take the same way, it is guaranteed that all flits are delivered in the same order as they were sent. Due to instantly forwarding flits in x- and y-rings, only few buffers are required: Send and receive buffers are organized as *first in first out (FIFO)*, while the organization of corner buffers is dependent on the schedule, see Subsection 3.2.1. In our implementation, we realized send buffers with 4 slots and receive buffers with N slots. Thereby, it is possible to put/take flits to/from the buffers in the same cycle as they are put/taken from the network. This is possible due to the concept of

the RC/MC called *pipeline-integrated message passing (PIMP)* [MFSU19]. Usually, the NoC is connected via an NI [AIS09]. At the RC/MC, the NI is integrated into the execute stage of the pipeline for fast access. Thus, it takes only one cycle to insert a data word from the pipeline into the NoC. After it was delivered, a data word is available in the pipeline after one cycle. The size of the corner buffers is dependent on the schedule, which will be explained in the next Subsection 3.2.1.

3.2.1. Schedules for the Coordination of Flits

A NoC can work either in *best effort (BE)* or *guaranteed service (GS)* mode [SPG97, GvMPW02]. BE means that flits are forwarded whenever possible, i.e. the NoC is optimized for maximum throughput (for high-performance computing). However, BE can result in flits taking extra rounds or other behavior which is hard to predict. Therefore, *schedules* are employed to be able to provide a maximum delay how long flit transfer takes (GS) [SPG97]. A schedule manages communication in a NoC: It defines for each point in time which flit is at which point in the NoC (TDM). This means that for each pair of senders and receivers the position within the NoC is uniquely defined – ensuring that no other node will place a flit there at the same time. Thus, a schedule avoids collisions and extra rounds and enables us to estimate a *worst-case transportation time (WCTT)* for a flit transfer. This also guarantees *fairness*: All nodes can utilize equal transportation slots – no "babbling idiot" can slow down network communication.

Schedules can be divided into two groups: Custom and general-purpose schedules [SBSK12, MU14]: For a custom schedule, all details of communication have to be clear at design time. It is then computed for a specific combination (incl. task placement) of all nodes on the chip. The advantage of a custom schedule is a good (worst-case) performance, but at the cost of recomputing the complete schedule each time something is changed. On the other hand, a general-purpose schedule describes a regular pattern defining how many flits may be sent or received by the nodes. As such, a general-purpose schedule is application-independent, but its (worst-case) performance may be worse than that of custom schedules. Its advantage is that recomputation is not necessary when further applications are added, e.g. when several applications are put together to run on one ECU in a car (composability). Moreover, general-purpose schedules allow us to give general statements about communication behavior and changes are easier to handle than with custom schedules. Furthermore, we can exploit some of their properties to improve their general (worst-case) performance. Therefore, we will only consider general-purpose schedules in the remainder.

3.2.2. General-Purpose Schedules

General-purpose schedules define a pattern for the communication between all nodes in a NoC [SBSK12]. One instance of this pattern considering all nodes is called *period*. These periods are repeated indefinitely, allowing nodes to send flits to the same nodes again at each further period. Periods may be divided into *rounds* describing a repeated pattern for a subset of all nodes [MU14]. For example, a schedule might process flit transfer columnwise. Then, each round may handle the transfer of flits of a specific column.

Table 3.1.: Overview on schedules considered in this thesis.

name	description	admission time	transp. time
All-To-All	Each node is allowed to send one flit to every other node. It also may receive flits from every other node.	$\left(\frac{n^2(n-1)}{2}+2\right)\cdot f$	$\frac{n^2}{2}+2n$
One-To-One	Each node can send and receive at most one flit.	$n\cdot\chi\cdot f$	$2n$
One-To-All	Each node may send one flit to one other node and can receive flits from all other nodes.	$n^2\cdot\chi\cdot f$	$2n$

In Table 3.1, we give an overview on general-purpose schedules relevant for our thesis. For each schedule, we give a brief description as well as its *admission time* and *transportation time*. The admission time states how long a flit may wait in the send buffer before it is injected into the NoC (e.g. because it has to wait for previous flits to be sent first or because the schedule allows flit insertion only at specific points in time). On the other hand, the transportation time states how long the flit takes to travel through the NoC. Added together, transportation time and admission time give the WCTT, i.e. how long it takes for a flit in the worst-case to travel through the NoC from a sender node to a receiver node including waiting times.

The first schedule called *All-to-All (A:A)* was proposed by Schoeberl et al. [SBSK12]: Within one period of the schedule, each node is allowed to send and receive flits to/from any other node. However, each node is allowed to send at most one flit to one particular node. In the worst-case this means that all nodes send and receive one flit to/from every other node. When utilizing the PaterNoster router [MU12], flits are sent alternately to columns with far and close distance. Thus, it is quite difficult to put them into the send buffer in the right order. To deal with this issue, Sewing implemented and evaluated a variant with a fully searchable send

buffer [Sew18, Sew19]. This *variant 1* leads to a considerably improved performance compared to variants with a FIFO send buffer. Therefore, we will consider this variant in the rest of this thesis and especially for our evaluation[2]. The corner buffer has to provide at least n buffer slots. Because all nodes can communicate with all other nodes in the same period, periods are quite long ($O(n^3)$) [SFMU16]. This leads to long WCTTs when the same two nodes exchange several flits.

An approach to overcome this issue is the *One-to-One (1:1)* schedule from Mische et al. [MU14]: Each node is allowed to send and receive at most one flit each period. This is much stricter than All-To-All, but results in shorter periods ($O(n)$) and communication times when not all nodes have to be reached [SFMU16]. Furthermore, the worst-case is the same as the average case: Each node sends and receives one flit. Sending several flits takes several periods as does sending one flit to different receivers (χ). Thereby, n corner buffer slots are installed at each node. In his master's thesis [Wal19], Walter implemented the 1:1 schedule for the RC/MC and optimized it. He also integrated `ready` synchronization (see Chapter 4) and hardware supported barriers (like in Chapter 6). However, before communication between two nodes can take place, it is always necessary to first synchronize them via an additional synchronization NoC to ensure that at most one flit is received by each node in each period. The time for synchronization is dependent on the organization of the synchronization NoC and has to be added to the admission and transportation times from Table 3.1.

Therefore, we will focus on the *One-to-All (1:A)* schedule [MU14] in this thesis, which offers a good trade-off between the A:A and 1:1 schedules. It allows each node to send at most one flit to one other node per period and each node may receive flits from all other nodes. This is very easy to realize as each flit can only be sent out when the next period has begun. On the receiver side, nothing has to be respected because the schedule is designed in a way that each cycle one flit from some other node may arrive. Therefore, periods are not too long ($O(n^2)$) and no synchronization NoC is needed. Sewing implemented and compared several variants of the A:A and 1:A schedules in his master's thesis and designed a new variant of the 1:A schedule without the need for corner buffers [Sew19]. It is explained in detail in Appendix B.3. In the remainder, we will only focus on this *variant 3*. However, the presented concepts also work with the variants 1 and 2, which require 1 and n corner buffer slots, respectively.

A detailed comparison of different general-purpose schedules is carried out by Stegmeier [Ste19]. In the remainder of this thesis, we will present several extensions for the RC/MC. Only a part of them works with all schedules: `ready` synchronization

[2]The admission time of A:A may vary between different implementations. Our admission time in Table 3.1 comes from the implementation of Sewing [Sew18, Sew19].

(Chapter 4) is required for all schedules to ensure proper operation without buffer overflows. In the Chapters 5 and 6, we focus on improvements for the 1:A schedule. Finally, in the evaluation in Chapter 7, we will compare the 1:A schedule including these improvements with the 1:1 schedule and the A:A schedule.

3.3. Programming Model for the RC/MC

For high efficiency and timing predictability, software for our platform should be developed in a way following the Bulk Synchronous Parallel (BSP) model. The backgrounds are presented in Subsection 3.3.1. To implement programs in practice, we recommend the use of Message Passing Interface (MPI) for communication and synchronization operations. An overview is given in Subsection 3.3.2.

3.3.1. Bulk Synchronous Parallel Model

Estimating a WCET is already complex for applications executed on a single-core platform [WEA+08]. Parallel execution increases this complexity, because the behavior of other nodes may directly influence the timing of the currently considered node [RBS+10, NPB+14, UBF+16, FJO+16]. One way to deal with this issue is to implement parallelism only at specific points in the program, where the participating nodes work together in a coordinated way (for a pattern-based approach see e.g. [McC10, JGU+14, FJO+16]).

For a massively parallel platform like the RC/MC, the *Bulk Synchronous Parallel (BSP)* model [Val90, SHM97] seems to be most promising to coordinate nodes and achieve good timing predictability. Thereby, the same program code is executed simultaneously on all participating nodes in *supersteps*:

1. At the beginning of each superstep, *local computation* takes place, i.e. each node works on its own.

2. Then, there is a phase of *global communication*, where all nodes may exchange data, e.g. results from the local computation.

3. Finally, all nodes wait at a *barrier* until all other nodes have finished their superstep. Afterwards, they can continue with the next superstep.

As outlined by Lisper [Lis12], BSP is beneficial for timing analysis. From his point of view, the excution time of local computation and global barriers should be easy to bound. Thereby, it is sufficient to estimate the WCET of local computation parts only once, because all nodes execute the same code. Thus, Lisper sees the main challenge at the timing analysis of communication. At the RC/MC platform, this issue is overcome with the GS NoC and the schedules, which enable us to give upper timing bounds on communication.

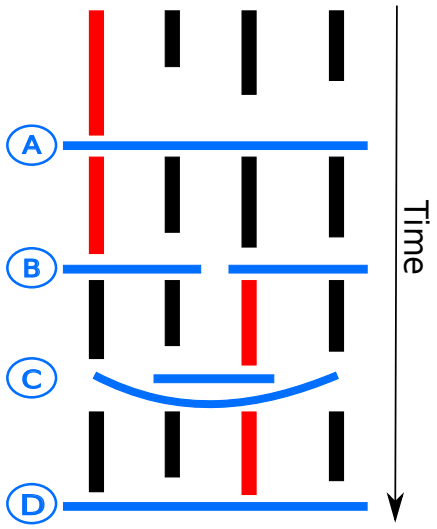


Figure 3.3.: Structure of BSP-like programs running on the RC/MC: Each column represents one node. Long vertical bars represent local computation – the longer the bar, the longer it takes. (A), (B), (C) and (D) represent communication between nodes as well as barriers. From the beginning to (A), we have a global superstep, while the following parts are local supersteps (between (A) and (B), (B) and (C), (C) and (D)). These local supersteps together form one global superstep, starting at (A) and ending at (D). Thereby, the maximum WCET estimate of all local supersteps has to be considered to get the WCET estimate of the global superstep (longest path, coloured in red).

In contrast to the BSP model, we allow barriers to be non-global. On the one hand, other applications may be executed on the same many-core. As long as they do not communicate with each other, they can be treated like working in total isolation and do not need joint barriers. On the other hand, an application may consist of some parts with application-wide barriers and other parts with non-global barriers. Figure 3.3 gives examples for both: It illustrates four nodes (represented by the columns) carrying out local computation (long vertical bars – the length stands for the computation time) and then communicating with each other/meeting at barriers (horizontal bars at (A), (B), (C) and (D)). A classical BSP superstep is shown from the beginning to (A): Local computation is followed by global communication/a global barrier. Afterwards, at (B) and (C) only a subset of all nodes is involved at communication and the corresponding barriers. Thus, we call these parts *local supersteps* and the original supersteps *global supersteps* for a clear separation of concerns. Internally, local supersteps can be treated like global supersteps due to not influencing other nodes. However, when local supersteps merge, the longest path of all local supersteps has to be respected as worst-case, i.e. the red path in

Figure 3.3 indicates the worst-case path.

Moreover, we allow implicit barriers, e.g. when all nodes wait for one node to broadcast some result. When this node is on the longest path, all nodes have arrived at this point in the program and wait for the result. In this case, the phases of global communication and barrier join to one common part of the superstep. Adding a dedicated barrier is not necessary.

For estimating a WCET for our BSP-like programs, the WCET estimates of all global supersteps can be added, because each of them is self-contained and they are executed one after another. Thereby, the WCET estimate of a global superstep is composed of the WCET estimates of local computation, the WCTTs of communication and the time for the barrier, which are to be added. The same holds for local supersteps. However, when local supersteps are involved, the maximum of all local supersteps added together gives the WCET estimate of the global superstep (like the red worst-case path in Figure 3.3).

We see the phases of global communication and barrier to be not only pure communication operations, but they may also contain some code to be executed on the PEs. On the one hand, this is necessary in any case, because communication only takes place when the corresponding assembly instructions are executed (see Section 3.4). On the other hand, we see communication operations to be more than sending a message from one node to another. Nodes may interact in a more sophisticated way, e.g. distributing or collecting data or broadcasting a result value to all participating nodes. Therefore, we consider communication not as single operations concerning only message transfer, but always as holistic communication operations, including sequential code for coordinating communication.

As a result, the timing analysis of global communication not only has to consider WCTTs, but also the related sequential code. Together they form a basic communication block, which may be reused in the same or other applications. As long as this communication block is reused for equivalent situations (e.g. same number of participants, same number of values to be transmitted), it will always have the same worst-case behavior. Thus, estimating a WCET is necessary only once for each communication block, independent where and how often the same communication block is utilized again.

3.3.2. Realization in Software via MPI Collective Operations

On the software level, our programming model can be implemented with *message passing interface (MPI)* [Mes15], the de facto standard for message-based communication [Hem94]. It realizes the BSP concept with *collective operations*, where all or a subset of all nodes work together (representing global communication). Thereby, most of these collective operations implement an implicit barrier combining the phases of global communication and barrier. Some examples of MPI collective

operations are listed in Table 3.2.

Table 3.2.: Examples of MPI collective operations.

name	description	parameters
`MPI_Barrier`	All participating nodes wait for each other until they have all arrived at this point of the program. Then they continue program execution.	Number of participating nodes
`MPI_Bcast`	One node broadcasts one or several values to other nodes.	Number of participating nodes, Number of values to be broadcasted
`MPI_Gather`	Values distributed at different nodes are collected at one node.	Number of participating nodes, Number of values to be collected
`MPI_Reduce`	Values distributed at different nodes are summarized at one node by reducing them, e.g. global minimum, maximum or sum.	Number of participating nodes, Number of values to be reduced, Type of operation

`MPI_Barrier` leaves global communication out and only implements an explicit barrier. At `MPI_Bcast`, all participating nodes wait for one or more values from a root node (global communication). An explicit barrier is not necessary when the collective operation is realized in a way that first ensures all nodes arrive at the point in the program where they wait for the value to be broadcasted and then the broadcast operation starts. `MPI_Gather` works the other way round: One node collects values from all other participating nodes. Therefore, global communication and barrier can again be joined when implemented in the right way: The nodes sending data to the root node should only continue program execution when all participating nodes have sent all values to the root node. `MPI_Reduce` is a variation of `MPI_Gather`: The values distributed on different nodes are summarized and the root node finally holds the overall result. It may be implemented in a similar way as `MPI_Gather`. All of these collective operations in Table 3.2 are reused in a wide field of applications. In Figure 3.3, they may be placed at (A), (B), (C) or (D).

As already mentioned in the previous Subsection 3.3.1, each communication operation has the same worst-case behavior when it is reused for equivalent situations. For example, on a given platform with $N = 16$ and a 1:A schedule, an execution of `MPI_Barrier` for all nodes will always have the same WCET estimation, because the same code will be executed and the same number of nodes has to communicate

with each other. Now we get one step further: The column *parameters* in Table 3.2 shows which parameters influence the WCET of a collective operation.[3] With these parameters, we can assemble a formula for each MPI collective operation estimating its WCET. Thereby, each parameter becomes a variable in the formula. In the timing analysis of a parallel program, each variable has to be set to a concrete value to get a WCET estimation of the MPI collective operation. As a result, we are able to estimate WCETs for these MPI collective operations for different situations, while carrying out a timing analysis only once.

In Section 3.6, we will illustrate an example how to analyze an MPI collective operation. Further formulas for MPI operations will be assembled in Chapters 5 and 6. They will be utilized in the case studies in Chapter 7.

3.4. Programming the RC/MC

Our RC/MC processor employs the RISC-V ISA version 2.2 [WAE17]. To realize explicit message passing, we had to extend the RISC-V ISA with some customized instructions. They are introduced in [MFSU19] and implemented in [Gor18]. An overview on them is given in Tables 3.3 (for sending and receiving flits) and 3.4 (for checking the status of buffers). Their execution times as well as the execution times of all RISC-V assembly instructions on the RC/MC platform are listed in Appendix B.2.

Table 3.3 describes instructions for sending and receiving flits: `snd` (send) expects two source registers – one contains the data word to be sent and the other one specifies the destination node. Because the RISC-V pipeline only allows one destination register, receiving a flit and determining the sender node is split into two instructions called `rcvp` (receive payload) and `rcvn` (receive node id). Thereby, removing a flit from the receive buffer is integrated into the instruction `rcvp`.

Before executing the instructions from Table 3.3, it is crucial to check the status of the send/receive buffers with the branch instructions in Table 3.4: When the send buffer is full, `bsf` (branch if send buffer is full) jumps to the given address. In contrast, `bsnf` (branch if send buffer is not full) jumps when there is space left for at least one flit. With these instructions, the send buffer can be checked before inserting a new flit via the `snd` instruction. Should `snd` be called when the send buffer is full, an exception is raised. The same holds when accessing the receive buffer and no flit is stored there. Therefore, the branch instructions `bre` (branch if receive buffer is empty) and `brne` (branch if receive buffer is not empty) work the same way: They

[3] A further parameter not listed here is the WCTT composed of transportation time and admission time from Table 3.1. It is mainly dependent on the NoC dimension n. However, as long as the schedule and n remain the same (i.e. we remain on the same platform), the WCET of a collective operation is only dependent on the parameters in Table 3.2.

Table 3.3.: Overview on our RISC-V instruction set extension for sending and receiving flits with pipeline-integrated message-passing [MFSU19, Gor18].

mnemonic	destination register	source register 1	source register 2	function
`snd`		*receiver*	*payload*	send *payload* to *receiver*
`rcvp`	*payload*			store the *payload* from the oldest flit in the receive buffer in the destination register and remove this flit from the receive buffer
`rcvn`	*node id*			store the *node id* of the sender from the oldest flit in the receive buffer in the destination register

Table 3.4.: Overview on our RISC-V instruction set extension checking the status of pipeline-integrated message-passing [MFSU19, Gor18].

mnemonic	source register 1	source register 2	immediate value	function
`bsf`			*address*	when send buffer is full, jump to *address*
`bsnf`			*address*	when there is space left in the send buffer, jump to *address*
`bre`			*address*	when receive buffer is empty, jump to *address*
`brne`			*address*	when there is a flit waiting in the receive buffer, jump to *address*

jump to the given address when the receive buffer is empty (`bre`) or when there is a flit stored there (`brne`).

While waiting for a free send buffer slot or the arrival of the next flit in the receive buffer, the PE may execute alternative code by branching there. Alternatively, it can spin on the status branch instruction. Or the instruction may be extended to energy saving waiting: The PE could sleep and be woken up by the NI when the status changes. However, this is beyond the scope of our thesis. In the remainder, we will always spin on the status branch instruction to have a simple case for timing analysis.

Table 3.5.: Overview on RC/MC specific *Control and Status Registers (CSRs)*.

CSR #	description
`0xc70`	Number of nodes present on the platform
`0xc71`	ID of the node where the code is executed

Moreover, we added two CSRs to easily retrieve parameters of the architecture, which might be valuable for the distributed computation. They are listed in Table 3.5. From the CSR `0xc70`, we can read how many nodes are present on the platform. This enables us to write programs in a general way, so that they can always utilize *all* available nodes. The other CSR `0xc71` returns the ID of the current node. This allows e.g. to check if the current node has to coordinate communication between other nodes (i.e. is "root/master node").

In the next chapters, we will introduce several more instructions to efficiently work with our hardware extensions. These new instructions will be presented and explained in the corresponding chapters. An overview on all RISC-V ISA extensions is given in Appendix A. It also contains the exact binary definitions of our new assembly instructions. Thereby, we utilized opcodes which are reserved for custom extensions of the RISC-V ISA: They are called *custom-2* (sending/receiving flits) and *custom-3* (status branches) in the RISC-V specification [WAE17].

3.5. Hardware Prototype and Simulation

For the verification that our concepts work, we implement them as *Very High Speed Integrated Circuit Hardware Description Language (VHDL)* model. Thereby, our work builds on the RC/MC implementation from Mische et al. [MFSU17], with updated PIMP NI (integrated into the execute stage) as described in [MFSU19] and implemented in [Gor18]. An estimation of its power consumption is available from Bauer [Bau18].

The basic structure of a node is illustrated in Figure 3.4: A 5-stage pipeline is connected to the core-local memory and to the network router. The send buffer (`SB`)

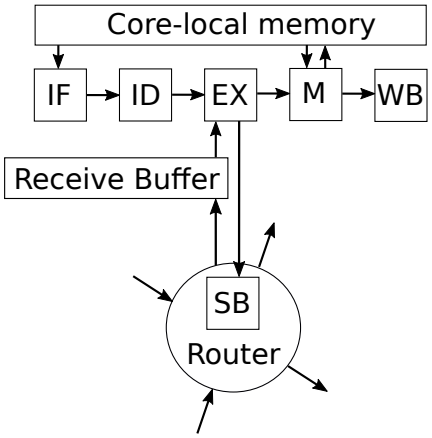


Figure 3.4.: Hardware structure of a node (simplified): 5-stage pipeline connected to core-local memory and to the NoC router via the send and receive buffers.

is located in the network router, while the `receive buffer` is part of the execute stage due to the PIMP concept. In the figure we omitted the register set and I/O connections to keep it simple.

To estimate hardware costs, we carry out a synthesis of the VHDL model via Altera Quartus Prime 16.0.2 [Int19a] for a Stratix V *Field-Programmable Gate Array (FPGA)* board [Int19c]. Due to the size of the FPGA chips, it is only possible to synthesize the VHDL model for a limited number of nodes. Furthermore, synthesis times increase quite fast, because optimal placement of the logic modules is an NP-hard problem [CM07]. For being able to place as many nodes as possible on the chip, our VHDL model only implements the RV64I ISA [WAE17] including our RC/MC specific extensions. Hardware costs are measured in *Adaptive Logic Modules (ALMs)* [Alt06], registers and memory bits. Thereby, an ALM is the basic building block of Altera/Intel FPGA chips. It is composed of a 6-input *look-up table (LUT)*, two adders and two registers and supports various configurations[4]. When we evaluate hardware costs, we compare the costs of our VHDL model synthesized for the FPGA chip before and after our new components are integrated. For better comparability, we divide the ALMs, registers and memory bits of the total design through the number of synthesized nodes to get numbers for one node. Because all of our extensions will not have any impact on memory bits, their number remains the same before and after our hardware extensions are integrated (each node in our prototype utilizes 69 632 memory bits). Thus, we will never report memory bits when analysing hardware costs.

For a functional test of the VHDL model, we employ the *G Hardware Design*

[4]For details see [Alt06, Int19b].

Language (GHDL)[5]. It is a VHDL compiler, which can be utilized to simulate the execution of a VHDL model. However, GHDL cannot be used for larger hardware designs as it requires lots of RAM (more than 16 GB for a 6x6 node NoC). Thus, we also employ a cycle-accurate simulator written in C and called *many-core simulator (MacSim)* [MMU11, MFSU17]. It implements the full RV64IMFD ISA [WAE17] including our RC/MC specific extensions. To ensure a cycle-accurate comparison between the VHDL model and MacSim, we use a technique elaborated by Tafertshofer [Taf16] and improved by Gorlo [Gor18]. Thereby, GHDL and MacSim create output files containing the changes to registers at each cycle during the execution of a program. These output files are compared and when they match, the functional behavior of both the VHDL model and MacSim coincide. To achieve a good quality of the comparison, we execute several programs running a few million cycles.

In the remainder, all hardware extensions are implemented both in the VHDL model and in the MacSim. The hardware costs are estimated via a synthesis utilizing Altera Quartus Prime and actual execution times of applications are taken from executing them in MacSim. Because the RC/MC does not contain any speculative components and hidden states, a repeated execution of the same program with the same parameters leads to exactly the same results. Therefore, a single execution is sufficient. When we estimate a WCET of the execution of a program or function, we combine a WCET analysis of local computation parts with a manual NoC WCTT analysis as described in the following Section 3.6.

3.6. Timing Analysis for the RC/MC

In the timing analysis of local computation, we assume that the RV32IMFD ISA including our RC/MC specific ISA extensions is implemented. The execution times of all assembly instructions on our platform are fixed – they are listed in Appendix B.2. For the timing analysis of program code, we employ the static WCET analysis tool *Open Tool for an Adaptive WCET Analysis (OTAWA)*[6] [BCRS11]. We extended it to support the RV64IMAFD ISA including our RC/MC specific ISA extensions[7]. However, the OTAWA loader is currently only implemented for 32 bit binary files. Therefore, our code will support 32 and 64 bit, but we will carry out our analyses with 32 bit compiled files. Furthermore, all software we analyze was

[5] Homepage: `http://ghdl.free.fr/`

[6] Homepage: `https://www.otawa.fr/`

[7] See `https://www.github.com/hcasse/riscv` for the RISC-V description for the required GLISS2 instruction set simulator [RCS09], `https://www.github.com/hcasse/otawa-riscv` for the RISC-V loader for OTAWA and `https://www.github.com/hcasse/otawa-rcmc` for the RC/MC extension for OTAWA.

adapted or implemented to follow the recommendations of Gebhard et al. [GCH11] and Bonenfant et al. [BBB+10] for good timing predictability. Moreover, we had to disable compiler optimizations (`-O0`), because otherwise it is impossible for OTAWA to link flow facts and binary file together. For the generation of flow fact files, we utilize oRange [BdMS08], which is part of the OTAWA framework. It also tries to find loop bounds for all loops in the analyzed program.

Although our analysis for sequential code parts is carried out for a 32 bit platform, we assume flits to be 64 bits wide. This is because except OTAWA everything for our platform is realized with 64 bit support. OTAWA's WCET estimations for 32 bits may lead to slightly improved results, because some values may have to be processed in two steps instead of one due to reduced register bandwidth. However, all results may be seen as pessimistic variations of a fully 64 bit analysis.

When considering WCTTs of flits, the transportation time and the admission time from Table 3.1 on page 14 have to be added for the corresponding schedule like in Formula 3.1 for the A:A schedule. For example, when $n = 4$, the WCTT of one flit is 42 cycles, while it is 120 cycles for four flits (see details in Formulas 3.2 and 3.3).

$$WCTT_{AA}(f) = \left(\frac{n^2(n-1)}{2} + 2\right) \cdot f + \frac{n^2}{2} + 2n \tag{3.1}$$

$$WCTT_{AA}(1) = \left(\frac{4^2 \cdot (4-1)}{2} + 2\right) \cdot 1 + \frac{4^2}{2} + 2 \cdot 4 = 42 \tag{3.2}$$

$$WCTT_{AA}(4) = \left(\frac{4^2 \cdot (4-1)}{2} + 2\right) \cdot 4 + \frac{4^2}{2} + 2 \cdot 4 = 120 \tag{3.3}$$

Following our programming model from Section 3.3, applications for our platform are always composed of alternating phases of local computation and global communication/barriers. Thereby, global communication/barriers are represented by MPI operations. These consist of program code and flit transfers. With the following example, we illustrate how the analyses of both can be combined to get a WCET estimation of an MPI operation. The analysis follows the concept described in [FSMU16] and was originally carried out by Brügmann [Brü19]. More MPI operations were analyzed for our platform by Unte [Unt18] and Bürger [Bür19].

3.6.1. Example: Timing Analysis of `MPI_Barrier`

We analyze a simple implementation of the operation `MPI_Barrier`, its structure is visualized in Figure 3.5. Although all nodes call the same function `MPI_Barrier`, they do not necessarily execute the same code: There is a case distinction between a root node and participating nodes. The root node collects information from all nodes that they have arrived at the barrier and notifies them that they can continue program execution when all have met at the barrier. Therefore, we see a root node on the left of Figure 3.5 and a participant node on the right. Usually, there is more

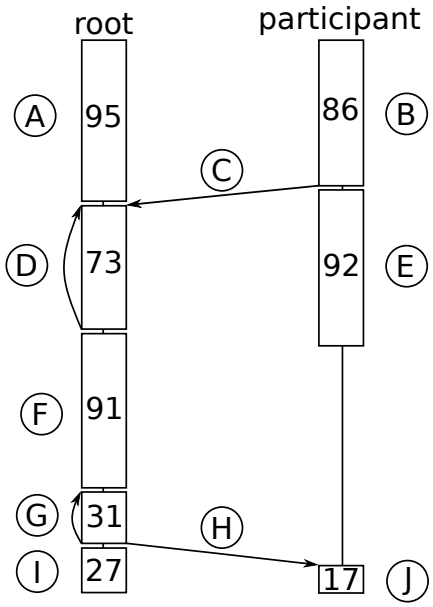


Figure 3.5.: Simple implementation of `MPI_Barrier`. The root node manages the barrier, while the participant node tells it that it arrived and waits for barrier release. There may be more participant nodes, but we included only one to keep the figure simple. Other participant nodes would behave exactly the same way as the participant node in the figure. Boxes represent local code execution, arrows between root and participant denote flits, arrows from the end of a box to the beginning of the same box illustrate loops. Numbers inside of the boxes specify the WCET estimation of this code part. The meaning of the different steps is described in Table 3.6.

than one participant. However, we illustrated only one participant in Figure 3.5, because all participants behave in the same way. It should be mentioned that more participants might require more buffer space at the root node. Throughout the example, we assume that buffers are large enough. A technique how we can rely on this will be presented in Chapter 4. `MPI_Barrier` works as described in Table 3.6, where the WCET estimates of the different steps are also given.

In the formulas for the WCET estimates in Table 3.6, $\#participants$ is the number of participants arriving at the barrier (excluding the root node[8]). Moreover, $WCTT(1,\chi)$ denotes the WCTT for sending one flit to χ receivers and $WCTT(x,5)$ the WCTT for sending x flits to 5 receivers. At the latter, the admission time is only added once, because at the next period the next flit is already waiting in the send buffer. This is the first case at step (G): $31 + WCTT(1,\#participants)$. Therefore, the WCET estimate is driven by flit transportation. In the other case $31 \cdot \#participants + WCTT(1,1)$, a loop iteration requires more time than a period. Thus, we consider the WCET estimate of all loop iterations and add the WCTT of the last flit to be transported (because all other flits have already reached their destination). Due to the WCTTs being dependent on the schedule and the NoC size (see Section 3.2), both cases are relevant and have to be considered for the worst-case path.

$$\begin{aligned} WCET_{root}^{SimpleBarrier} &= \max(95, 86 + WCTT(1,1)) \\ &\quad + 73 \cdot \#participants \\ &\quad + 91 \\ &\quad + 31 \cdot \#participants \\ &\quad + 27 \end{aligned} \tag{3.4}$$

$$\begin{aligned} WCET_{participant}^{SimpleBarrier} &= \\ &= \max(95, 86 + WCTT(1,1)) \\ &\quad + 73 \cdot \#participants \\ &\quad + 91 \\ &\quad + \max(31 + WCTT(1,\#participants), 31 \cdot \#participants + WCTT(1,1)) \\ &\quad + 17 \end{aligned} \tag{3.5}$$

Altogether, we get $WCET_{root}^{SimpleBarrier}$ in Formula 3.4 as estimation for the root node and $WCET_{participant}^{SimpleBarrier}$ in Formula 3.5 for the participant nodes. We assemble $WCET_{Total}^{SimpleBarrier}$ in Formula 3.6 as maximum of both, because we are always

[8]For example at a global barrier in a $4x4 = 16$ node NoC $\#participants$ would be 15.

Table 3.6.: Description of the structure and WCET estimates for a simple implementation of `MPI_Barrier` as shown in Figure 3.5.

step	description	WCET estimate
(A), (B), (C)	First, root and participant nodes initialize, e.g. they find out whom they communicate with and if they are root or participant nodes. This initialization code has a WCET estimate of 95 cycles for the root node, while it is 86 cycles for the participant nodes. At the end of the initialization, the participant nodes send a flit to the root node notifying it that they have arrived at the barrier.	$\max(95, 86 + WCTT(1,1))$
(D)	Then, all arrived flits are processed at the root node. Thereby, the time for waiting for the flits from the participant nodes was already respected at the WCET estimate of steps (A) to (C). Thus, we can assume that all flits already have arrived. Processing takes place in a loop, which has to be iterated once for every participant and each iteration has a WCET estimate of 73 cycles.	$73 \cdot \#participants$
(E), (F)	Both the root node and the participating nodes execute some sequential code before they can continue with the next step.	root: 91 participant: 92
(G), (H)	Now the root node sends a flit to each participant node to tell it that it can continue program execution. The loop for sending out these flits needs estimated 31 cycles in the worst-case for each iteration. When the period of the schedule is longer than 31 cycles, then the WCET estimate is driven by the WCTT of the flits. Otherwise, the execution of the local code is on the worst-case path.	$\max($ $31 +$ $WCTT(1, \#participants)$, $31 \cdot \#participants$ $+ WCTT(1,1)$ $)$
(I), (J)	At the end of the function, there is a short sequential code part both at the root and the participant nodes. It has a WCET estimate of 27 cycles at the root node and 17 cycles at the participant nodes.	root: 27 participant: 17

interested in the WCET estimation for a complete MPI operation.

$$WCET_{Total}^{SimpleBarrier} = \max(WCET_{root}^{SimpleBarrier}, WCET_{participant}^{SimpleBarrier}) \tag{3.6}$$

There may be some points where the timing analysis may be improved, e.g. by reducing waiting times or considering pipeline states. However, this is beyond the scope of our thesis – our principal concern is that the timing analysis delivers suitable results for our platform.

Basically, the same structure and a similar WCET formula as for `MPI_Barrier` may be utilized for a simple implementation of `MPI_Bcast`. Thereby, the participant nodes send the flit to the root node to notify it that they are ready to receive data (flit at step (C)). This will be described in detail in the following Chapter 4. At step (G) the loop would have to be extended to a nested loop: The root node iterates over all values to be transmitted (outer loop) and all participant nodes (inner loop). This order is important to optimally utilize the schedule and avoid buffers to run full. In the timing analysis, the loop iterations of both loops have to be respected. An implementation of `MPI_Bcast` following this principle was analyzed by Bürger [Bür19]. His result is included in Section 5.8.

These considerations show that not only the timing analysis of MPI operations may be reused, but also parts of the timing analysis may be reused for similar MPI operations.

4

`Ready` Synchronization: Real-Time Flow Control

Abstract. At the RC/MC architecture as it was presented in Chapter 3, all nodes are allowed to send flits to all other nodes at any time (limited by the time-division multiplexing (TDM) schedule). When all nodes send flits to one node at the same time, this may lead to a buffer overflow. Thus, we introduce a simple synchronization mechanism ensuring that data is transferred when nodes need it and are able to handle it. Thereby, it avoids full buffers and the need to handle flits that arrive too early. However, software synchronization is not able to completely achieve these objectives, because its synchronization flits may still interrupt data transfers and fill buffers.
Therefore, we propose a lightweight hardware synchronization. It requires only small architectural changes as it comprises only very small components and it scales well. For controlling our hardware supported synchronization, we add two new assembly instructions. Furthermore, we show the difference in the software development process and evaluate the impact on the execution time of global communication operations and required receive buffer slots.

4.1. Introduction

Besides the TDM schedule, there is no restriction for nodes when and where to send data at the RC/MC architecture presented in Chapter 3 – each node is allowed to send flits to any other node at any time. In the worst case, this may lead to all nodes sending flits to one node, which currently is busy processing some sequential program part[1]. It is then not able to process any incoming flit, which results in the receive buffer getting filled more and more. Therefore, nodes either have to handle incoming flits at any time or buffers have to be of an appropriate size. However, the largest receive buffer runs full when a node is too busy to process flits from it. Interrupting the execution of a node to handle incoming flits is also not an option, because our platform is intended for real-time programs: Giving upper timing bounds is impossible when nodes can be interrupted at any time.

One might ask if message coordination by just strictly following the BSP model from Section 3.3 would solve the problem. However, even at BSP-like applications, execution might diverge e.g. when the root node has to do coordination work or when some nodes follow different case distinctions of a program. Moreover, our platform is intended to support the execution of several applications simultaneously. Thus, flits from cooperating applications might arrive at any time. Another issue arises from the receive buffer of each node being organized as FIFO buffer. When a node communicates with another node and flits from other nodes arrive, these flits have to be handled although they are not needed yet.

Altogether, we cannot give timing guarantees and ensure correctness when an arbitrary number of flits from other nodes could arrive at any time.

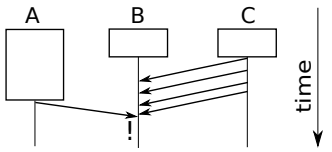


Figure 4.1.: Node C fills the receive buffer of node B, which is currently waiting for flits from node A. As a result, the receive buffer of node B is full and the request from node A gets lost. Boxes represent local computation times and arrows the delivery of flits.

One of the basic problems is illustrated in Figure 4.1: There are three nodes A, B and C running a parallel application. Each of them does some local computation (boxes), followed by communication (flits represented as arrows). The computation of node A takes a little bit longer than on nodes B and C. Meanwhile, node C

[1] An example would be a naive implementation of `MPI_Gather` [Mes15], where all nodes send values to one root node, which collects and stores them.

finishes its local computation and sends several flits to node B. Node A sends a request to node B, but the receive buffer of node B is already full with flits from node C. Because our TDM schedules do not provide extra rounds for flits (like the BE approach in [MU12]), the flit from node A gets lost.

Therefore, we introduce a synchronization mechanism we call `ready` synchronization: Our basic idea is that the receiver node "puts forth its hand" to the sender node when its receive buffer is empty and it is ready to handle incoming flits. For this, the receiver node sends a `ready` flit to the sender node. Afterwards, it is not necessary to send any more handshake flits between sender and receiver node. When the sender node has received the information that the receiver node is ready, it just starts sending as soon as it has reached the appropriate part in the program. When the receiver node is not yet ready, the sender node waits until the `ready` flit arrives. In contrast to other approaches, our `ready` synchronization is intended to find the point in time when communication in *both* directions can be started. This means when the receiver wants to reply to some action of the sender, another synchronization is not necessary as long as both agree that communication can still go on.[2]

By adding `ready` synchronization in the example in Figure 4.1, only `ready` synchronization flits and desired data flits arrive at the node. As a result, it is possible to give an upper timing bound, because each other node may send at most one `ready` synchronization flit. However, this leads to pessimism and requires a lot of buffer space to handle the worst case when *all* `ready` flits arrive at the same time. Thus, we suggest to realize our `ready` synchronization with hardware support. Our hardware implementation stores synchronization information and makes it available for the PE when it asks for it. Thereby, we focus on minimal hardware and synchronization overhead.

Altogether, we develop a cheap and simple hardware synchronization mechanism which can easily be controlled in software, to increase performance and timing predictability while decreasing receive buffer size and hardware costs. Our approach is independent of router design and network topology. Thus, it can be applied to other architectures than the RC/MC processor, too. We already published a more general version of this chapter in [FSMU18].

The remainder of this chapter is structured as follows: In the next Section 4.2, we present related work. Afterwards, we introduce our synchronization concept in Section 4.3. We describe how to implement it in software in Section 4.4 and in hardware in Section 4.5. Afterwards, we evaluate our synchronization mechanism in Section 4.6. Finally, we conclude in Section 4.7.

[2] It is up to the software developer when the next `ready` synchronization is reasonable.

4.2. Related Work

Most classical synchronization approaches were developed for distributed systems [RH90, TVS07], where several constraints have to be respected. For example, communication times might be very long, messages might get lost or a node may drop out surprisingly. In a NoC, all nodes are reliable and communication times are short [AIS09, BM06]. However, NoCs contribute a lot to the power consumption of many-core chips. The NoC of the Intel 80-core Teraflops research chip consumes 28% of the power per tile [VHR+08]. This percentage increases when more nodes are put on the chip [Bor10]. A high amount of this contribution stems from buffers. They need a lot of chip area, e.g. 60% of the tile area of the Tilera TILE64 many-core [WGH+07]. Nevertheless, compared to buffers in distributed systems, buffers in NoCs seem to be very small. This is due to distributed systems having plenty of buffer space because the main memory and swap space (hard disk) may be employed. Therefore, flow control is indispensable.

Our approach is a variation of *stop-and-wait protocols (sawps)* [KR12, TW10]: In the original sawp, the sender has to wait for an acknowledgement from the receiver after sending a flit and before sending the next flit. This means that each flit has to be acknowledged seperately and leads to a high overhead. In contrast, in our approach the sender waits for a synchronization flit before starting to send. Instead of acknowledging each flit several flits can be sent. Then, the next synchronization takes place (see details in Section 4.3).

Another concept is credit-based flow control [KM95], which works as follows: When a node wants to send data to another node, it asks it for *credit*. Then, the receiver node tells the sender how many receive buffer slots it can use. Therefore, the sender knows how many flits it can send. While sending, the receiver might update the credit, then the sender can send more flits. Credit-based flow control is implemented e.g. in the Æthereal NoC [GDR05]. Thereby, a forward channel is used to send data and a reverse channel to give feedback about buffer utilization. Our approach does not dynamically exchange detailed information about buffer utilization. Instead, it is intendend to only find the starting point of the communication.

In Message Passing Interface (MPI), the standard for message-based communication [Mes15], a function `MPI_Ssend` is defined for synchronous sending/receiving. It requires that the receiving nodes have already called the function before the sending node calls it. Therefore, it implements something similar as our `ready` synchronization for synchronous communication. However, this takes place at a higher abstraction level, while our synchronization is realized at low software level or even at hardware level. Moreover, our approach also works when the sender arrives at a communication part first.

Ruadulescu et al. [CMR+06] describe an approach to optimize the buffer size on NoCs with credit-based flow-control. They employ *time-division multiple access*

(TDMA) with an application-specific schedule, i.e. contention-free paths are determined at design time of the *System-on-Chip (SoC)*. Ruadulescu et al. focus on building a SoC with buffers being as small as possible for a specific application. In contrast, our approach works at execution time and is application independent. Instead of determining the optimal buffer size at design time, our focus is on using buffers of a specific size as efficient as possible without having buffer overflows.

4.3. Synchronization Concept

In NoCs, bandwidth is restricted and a lot of communication takes place between nodes. Therefore, our idea is to avoid buffer overflows with just one synchronization flit – we call it `ready` flit. It does not contain payload and is just used to indicate that the receiver node is ready to receive data.[3] When the receiver node is ready to handle incoming data from the intended sender, it starts communication by sending a `ready` flit. This way, the receiver node indicates that its receive buffer is free[4] and any incoming flit will immediately be processed.

On the other hand, the sender node does not send any flits to the receiver node before it receives the `ready` flit. Meanwhile, it has to wait for it or execute some alternative code. When the `ready` flit arrives at the sender, it knows that the receiver is now ready and can start sending. In this way, it is ensured that the receiving node has free buffer slots and is ready to handle the received flits. Should it be necessary to tell the receiver how many flits follow, a header flit containing all relevant data might be sent and processed in software.

The concept of ready synchronization is illustrated in Figure 4.2: As in Figure 4.1, node B waits for data from nodes A and C. Since it first needs data from node A, it sends a `ready` flit there. Node A starts sending flits after its local computation has finished. After receiving and processing all data from node A, node B sends a `ready` flit to node C, which in turn starts sending. This leads to node C waiting until node B is able to receive and process data and avoids any slowdown caused by full buffers. Altogether, problems occuring when nodes do not meet a communication part at the same time are avoided: A node sending flits waits until the receiving node signalizes that it is ready. And a node receiving flits only sends one `ready` flit when it is ready to receive data and then waits for flits from the sender (flit transfer

[3] When implementing `ready` synchronization in software, a particular payload is defined to represent `ready` flits. In the hardware implementation, a payload is not possible because `ready` flits do not reach the PE of a node.

[4] The safe way utilizing `ready` synchronization is to send a `ready` flit when the receive buffer is free. However, an optimization could be to send out the `ready` flit as soon as enough receive buffer slots are free. Thereby, it might be hard to estimate how many receive buffer slots are enough. Thus, we recommend that `ready` flits are only sent out when the receive buffer is empty.

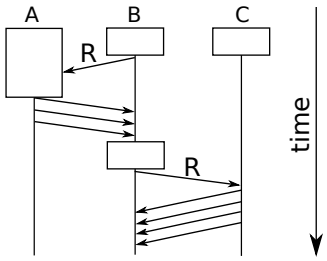


Figure 4.2.: Communication with `ready` synchronization: Each node waits with sending flits until the receiver node is ready (`ready` flits are denoted with *R*).

starts "on demand"). Should the sender not be ready to send data yet, it just has to store the `ready` flit and can start sending as soon as it has finished its computation.

Our procedure is completely safe when synchronizing each flit (like in sawp), but then synchronization overhead is way too high. Instead, we intend that `ready` synchronization takes place only once per message or program block (which may even imply communication in both directions). This works well as long as a message does not exceed a certain length and each program block is written in a way that ensures that the receiver can process incoming flits fast enough. When a sender delivers faster than its receiver can process incoming flits and it sends more flits than the receiver node has buffer slots, buffers could still run full. In this case, the maximum number of flits before the next `ready` synchronization has to take place is limited. The number of receive buffer slots at the receiver node has to be respected as well as the difference in time between executing a send and a receive operation.[5] When processing of flits at the receiver node is at least as fast as flits are sent from the sender node, there is no problem and therefore no restriction on the number of flits to be sent.

On architectures different from the RC/MC, it has to be considered that not all flits might arrive in constant time periods, although they may be sent in such. On their way through the NoC, flits may be hindered (e.g. collisions, deflection routing) on their direct way between sender and receiver. Thus, there might be periods where few flits arrive and others with more flits arriving (jitter) – which might be more than the receiver node might handle at a time. An ideal number of flits between two `ready` synchronizations is then up to the software developer and influenced by specific details of the architecture. Therefore, attention has to be paid that `ready` synchronization takes place in appropriate periods to avoid full buffers.

[5] For example, when a `send` operation takes 20 cycles and a `receive` operation takes 25 cycles, it takes 4 sends (80 cycles) to permanently occupy one more receive buffer slot.

4.4. Software Implementation of `ready` Synchronization

The `ready` synchronization concept is realizable without requiring additional hardware. Thereby, each node maintains a software array with one entry for each other node.[6] These entries indicate whether a node is ready to receive data (i.e. if a `ready` flit was received from a particular node). Nodes can send `ready` flits at any time. Thereby, `ready` flits are normal flits with a defined content (`-1` in the following two Code examples 4.1 and 4.2).

Code example 4.1 illustrates the source code of the sender side: After the initialization, the sender node has to check the `ready_array` for the state of the receiver node at `check_ready_array`. When it already indicates ready (value from `ready_array` is $\neq 0$), it can jump to `send`, where it first resets the entry in the `ready_array` and then starts sending (at `send_flit/wait4sendbuffer`). Otherwise, the code at `wait4ready` is executed: We wait for the next flit in the FIFO receive buffer and then fetch it. When it is a `ready` flit, the corresponding node is marked as ready in the `ready_array`. Otherwise, the received flit is a data flit and handled at `rcv_data`. This case is for debugging purposes only as it cannot occur when all nodes execute programs consequently implementing `ready` synchronization. After receiving the `ready` flit, `check_ready_array` checks if it was the one for the receiver node. In this case, it jumps to `send`, where sending data takes place as already described above. Otherwise, it waits for the next `ready` flit at `wait4ready`.

On the other hand, Code example 4.2 shows the source code of the receiver side: After the initialization, we check if the receive buffer is empty. When it is, a `ready` flit is sent to the sender node at `sendready`. Otherwise, the next flit is taken out of the receive buffer and checked for its type. First, we check if it is a data flit. This should never occur when `ready` synchronization is consistently implemented in all applications running on the processor. Thus, it should be a `ready` flit and the corresponding entry in the `ready_array` is marked as ready (set to 1). Then, the receive buffer is checked again. Since there are at most as many `ready` flits to receive as there are other nodes, the receive buffer will eventually be empty and the function can continue at `sendready`. After sending the `ready` flit, we wait for new flits arriving in the receive buffer at `wait4receive`. Each incoming flit is received and checked if it comes from the sender specified in register `a0`. In this case, it is stored in an array, which was specified in register `a1`. Otherwise, the node who sent the flit is marked as ready in the `ready_array`. As already described above, the `rcv_data` case should never be reached and is therefore only for debugging purposes.

[6] For easier and uniform addressing, the `ready_arrays` in Code examples 4.1 and 4.2 include one entry for the own node, which is never used.

Code example 4.1 Send operation including software `ready` synchronization.

```
.data
# array for storing ready states of nodes (16 in this example)
# 0 indicates that a node is not yet ready to receive data, while
# 1 denotes that data can be sent
ready_array: .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

.text
.globl send_data_sw_rdy
send_data_sw_rdy:               # function to send data with software ready
                                # expects receiver node in a0 ...
                                # ... address of data to send in a1 ...
                                # ... and amount of data to send in a2
# initialization
    addi t3, zero, -1           # -1 as value for ready flits
    addi t4, zero, 1            # 1 as value for nodes which are ready

# check ready array if receiver node is already marked as ready
check_ready_array:
    lb   t2, ready_array(a0)    # load state of receiver node ...
                                # ... from ready array
    bne  t2, zero, send         # when receiver node is ready (!=0)...
                                # ... jump to send operation

# receive flits and wait for ready flit
wait4ready:
    bre  wait4ready             # wait for flit in receive buffer
    rcvn t0                     # store node id of received flit in t0
    rcvp t1                     # store payload of received flit in t1
    bne  t1, t3, rcv_data       # received flit is not a ready flit ...
                                # ... process it as data flit
    sb   t4, ready_array(t0)    # store 1 to node's entry in ready array
    j    check_ready_array      # check if receiver node is now ready

# when node is ready, send data
send:
    sb   zero, ready_array(a0) # reset entry in ready array to 0
send_flit:
    ld   t0, 0(a1)              # load data to send
wait4sendbuffer:
    bsf  wait4sendbuffer        # wait for free slot in send buffer
    snd  a0, t0                 # send data to receiver node
    addi a2, a2, -1             # less data to send
    addi a1, a1, 8              # point to next data word to send
    blt  zero, a2, send_flit    # process next data word when existent
    ret                         # return when all data is processed

# received a data flit instead of a ready flit
rcv_data:
# for debugging purposes only, should never occur
# when all nodes execute code with ready synchronization
```

Code example 4.2 Receive operation including software ready synchronization.

```
.data
# array for storing ready states of nodes (16 in this example)
# 0 indicates that a node is not yet ready to receive data, while
# 1 denotes that data can be sent
ready_array: .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

.text
.globl receive_data_sw_rdy
receive_data_sw_rdy:            # function to receive data with software ready
                                # expects sender node in a0 ...
                                # ... address where to store received data in a1 ...
                                # ... and amount of data to receive in a2
# initialization
    addi t3, zero, -1           # -1 as value for ready flits
    addi t4, zero, 1            # 1 as value for nodes which are ready

# wait until receive buffer is empty
wait4freeRbuffer:
    bre  sendready              # receive buffer empty => send ready flit to sender
# receive buffer not empty => receive flit
    rcvn t0                     # store node id of received flit in t0
    rcvp t1                     # store payload of received flit in t1
    bne  t1, t3, rcv_data       # received flit is not a ready flit ...
                                # ... process it as data flit
    sb   t4, ready_array(t0)    # store 1 to node's entry in ready array
    j    wait4freeRbuffer       # check if receiver node is now ready

# send ready flit to sender
sendready:
    bsf  sendready              # wait until send buffer has free slots
    snd  a0,t3                  # send ready flit to sender

# receive flits from sender
wait4receive:
    bre  wait4receive           # wait for flits in receive buffer
    rcvn t0                     # store node id of received flit in t0
    rcvp t1                     # store payload of received flit in t1
    bne  t0, a0, other_nodes    # jump when flit comes from different sender ...
    sd   t1, 0(a1)              # ... otherwise store received data in memory
    addi a2, a2, -1             # less data to receive
    addi a1, a1, 8              # next memory location to store received data
    blt  zero, a2, wait4receive# process next data word when existent
    ret                         # return when all data is processed

other_nodes:
    bne  t1, t3, rcv_data       # received flit is not a ready flit ...
                                # ... process it as data flit; otherwise ...
    sb   t4, ready_array(t0)    # ... store 1 to node's entry in ready array
    j    wait4receive           # wait for next flit

# received a data flit instead of a ready flit
rcv_data:
# for debugging purposes only, should never occur
# when all nodes execute code with ready synchronization
```

4.5. Hardware Supported ready Synchronization

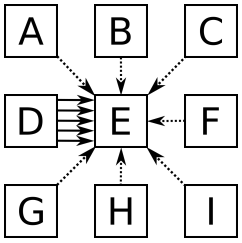


Figure 4.3.: Node D is sending data flits to node E (solid arrows), all other nodes send ready flits (dotted arrows) at the same time. Therefore, buffer space is needed for incoming data flits as well as for incoming ready flits.

The main drawback of the software ready implementation is that in the worst case ready flits from all other nodes have to be stored in the receive buffer besides data flits from communication with some node. Thus, the receive buffer has to be large enough to save all arrived ready and data flits: It should have at least as many receive buffer slots as there are nodes on the chip. Figure 4.3 illustrates an example with 8 other nodes: Node D sends a lot of data flits to node E (solid arrows). Then, all other nodes (A-C, F-I) send their software ready flits to node E at the same time (dotted arrows). Node E requires 7 receive buffer slots for ready flits and 5 for data flits from node D. This sums up to 12 receive buffer slots, which is even more than there are nodes in the example.

There are also some minor drawbacks at the software ready implementation: It is possible, though very unlikely, that some data flit has exactly the same payload as a ready flit – how can data and ready flits be distinguished in a safe way? Moreover, program execution is slowed down by flits that arrive and have to be handled, but are not needed yet.

Due to these reasons, ready synchronization should be implemented in hardware. Then, ready flits can be handled in a dedicated ready hardware component bypassing the normal receive buffer in the NI. This allows to minimize the number of receive buffer entries, which is highly desirable as buffers occupy a high amount of hardware logic [WGH+07]. Our new ready hardware component handles ready flits immediately when they arrive and provides a simple check mechanism for the sender to know whether some node is ready.

4.5.1. Hardware Implementation Considerations

At its basic principles, the hardware ready implementation works similar to the software ready implementation. The difference is a new ready hardware component to process ready flits independent from data flits. It processes ready flits, while data

flits are still stored in the receive buffer and handled by the PE. For the differentiation of ready and data flits, we introduce an additional 1-bit signal isReady everywhere between sender and receive logic. For data flits, it is 0 and for ready flits it is set to 1. In addition, the array storing information which nodes are ready is replaced by a hardware bit array.

When a data flit arrives, it is stored in the receive buffer just as before and when a ready flit arrives, the sender's corresponding bit in the hardware bit array is set to 1. Each node has its own hardware bit array where each bit corresponds to one particular node. When a node wants to send data to some other node, it can look up in its own hardware bit array if the corresponding node is ready.

All flits that arrive at the node are either needed by the PE (data which was requested and is now processed) or they are ready synchronization flits. The latter neither reach the receive buffer, nor the PE – they are processed in the NI and put in the hardware bit array. Therefore, it is avoided that flits which are currently not needed reach the PE. As a result, there is no need to distinguish between ready and data flits in the program code anymore. Furthermore, this is a scalable solution since each additional node only requires one bit of additional storage at each node.

4.5.2. New Instructions

Two new instructions are required for handling ready information: send ready (srdy) for sending ready flits and branch if not ready (bnr) to check if a specific node is marked as ready in the hardware bit array. They can be seen in Table 4.1.

Table 4.1.: Overview on our RISC-V instruction set extension for ready synchronization (originally published in [FSMU18]).

mnemonic	source register 1	source register 2	immediate value	function
srdy	*sender*			send ready flit to *sender*
bnr	*receiver*		*address*	check if *receiver* is ready; when it is not, jump to *address*

The receiver node sends a ready flit to the *sender* node via srdy. This instruction might be called some cycles before the receive instructions on the same node to reduce waiting times (the time between sending the ready flit and receiving the answer from the sender node). Since srdy is a modified snd instruction, it is necessary to check the status of the send buffer before executing it with bsf or bsnf. Should the send buffer be full when it is executed, srdy will raise an exception.

bnr is a branch instruction with the operand *receiver* and the immediate value *address*. It checks if *receiver* is ready to receive data and jumps to *address* if it is not. For

this purpose, the corresponding bit is checked in the hardware bit array. Following the status branch instructions in Section 3.4, `bnr` is also implemented as branch instruction for checking a status. Although branch instructions with side effects are uncommon, we designed `bnr` with one small side effect: When the receiver node is ready, the bit in the hardware bit array is reset to 0. This is necessary to ensure that at the next program part which requires synchronization a new `ready` flit is awaited before flits are sent. An alternative to this behavior would be a dedicated `reset` instruction. However, it may be forgotten and lead to time-consuming debugging by the software developer. Moreover, the `ready` information is not valid after the `bnr` instruction anymore, because then flit transfer starts and the receiver node has to handle these flits. Thus, the `ready` information should not be kept beyond the point where `bnr` is executed. Altogether, with the bit being reset at the `bnr` instruction, there is no need for other specialized instructions and code stays simple with only two additional instructions.

4.5.3. Implementation

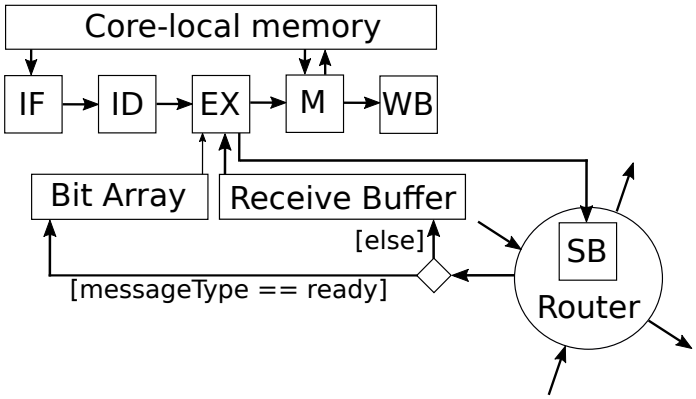


Figure 4.4.: Hardware structure of a node with `ready` synchronization: After the router, a distinction between `ready` and data flits is made. `ready` flits enable the bit of the sender node in the `Bit Array`, all other flits are stored in the `Receive Buffer`. The thin arrow between execute stage and `Bit Array` illustrates that only one bit is fetched from the `Bit Array`. It indicates whether a node is ready to receive data. Not illustrated is that the connection to the send buffer (`SB`) has to be extended by one bit to support the `ready messageType`.

The hardware implementation described in this chapter was first carried out by Sewing [Sew18] and later improved by Bitterlich and Unte [BU19]. As already explained in Subsection 4.5.1, an additional 1-bit signal `isReady` and a hardware bit array are to be implemented. At our VHDL model, flits are distinguished with an

enumeration type called messageType[7]. Instead of installing a seperate 1-bit signal isReady, we extend the messageType with an additional enumeration type ready. Since the original messageType required one bit for two types and now requires two bits for three types, it essentialy implements the same change as introducing an additional 1-bit signal. Architectural changes besides the messageType are illustrated in Figure 4.4: A node consists of a 5-stage pipeline with core-local memory, which is connected to the network router. Thereby, the send buffer is part of the router and the components between the 5-stage pipeline and the router are part of the execute stage (EX) to guarantee fast access to the Bit Array and the Receive Buffer. In contrast to Figure 3.4, we now added the hardware bit array, which is called Bit Array in the figure. It stores the ready status of other nodes at the corresponding position and as such has the same role as the ready_array in the software implementation (see Code examples 4.1 and 4.2). For accessing single bits in the Bit Array, a multiplexer has to be integrated (not seen in the figure). To determine whether an arriving flit should be given to the Bit Array or the Receive Buffer a case distinction based on the messageType is introduced.

Table 4.2.: Encoding of ready related instructions in our RISC-V instruction set extension.

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	00000	rs1	001	00000	1011011	srdy
imm[12|10:5]	00000	rs1	101	imm[4:1|11]	1111011	bnr

For handling ready flits and ready status information, our new instructions from Subsection 4.5.2 also have to be implemented. Their encoding is shown in Table 4.2, the opcodes are the same as for the PIMP instructions in Section 3.4.

srdy is a variation of the snd instruction with two differences: First, there is no payload, because a ready flit never reaches the PE. Second, the messageType is set to ready. When the ready flit arrives at the target node, it is not put into the receive buffer. Instead, the ready messageType allows the detection as ready flit. Then, the corresponding bit in the Bit Array is set to 1.

bnr follows the implementation of the other PIMP status related instructions in Section 3.4. In contrast to bsf, bsnf, bre and brne, bnr requires one source register for specifying the node which ready status is to be checked.

Some minor changes at the decode and execute stages are also necessary to

[7] At the original VHDL model, messageType comprises the types none and data. An overview over all messageType is given in Appendix B.1

support our new instructions.

4.5.4. Programming model

Code example 4.3 Send operation including hardware `ready` synchronization.

```
.globl send_data_hw_rdy
send_data_hw_rdy:           # function to send data with hardware ready
                            # expects receiver node in a0 ...
                            # ... address of data to send in a1 ...
                            # ... and amount of data to send in a2

    ld    t0, 0(a1)         # load data to send

wait4ready:
    bnr   a0, wait4ready    # wait for ready flit

wait4sendbuffer:
    bsf   wait4sendbuffer   # wait for free slot in send buffer
    snd   a0, t0            # send data to receiver node
    addi  a2, a2, -1        # less data to send
    beq   a2, zero, finish  # when all data sent => finish
    addi  a1, a1, 8         # point to next data word to send
    ld    t0, 0(a1)         # load next value to send from array
    j     wait4sendbuffer   # send next data word

finish:
    ret                     # return when all data is sent
```

Our Code examples 4.3 and 4.4 illustrate how easy hardware `ready` synchronization can be employed in software. The first Code example 4.3 shows a send operation with the same behavior as in Code example 4.1. However, there is no need for manually handling any incoming flits anymore. Therefore, the operation is much shorter and simpler. Basically, it is a variation of the `send` part of Code example 4.1. The main difference is the `bnr` instruction at `wait4ready` checking the `ready` status of the receiver node before waiting for the send buffer and sending a data flit. A few minor modifications were made to efficiently send out data flits. Altogether, the send operation now consists of only 10 assembly instructions.

At the second Code example 4.4 comprising the receive operation, changes are similar and it is reduced to 11 assembly instructions. In the beginning, the receive buffer is checked if it is empty. Since no `ready` flits can be in the receive buffer and all data flits should already be processed before, `brne` at `wait4freeRbuffer` should never jump to `rcv_data`, which remained for debugging purposes only. Therefore, execution should directly continue at `sendready`. Here, Code example 4.4 is similar to the same part in Code example 4.2. For sending out the `ready` flit, now the specialized instruction `srdy` is employed. Afterwards *all* incoming flits can directly be stored in the local memory as only flits from the corresponding sender node

Code example 4.4 Receive operation including hardware ready synchronization.

```
.globl receive_data_hw_rdy
receive_data_hw_rdy:        # function to receive data with hardware ready
                            # expects sender node in a0 ...
                            # ... address where to store received data in a1 ...
                            # ... and amount of data to receive in a2

# wait until receive buffer is empty
wait4freeRbuffer:
    brne rcv_data           # when receive buffer is not empty => data received

# send ready flit to sender
sendready:
    bsf  sendready          # wait until send buffer has free slots
    srdy a0                 # send ready flit to sender

# receive flits from sender
wait4receive:
    bre  wait4receive       # wait for flits in receive buffer
    rcvp t1                 # store payload of received flit in t1
    sd   t0, 0(a1)          # store received data in memory
    addi a2, a2, -1         # less data to receive
    beq  a2, zero, finish   # when all data received => finish
    addi a1, a1, 8          # next memory location to store received data
    j    wait4receive       # receive next data word

finish:
    ret                     # return when all data is received

# received a data flit
rcv_data:
# for debugging purposes only, should never occur
# when all nodes execute code with ready synchronization
```

arrive. As it is not necessary to distinguish between incoming ready and data flits anymore, the code for receiving flits is now much shorter and simpler. Like in Code example 4.3, some minor optimizations were also applied.

From the timing perspective, the parts for sending and receiving data (starting at wait4sendbuffer and wait4receive) have the same length and need the same time for execution.[8] On the first sight, bre might cause a branch delay when jumping to itself. However, it only jumps when the receive buffer is empty. When it is empty, there is no data to receive and the branch delay does not delay any data receipt. Should the receive buffer be filled with one or more flits, bre will never jump and therefore cause no delay as the next instruction is already in the pipeline.

[8] At the RC/MC, load and store instructions only take one cycle due to the local memory being realized as scratchpad. However, it is not necessary to have this short memory access time. It is sufficient when a store instruction is at least as fast as a load instruction. Otherwise, nops have to be inserted at the send operation to avoid that flits are sent faster than they are received.

bsf only jumps when the send buffer is full. This is the case when one period of the TDM schedule takes longer than the execution of the loop at wait4sendbuffer and several flits are sent. Because the TDM period is respected in the timing analysis of the NoC, this is no problem.

At the software ready implementation in Code examples 4.1 and 4.2, receiving data takes longer than sending data. Thus, the receive buffer might still run full when a lot of data is to be transmitted. However, the hardware solution delivers a better performance at low hardware cost and avoids this problem. Therefore, the hardware version of our ready synchronization should always be preferred.[9]

4.5.5. Expected Hardware Costs

Ready flit handling requires additional hardware logic: In the focus is the hardware bit array, how it is integrated in the execute stage and how it processes incoming ready flits. It is the largest part of our ready hardware component at the nodes and grows with the number of nodes in the NoC. However, it scales very well as it only grows with one bit per additional node. Additionally, it needs a multiplexer to access the single bits. Further logic is required for the additional ready messageType, which needs 1 additional bit for the messageType and has to be passed through from the senders' send buffer to the receiver. Other signals comprise the flits' source, destination and payload (64 bit payload, 8+8 bit for sender and receiver information in a 16x16 NoC). Thereby, the overhead is only 1.25% (in a 16x16 NoC) compared to the other information that is sent through the NoC. Moreover, the additional messageType requires an (extended) multiplexer to distinguish where an arrived flit goes. Finally, the additional send instruction srdy and the branch instruction bnr have to be provided. However, srdy does the same like the regular snd besides setting the ready messageType. This overhead should be negligible. bnr works similar to the PIMP status related branch instructions and should also induce rather low overhead.

The additional logic might pay itself by saving buffer slots: Since the PE does not need to handle unwanted flits anymore, processing times are reduced. The received data flits can immediately be processed. Therefore, fewer buffer slots should be necessary. But most promising seems that received ready flits do not enter the receive buffer due to their handling in hardware. Thus, the receive buffer is only needed for data flits and does not have to carry any synchronization flits. Altogether, the load in the receive buffer should be drastically reduced and the PE can focus on flits it currently wants to process.

[9] Adding nops at the software send operation would avoid that the receive buffer could run full at the receiver node, but would also decrease performance.

4.6. Evaluation

For the evaluation, we consider an RC/MC processor as described in Chapter 3 with 2x2 = 4 nodes and the generic TDM schedule 1:A. We compare four different programs with hardware and software `ready` synchronization.

First, we compare the effort for the Code examples 4.1 to 4.4 in Subsection 4.6.1. In Subsection 4.6.2, we execute several small programs with software and hardware `ready` synchronization. Afterwards, we check how many receive buffer slots are required for these small programs in Subsection 4.6.3. Finally, in Subsection 4.6.4, we compare the actual hardware costs of our hardware `ready` synchronization with the costs of buffer slots.

4.6.1. Comparison of Software and Hardware Implementation Effort

In the previous sections, we already presented code examples for sending and receiving flits: On the one hand, Code examples 4.1 and 4.2 illustrate how `ready` synchronization would be realized as software implementation without hardware support. Thereby, all nodes maintain a software `ready_array`. Always when flits are received, a distinction between `ready` and data flits has to be made and the software `ready` array has to be updated in the first case. Although the receive function should only be called when all previous data transmissions are finished, it first has to check the receive buffer for `ready` flits before sending out its own `ready` flit.

On the other hand, Code examples 4.3 and 4.4 show the minimalistic effort of `ready` synchronization with hardware support: After checking the `ready` state of the receiver node (and waiting for it if necessary), the sender can directly start sending flits. At the receiver node, the receive buffer should always be empty when starting with the function. Thus, it can send its `ready` flit to the sender node early. Afterwards, the receive operation focuses on the necessary minimum of instructions (7 assembly instructions in a loop).

Altogether, our hardware `ready` implementation saves execution cycles and simplifies code, which is beneficial e.g. for maintainability and code analysis. Moreover, sending and receiving flits takes equal cycles with hardware support. Thus, it completely avoids buffer overflows and delivers better performance and timing predictability.

4.6.2. Execution Times

We executed several programs where four nodes work together, e.g. for exchanging data. In the following, we compare software versus hardware implementation of `ready` synchronization. Both are implemented in the way we described in the previous sections. Our programs are (i) four nodes meeting at a barrier and a (ii)

broadcast of 1280 values from one node to three other nodes.[10] Furthermore, we have an (iii) All-to-All broadcast, i.e. all nodes broadcast one 64-bit value to all other nodes. Finally, we have (iv) a global reduce operation, where a global sum is computed from values coming from all nodes.

These programs are small building blocks used in distributed memory programs. The more these building blocks are employed in programs, the bigger is the effect. We do not compare with an implementation without `ready` synchronization as this would lead to buffer overflow.

Table 4.3.: Overview on benchmarks and their execution times.

Name	Software `Ready` [cycles]	Hardware `Ready` [cycles]	Saving
Barrier	237	116	51%
Broadcast	29 764	18 124	39%
All-to-All broadcast	1 167	1 007	14%
Reduce, global sum	1 208	1 055	13%

Table 4.3 gives an overview of our results. In comparison to the software implementation, the execution time of all programs is reduced in the hardware implementation. The savings reach from 13% to 51%. As Code examples 4.1 to 4.4 already illustrate, less code is executed. It should be noted that no `ready` flits from non-participating nodes interfere these executions. When there would be some, they would interrupt the software `ready` implementation and increase its execution times. However, they would have no impact on the hardware `ready` execution times, because they are handled by our specialized hardware.

4.6.3. Saving of Buffer Slots

Now, we check how many slots in the receive buffer are required at least to avoid buffer overflows. For this, we execute the above programs in the MacSim and decrease the receive buffer size until a buffer overflow occurs. The results can be seen in Table 4.4. Thereby, the numbers at *No Synchronization* represent the buffer space needed in the worst case when there is no synchronization present. It results from the maximum number of flits sent by other nodes to one node. Thereby, it is assumed that all nodes send all their flits at the same time and the receiving node is not yet ready to process them. It practice, this would require buffers of an unlimited

[10] A broadcast operation with one flit would result in numbers similar to the Barrier and All-to-All broadcast. Therefore, we took a larger broadcast to give an idea about what happens when lots of data is transmitted.

size. Thus, these values are of theoretical kind.

Table 4.4.: Overview on required receive buffer slots

Name	No Sync. [buffer slots]	Software `Ready` [buffer slots]	Hardware `Ready` [buffer slots]	Saving
Barrier	2	1	1	0%
Broadcast	1280	251	1	>99%
All-to-All broadcast	3	2	2	0%
Reduce, global sum	2	2	1	50%

Hardware `ready` flits do not occupy receive buffer slots. Therefore, less or in the worst case equal receive buffer slots are required compared to the software `ready` implementation. As before, there are no other nodes sending interfering `ready` flits. Otherwise, more buffer slots would be required in the software implementation. At the barrier and reduce program, only two buffer slots are occupied without synchronization. This is because of the implementation of these programs, where tree-based algorithms are employed. At the broadcast implementation, 1280 64-bit values are to be broadcasted, which are 1280 flits. As can be seen in Table 4.4, a lot of buffer slots can be saved. The reason is the implementation: It is not one node sending flits to all other nodes. Instead, the broadcast operation is distributed in the network as a binary tree. Thus, there are intermediate nodes having to receive data and forward it to other nodes. In the software `ready` implementation, they need too much time to process flits. New flits arrive faster than the old ones are processed. Therefore, more and more flits retain at the intermediate nodes. In the hardware `ready` implementation, code parts are shorter and processing can take place faster. Therefore, it is avoided that buffers run full.

4.6.4. Actual Hardware Costs

In our original publication [FSMU18], we presented the numbers from the implementation by Sewing [Sew18]. To have a consistant view with the other hardware extensions presented later in Chapters 5 and 6, we now include the numbers from Bitterlich and Unte [BU19]. They integrated `ready` synchronization in our VHDL model and carried out a synthesis as described in Section 3.5. For 2x2 and 4x4 nodes, around 15 ALMs and 17 registers are required per node. Thereby, the total costs for a node are roughly 2200 ALMs and 2600 registers per node for 2x2 nodes and 2900 ALMs and 3400 registers for 4x4 nodes. This means that `ready` synchronization imposes an overhead of less than 1%. Moreover, the costs for one receive buffer

slot in the design are around 50 ALMs and 60 registers per node. Therefore, saving only one receive buffer slot already pays the hardware logic needed for `ready` synchronization.

4.7. Conclusion

Besides the TDM schedule, the RC/MC architecture in Chapter 3 imposes no limitations when a node sends flits to other nodes. Therefore, flits from several nodes could arrive at any time and have to be stored and handled. This requires large receive buffers and free computational capacity at the node receiving these flits. To avoid this, data flits should only be sent when the receiver node is ready to process them. Thus, a systematic synchronization mechanism should be applied. To keep network utilization low, we presented `ready` synchronization requiring only one flit. The `ready` flit is sent from the node that wants to receive data to the intended sender node. The sender node does not start sending until the `ready` flit arrives. When it arrives, it knows that the receiver node is ready to handle incoming flits and starts sending. This simple principle can be implemented in software, but is less performant than an implementation in hardware. For an efficient hardware implementation, we added two new instructions `srdy` and `bnr`, a `ready` `messageType` and a hardware bit array (plus management logic) at the receiver node. The latter allows to check if a specific node is ready to receive flits. Our evaluation shows that global communication operations execute 10-50% faster and need fewer receive buffer slots. By saving buffer slots, our hardware `ready` synchronization pays itself as it requires only a small amount of additional logic.

5

Hardware Broadcast/Multicast Extension to Improve Schedule One-to-All

Abstract. Broadcast and multicast are communication operations needed often in parallel programs. They are important for data exchange in distributed computations. When using the 1:A schedule, it takes several periods until data is broadcasted/multicasted to all (participating) nodes in the NoC, because of the restriction to send at most one flit per period. Therefore, we exploit the reserved slots in the 1:A schedule to enable broadcasting of flits to *all* nodes within one period. Furthermore, we extend our approach to hardware-supported multicast.

5.1. Introduction and Basic Idea

One of the shortcomings of the One-to-All schedule is the limitation that only one flit can be sent each period. Therefore, the execution of a broadcast or multicast operation takes several periods because communication has to be organized e.g. as binary tree in software. A binary tree is currently the most efficient way to implement these operations in real-time software according to Stegmeier et al. [SFMU18]. Thereby, it would take up to $(\lceil log_2(\chi+1)\rceil - 1)\cdot 2$ periods with a (full) binary tree

for broadcasting data to χ nodes (χ includes the sending node)[1] For example, it takes up to 8 periods to reach all nodes in a 4x4 NoC or 12 periods in a 8x8 NoC. We propose an extension which enables to send one flit to *all* other nodes within *one* period, realizing a broadcast operation. Moreover, we extend it to a hardware-supported multicast operation. This makes the One-to-All schedule an efficient alternative to the All-to-All schedule.

Our approach exploits the property of the One-to-All schedule that paths from a node are reserved to send a flit to *any* other node. Our idea is to send a flit to the farthermost node and the routers on the way make a copy of it and forward it to the other nodes. Since all of these paths were reserved for the flit to be sent from one node and the receiving nodes await flits from *any* node, there are no collisions and no conflicts. The flits only have to be marked as broadcast/multicast flits to be recognized by routers to copy them instead of just forwarding them. Based on this idea we implement hardware-based broadcast and multicast operations in this chapter.

The remainder is structured as follows: In the next Section 5.2, related work is presented. Afterwards, in Section 5.3, the concept and implementation considerations of the hardware-supported broadcast operation and in Section 5.4 of the hardware-supported multicast operation are described. Section 5.5 illustrates our actual implementation and Section 5.6 the programming model. Then, in Section 5.7 expected and real hardware costs are contrasted with each other. An evaluation takes place in Section 5.8. Finally, the results are concluded in Section 5.9.

5.2. Related Work

Research on hardware support for broadcasts/multicasts focuses typically on optimizing the *average-case execution time (ACET)* for high-performance systems and supercomputers. Thereby, communication between nodes usually does not follow a schedule, but flits/messages are forwarded as soon as possible (best effort).

One idea for multicast is to send the flits on the Hamiltonian path, i.e. a path in a graph where each vertex (here: node) is visited exactly once: Lin and Ni [LN91] propose multicast wormhole routing in multicomputer networks based on the Hamiltonian path. Thereby, either two messages (dual-path multicast routing) or several messages (multi-path multicast routing) are sent to reach many nodes in a short time. To realize multicast, messages always have to follow Hamiltonian paths.

Another idea is to send a flit through the NoC and nodes make copies of it: Panda

[1] log_2 comes from determining the depth of the tree, $\chi + 1$, because a binary tree is already full with $2^y - 1$ nodes. -1 compensates the rounding operation and $\cdot 2$ states that data has to be sent to two childs for each intermediate and the root node.

et al. [PSK99] realize multicast by adding one header for each node which should receive the message. The message is first routed to the first node, where the first header is removed. Then, the message is sent to the next node, as defined by the next header. This way, the message is constantly forwarded and additional headers are removed until all headers are worked off.

Both ideas do not work in the real-time field because the WCTT would be very high due to very long paths. Moreover, preparing these messages consumes a lot of time.

Other approaches integrate broadcast/multicast support directly in their architecture, avoiding delays by long paths:

The BlueGene/L torus interconnection network [ABC+05] is organized as a three-dimensional torus. It supports broadcasting of messages in x-, y- or z-direction. However, it is limited to only one of these directions as multi-dimensional broadcasting would have greatly complicated the logic.

Krishna et al. [KKC+08] developed a many-core with two 2D mesh NoCs: One NoC for data and one NoC for control signals. In the control signal NoC *global interconnection lines (G-Lines)* are employed for broadcasting control signals. To enable broadcasting in one cycle, $2 \cdot n$ G-Lines per direction per row/column are needed (e.g. 16 G-Lines for a 8x8 NoC). However, broadcasting only works for control signals like virtual channel signaling and buffer signaling.

As already illustrated in Chapter 2, several real-time many-core architectures are being developed. Nevertheless, no literature was found on hardware broadcast or multicast taking real-time aspects into account.

5.3. Concept: Hardware-supported Broadcast Operation

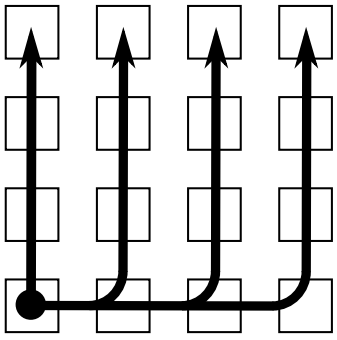


Figure 5.1.: For the node on the left bottom being sender, the paths denoted with arrows are reserved for flit traversal at each period of the One-to-All schedule.

At the 1:A schedule, each node can usually send only one flit per period to *one*

other node, but receive flits from *all* other nodes [MU14]. As can be seen in Figure 5.1, several paths through the NoC are reserved to reach *any* other node at the sending operation. This means that in x-direction the path to the farthermost node is free and in y-direction also (besides other paths).

While a flit takes this way to the farthermost node, each of the routers on x-direction can make a copy while forwarding the flit. This copy can be forwarded in y-direction, where the remaining routers also make a copy. In this way, all nodes can receive the flit although it is sent out only once – it is broadcasted.

For detecting that a flit is a broadcast flit and has to be copied, an additional 1-bit signal `copyFlit` has to be integrated between the nodes and in the send buffer. On architectures where a header flit always has to be sent first, the header flit should be extended with this information.

Table 5.1.: Overview on our RISC-V instruction set extension for broadcasts and multicasts.

mnemonic	source register 1	source register 2	immediate value	function
`mcst`	*message*			broadcast/multicast a 64 bit *message*
`bnar`			*address*	branch to *address* when not *all* nodes are ready yet
`bnra`	*part*	*nodes*	*address*	branch to *address* when the nodes in the indicated *part* of the `Ready Bit Array` are not yet `ready` *part* has to be 0 when up to 64 nodes are present
`mrdy`	*sender*	*number*		send `ready` flit to *sender* and store *sender* and *number* at NI to accept only *number* multicast flits from *sender*

An overview on the new instructions is given in Table 5.1. To send a broadcast flit, a new assembly instruction `mcst` *message* is introduced, where *message* is the data to be broadcasted. This new instruction works like a normal `snd` instruction with the difference that no receiver is to be provided (because it is a broadcast operation) and that it sets the new 1-bit signal `copyFlit`. We already called the mnemonic `mcst` (multicast), because at our implementation in Section 5.5 we will finally implement broadcast as special case of multicast. Like at `snd` and `srdy` instructions, `mcst` will raise an exception when the send buffer is full when it is executed. Therefore, it is

required to check the status of the send buffer before with `bsf` or `bsnf`.

Keeping in mind that flits should only be sent when the receiving nodes are ready (see Chapter 4), a new instruction `bnar (branch if not all ready)` is introduced: It works like `bnr (branch if not ready`, see Subsection 4.5.2), but checks if *all* nodes are ready. In contrast, the original `bnr` instruction only checks if a single node is ready. Sending a broadcast flit without the `bnar` instruction would require to do as many `bnr` checks as there are nodes.

5.4. Concept: Hardware-supported Multicast Operation

While a broadcast flit is always sent to *all* other nodes, a multicast flit only addresses a fraction of the nodes as receivers.

In theory, the broadcast flit could be sent to only a subset of the nodes on the chip by not sending it to the farthermost node but some other node. This would span a rectangle between the sender and target node, where all nodes within the rectangle are receiver nodes. However, this approach makes everything highly placement dependent, which we want to avoid to retain composability. Moreover, this would only allow multicasts to single rows or columns or nodes arranged in rectangular positions.[2]

Therefore, we intend that even at a multicast operation, the flit is sent to all nodes, but not all nodes will accept it.[3] Only those nodes which wait for a flit will perform a receive operation, the others just deflect it: At the NI of each node it is checked if a broadcast flit from this sender is to be received. When it is, it is handled like a normal data flit, otherwise it is deflected. On architectures where header flits are always sent before data flits, information about receivers could be saved in the header flit. However, this approach is limited: When the information if a node should receive the message or not only requires one bit per node, the complete size of a header flit would already be filled at 32 nodes (32-bit flits) or 64 nodes (64-bit flits), respectively. We intend our approach to work with many cores and therefore with more than 64 nodes.

Thus, each receiving node configures on its own which multicast flits it wants to accept and how many. With a new assembly instruction `mrdy` *sender*, *number* (send

[2] Indeed, Walter realized hardware barriers for nodes in rectangular positions exploiting a similar property in the 1:1 schedule [Wal19].

[3] This requires that all receivers know who sends flits they want. In operations like `MPI_Bcast` (see Subsection 3.3.2), there is a parameter specifying the node sending the broadcast. Furthermore, our programming model assumes that the same program code is executed simultaneously on all participating nodes (see Section 3.3). Thus, there will be case distinctions at the broadcast operations to decide who sends flits and who receives them. Then, it will also be clear for receiving nodes whom they await flits from.

multicast `ready`), each node can store from which sender multicast flits should be accepted (parameter *sender*) and how many (parameter *number*). It extends the `srdy` instruction as it collects the same information (besides additional information about the number of flits to receive) and the sender should not start before a `ready` flit was sent. Every time when a multicast flit comes by, it is forwarded from the router to the NI. There, the sender of the flit is compared with the *sender* stored. When the flit comes from the desired sender, it is stored locally, otherwise it is deflected. In the first case, the counter is decremented. When the counter reaches zero, all further incoming multicast flits are deflected, because the node does not expect any more multicast flits.

The only remaining problem occurs when the same node sends out several multicasts to different groups of participants. These multicasts may be mixed up, because all nodes accept multicast flits passing by from the desired sender, irrespective if they are intended for these receivers or not. To overcome this, nodes should meet at a barrier before and after a multicast operation to have a clear separation of different communication parts (which is also in accordance with the BSP model, see Section 3.3). Because barriers might be rated as costly in regard to communication effort and required time, we will present a hardware-supported barrier in the next Chapter 6.

In the case that a node receives multicast flits from different nodes in the same period (e.g. collecting results from a distributed computation), comparing only one sender is not sufficient. Thus, the NI might be extended to manage and count several senders of multicast flits. The maximum number of possible senders to be managed at the same time is upper bounded by the number of receive buffer slots because in the worst case all multicast flits might be received in the same period. For the comparison to still take place in one cycle, additional comparators have to be added in hardware. However, this largely complicates hardware logic. Therefore, we only support accepting multicast flits from one sender at a time in our implementation.

5.5. Hardware Implementation

In his master's thesis [Aue18], Auer implemented a first prototype of hardware-supported broadcast for the 1:A schedule without multicast support. His results indicate that our basic ideas work. Based on the experiences made in this work, the implementation in this section was elaborated. It extends the RC/MC processor as it was described in Chapter 3 and extended with `ready` synchronization (Chapter 4). The actual implementation in the VHDL model and the MacSim was carried out by Bitterlich and Unte [BU19].

Like in Subsection 4.5.3, we extend the enumeration `messageType` with an additional type `mcst` instead of an additional 1-bit signal. For sending a broadcast/mul-

Table 5.2.: Encoding of broadcast/multicast related instructions in our RISC-V instruction set extension.

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	101	00000	1011011	`mrdy`
0000000	00000	rs1	110	00000	1011011	`mcst`
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1111011	`bnra`
imm[12|10:5]	00000	00000	111	imm[4:1|11]	1111011	`bnar`

ticast flit, a new instruction `mcst rs1` (multicast) broadcasts/multicasts the data in register `rs1`. Thereby, broadcast is a multicast which is accepted by all nodes. An overview on the encoding of our new instructions related to broadcast/multicast is given in Table 5.2. To be able to realize a simple broadcast operation with only two assembly instructions, we introduce a new instruction `bnar imm`. It implements `branch if not all ready` as described in Section 5.3 (check if *all* nodes are marked as `ready`).

For multicasts, the new instruction `bnra rs1, rs2, imm` (`branch if not ready array`) is utilized. It compares the part of the `Ready Bit Array` which is addressed by `rs1` with the bits in `rs2` (the 64 bits in `rs2` represent the ready state of up to 64 nodes). When the `Ready Bit Array` is e.g. 256 bits (256 nodes), the value in rs1 could be 0, 1, 2 or 3 to address the first, second, third or last block of 64 bits in the 256 bit `Ready Bit Array`, see example in Figure 5.2. The first block represents the nodes 0 to 63, the second 64 to 127 and so on. In this way, the state of 256 nodes can be checked with only four `bnra` calls.When the `Ready Bit Array` is only up to 64 bits wide (representing up to 64 nodes), rs1 always has to be zero[4]. Because the number of nodes is not necessarily a multiple of 64 nodes, the last block may not completely be filled with bits representing nodes. These "undefined bits" will never return `ready`, because there should never be sent any flits to nodes that do not exist.

At the router, all flits with `messageType` `mcst` have to be copied. Then, they are forwarded to the north and to the west. Thereby, the `x` target address of flits copied to the north is set to the current column to signalize that they have arrived in the target column. The architectural design of an extended node can be seen in Figure 5.3: We combined the logic handling incoming `mcst` flits within the module `multicast filter`[5], which is a further extension of the execute stage. All incoming `mcst` flits

[4]Should the program running on the processor access a block that does not exist, this may lead to undefined behavior.

[5]In the VHDL model, there is no additional module `multicast filter` – there, only 2 additional `ifs` are required. We added the `multicast filter` module in the figure to give a better overview.

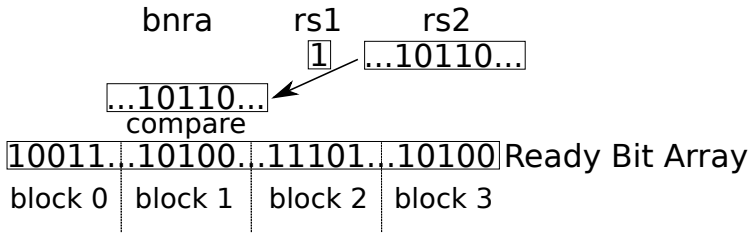


Figure 5.2.: Example of the `bnra (branch if not ready array)` instruction when the `Ready Bit Array` has four blocks: The value `1` in register `rs1` indicates that the bits in register `rs2` are to be compared with block 1 of the `Ready Bit Array`. Since our architecture has 64 bit registers, we would compare 64 bits in this example, representing the `ready` state of 64 nodes.

have to fulfill two requirements to reach the `receive` buffer: First, they have to come from the desired sender node, which is checked via the `multicastSender == flit.sender` statement. Second, it is counted if enough flits from the desired sender have already been received. The statement `multicastCount > 0` checks if more flits are wanted. If so, the received `mcst` flit reaches the `receive buffer` and is handled like a normal data flit there. In this case, the `multicastCount` register has to be decremented. It is utilized to count the multicast flits received and stop receiving when all multicast flits from the desired sender have arrived. To keep the hardware effort low, only receiving of broadcast/multicast flits from one sender is supported at a time. For handling our new instructions, minor changes in the decode and execute stages are also necessary.

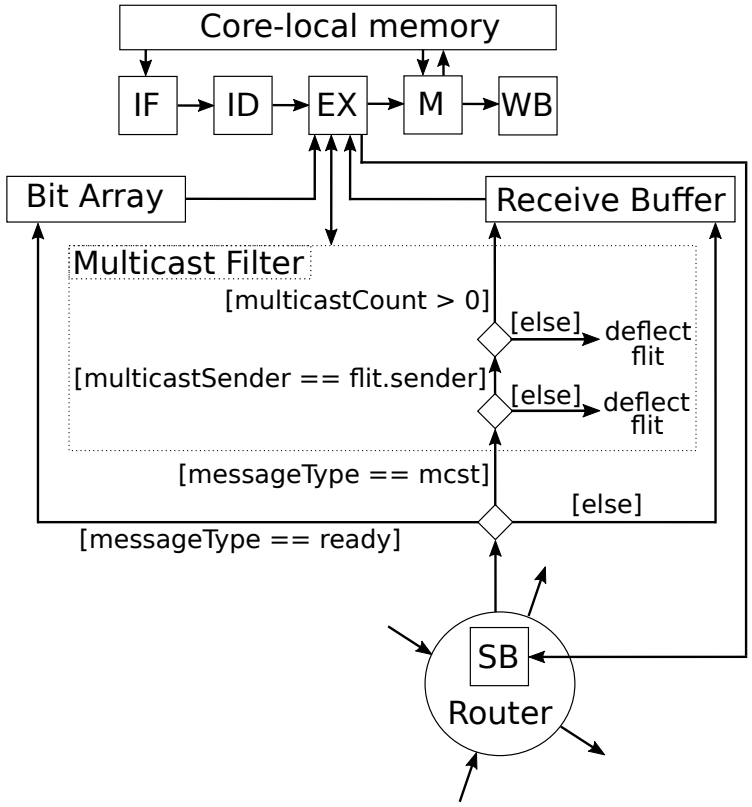


Figure 5.3.: Hardware structure of a node, extended with a `multicast filter` and a further case distinction for arriving flits. When they have `messageType` `mcst`, arriving flits are given to the `multicast filter`. It checks whether the flit comes from a source where a multicast flit is awaited from and if it is one of the desired flits. Not shown are the extensions in the decode and execute stages.

5.6. Programming Model

With our new hardware extension, a broadcast can be handled with only a few lines of assembly code using the instructions shown in Table 5.1. In Code example 5.1, we create a small function to broadcast an array and it is received in Code example 5.2.

For sending a broadcast (Code example 5.1), the address of the array to be sent has to be provided as parameter as well as the number of values to be sent. In the function, first it has to be checked whether *all* nodes are marked as `ready` in the `Ready Bit Array`. When they are, the loop processing the array can start with its first iteration: The first 64 bit value is loaded from the array. Then, we check if the send buffer has a free buffer slot and spin if necessary before broadcasting the first 64 bit value. Afterwards, the pointer to the array is redirected to the next value and

Code example 5.1 Broadcast on sender side.

```
.globl send_broadcast
send_broadcast:             # function to broadcast values from an array
                            # expects address of array in a0 ...
                            # ... and number of values in a1
check:  bnar check          # check if all nodes are marked as ready
loop:   ld   t0,0(a0)       # load value from array
w4sb:   bsf  w4sb           # wait for send buffer before sending value
        mcst t0             # broadcast loaded value
        addi a0,a0,8        # address of next value
        addi a1,a1,-1       # counter: one value less to process
        blt  zero,a1,loop   # next iteration with next value
finish: ret                 # otherwise finished => return
```

the value counter is decremented. When the value counter has reached zero, the function is finished and returns. Otherwise, the loop is entered again to proceed with the next value in the array. A further check for the `ready` state is not necessary because the receiving nodes already indicated that they are ready and able to process all incoming data.

Code example 5.2 Broadcast on receiver side.

```
.globl receive_broadcast
receive_broadcast:          # function to receive broadcast values
                            # and store them in an array
                            # expects number of sender node in a0 ...
                            # ... address of array to store in a1 ...
                            # ... and number of values in a2
wait4sb: bsf  wait4sb       # wait for send buffer before sending ready
         mrdy a0,a2         # send ready to sender node and store sender
                            # node and number of values to receive
loop:    bre loop           # wait as long as receive buffer is empty
         rcvp t0            # receive value ...
         sd   t0,0(a1)      # ... and store it in array
         addi a1,a1,8       # next place in array
         addi a2,a2,-1      # counter: one value less to process
         bne  a2,zero,loop  # continue with next value or ...
         ret                # ... return when there are no more values
```

Receiving a broadcast works in a similar way as sending and is illustrated in Code example 5.2. The function expects three parameters: The number of the node to receive values from, the address of the array where to store them and how many values are to be received.

First, we wait for the send buffer to have a free buffer slot. Now, we can start with

sending a `ready` flit to the sender node. At the same time, the number of the sender node as well as the number of flits to be received are stored in the NI (all done with the `mrdy` instruction). Then, the loop to process values starts: After checking and waiting for flits in the receive buffer, the received value is taken out from there and stored in the array. Afterwards, the pointer to the array and the value counter are updated. When there are more values to be received, the loop is entered again, otherwise the function is finished and returns.

Note that the `bre` (branch if the receive buffer is empty) instruction is part of the loop because it cannot be ensured that the next flit has already arrived when the loop starts its next iteration.

Both operations have an equal length at the loop (1 PIMP communication operation, 1 PIMP status branch, 2 `add` operations, 1 memory access and 1 branch to the beginning of the loop) to ensure that the receive broadcast operation is as fast as the send broadcast operation. To avoid buffers running full or needing a further `ready` synchronization, receive operations should always perform faster or at least as fast as send operations. This also means that the WCET of the receive operation must not be larger than the *best-case execution time (BCET)* of the corresponding send operation.

From the timing perspective, both Code examples 5.1 and 5.2 provide good timing analyzability: `bnar` is called as long as not *all* nodes are `ready`. This means, it realizes an implicit barrier. It can be upper bounded by the time required by the other nodes to send a `ready` flit via `mrdy`. `ld`, `mcst` and `add` instructions have a fixed latency of one cycle each. On the receiver side, the same holds for `rcvp`, `sd` and `add` instructions. However, before these instructions, we have a `bre` branch waiting until a flit arrives in the receive buffer. The WCTT for the flit transfer between sender and receiver can be estimated with the network schedule. At the end of the loop, the loop iteration branch is dependent on the number of values to be broadcasted/multicasted, which has to be provided by the software developer for timing analysis. In both functions, the loops need six cycles for execution with additional two cycles branch delay for each value to be transmitted. This holds both for the best and the worst case, because the instructions are always worked off in the same order and there are no variations at their execution times. The only variation would be when `bre` (branch if receive buffer is empty) branches: There is a branch delay of three cycles. But `bre` branches only when there are no flits in the receive buffer. Thus, it cannot occur that flits in the receive buffer pile up due to branch delays of `bre`.

Multicast on the sender side works almost the same way as broadcast: Instead of checking if *all* nodes are `ready`, only selected nodes are checked. In Code example 5.3,

Code example 5.3 Example for multicast on sender side for a 256 node RC/MC processor.

```
set_parameters_for_multicast:
# for this example, we address nodes 49-73 and 189-231
# block 0: 49-63, block 1: 64-73, block 2: 189-191, block 3: 192-231
addi t0,zero,-1      # set all bits in t0
srli a2,t0,49        # set bits 49 to 63 in register for block 0
slli a3,t0,54        # set bits 0 to 9 in register for block 1
srli a4,t0,61        # set bits 61 to 63 in register for block 2
slli a5,t0,39        # set bits 0 to 39 in register for block 3
jal  send_multicast # call function send_multicast


.globl send_multicast
send_multicast:              # function to multicast values from an array
                             # expects address of array in a0 ...
                             # ... number of values in a1 and ...
                             # ... and targeted nodes in a2 - a5
check0: bnra zero,a2,check0 # check if nodes in block 0 are ready
        addi t0,zero,1       # store 1 to access block 1 at bnra
check1: bnra t0,a3,check1    # check if nodes in block 1 are ready
        addi t0,zero,2       # store 2 to access block 2 at bnra
check2: bnra t0,a4,check2    # check if nodes in block 2 are ready
        addi t0,zero,3       # store 3 to access block 3 at bnra
check3: bnra t0,a5,check3    # check if nodes in block 3 are ready
loop:   ld   t0,0(a0)        # load value from array
waitsb: bsf  waitsb          # wait for free slot in send buffer
        mcst t0              # broadcast loaded value
        addi a0,a0,8         # address of next value
        addi a1,a1,-1        # counter: one value less to process
        blt  zero,a1,loop    # go on with next value
finish: ret                  # otherwise finished => return
```

we have an RC/MC processor with 256 nodes and select nodes 49 to 73 and 189 to 231 to multicast to. For selecting nodes, we adapted the broadcast function to accept four more registers as parameters. Furthermore, it does not use the `bnar (branch if not all ready)` instruction anymore, but the `bnra (branch if not ready array)` instruction. The remainder after checking the `Ready Bit Array` remains the same code as in the broadcast Code example 5.1. When we consider a processor with far more than 256 cores, the `a` registers are not sufficient anymore to hand over all parameters (especially the registers defining the required `ready` bits). Then, the receiver nodes may be selected in an array whose address is handed over to the function.

On the receiver side, multicast works exactly the same way as in the broadcast Code example 5.2. Because the loops for data exchange are the same as in the

broadcast example, the timing considerations still hold. Moreover, we recommend to add a barrier after the broadcast/multicast operation is finished, in accordance with the BSP model as described in Section 3.3.

5.7. Hardware Costs

In Subsection 5.7.1, we discuss the impact of our implementation from Section 5.5 on hardware costs. Then, in Subsection 5.7.2, we have a look on the actual hardware costs.

5.7.1. Expected Impact on Hardware Costs

Our hardware broadcast/multicast extension is designed with the goal of low additional hardware effort. At the hardware level, the extensions as described in Section 5.5 have to be implemented at each node and result in the following overhead:

The additional `messageType` `mcst` comes for free, because the `messageType` requires two bits and up to now only represents three states (`none`, `data`, `ready`). Therefore, the `mcst` `messageType` can utilize the free fourth state. Our additional send instruction `mcst` should impose negligible hardware effort as it does the same like a `snd` instruction besides setting the `mcst` `messageType`. However, it needs to know which is the node with the farthermost distance. To avoid complex hardware computing this node, it is computed at synthesis and its node id is saved in a register. For multicast support, some additional logic is added for the decision if a flit is to be accepted or deflected. This includes the registers `multicastSender` and `multicastCount`, the comparator and the decrementer. These seem to be the most expensive components of the multicast extension, although their price should be within reason. Resigning on multicast and implementing only broadcast would be the higher price, because only having broadcasts would prevent the possibility to execute different applications concurrently (on different nodes) on our platform. The additional instructions `bnar` and `bnra` require direct access to the hardware bit array to access 64 bits at a time instead of only one bit (at `bnr`). This should only impose small overhead as the hardware bit array is already part of the execute stage and stage-internal access ways are short.

Altogether, the need for additional hardware logic should be small.

5.7.2. Actual Hardware Costs

The described concepts were integrated into the existing VHDL model and synthesized for a Stratix V E FPGA by Bitterlich and Unte [BU19] as described in Section 3.5. Our broadcast/multicast extension requires around 300 additional ALMs and 140

registers per node (for RC/MC processors with 2x2 and 4x4 nodes). Thereby, around 170 ALMs and 50-60 registers are needed per node to realize the `bnar` and `bnra` instructions. The total costs of a node (including `ready` synchronization and our broadcast/multicast extension) in a 2x2 node RC/MC processor are 2483 ALMs and 2703 registers or 3223 ALMs and 3561 registers when having 4x4 nodes on the chip. Thus, we have an overhead of around 10% at the ALMs and 5% at the registers. Although the hardware costs are not too high, we expected them to be a little bit lower. Since `bnar` is only needed for a special case and may only save a few hundred cycles in the worst-case, it may be cut out to save hardware costs. Together with the the `bnra` instruction, `bnar` almost contributes half of the costs of the hardware broadcast/multicast extension. Maybe accessing so many `ready` bits at once cannot be implemented in an efficient way on an FPGA. However, since our hardware broadcast/multicast extension might have a high impact on the (worst-case) performance, it might be worth its extra costs.

5.8. Evaluation: Worst-Case Performance

For the evaluation, we carry out timing analyses for different implementations of `MPI_Bcast`. In this operation, one node broadcasts one or several values to other nodes. It is defined in the MPI standard [Mes15] and was already introduced in Subsection 3.3.2. For determining the receivers, a *communicator* is utilized. It defines which nodes belong together at communication. Typically, communicators are created in the initialization part of a real-time program. This saves time and allows to prepare communication parameters at a non-critical point of time. During program execution, the communicator is handed over to collective operations to express which nodes communicate with each other. `MPI_Bcast` is a typical operation in parallel programs to exchange intermediate or final results along several nodes.

In the next Subsection 5.8.1, we carry out a timing analysis of different implementations of `MPI_Bcast`. Afterwards, we compare their worst-case performance in Subsection 5.8.2. Because the timing analysis might have introduced pessimism and overestimation, we assume optimal code optimization at the theoretical comparison in Subsection 5.8.3.

5.8.1. Timing Analysis of Different `MPI_Bcast` Implementations

We first analyze `MPI_Bcast` with hardware support and then a tree-based implementation.

Timing Analysis of `MPI_Bcast` with Hardware Support

`MPI_Bcast` with hardware support works as illustrated in Figure 5.4 and described in Table 5.3. We illustrated only one participant node – all others behave exactly the same way and have the same timing. The operation works in two phases: First, `ready` information is collected and then data is multicasted.

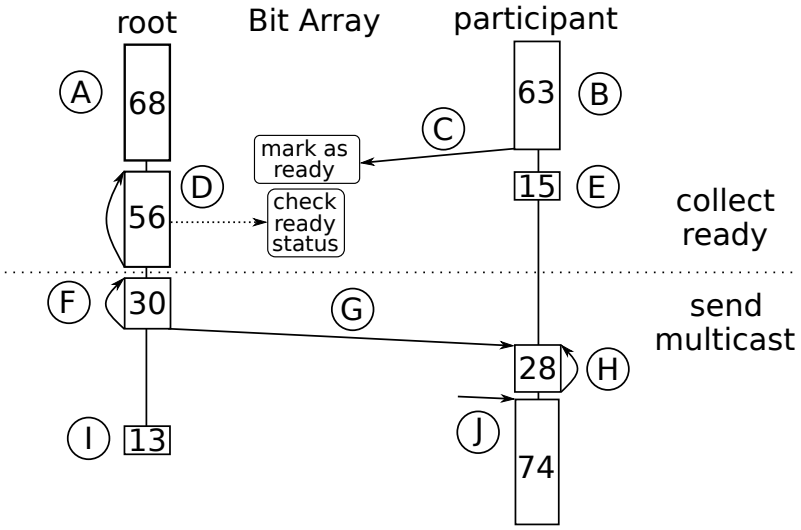


Figure 5.4.: Workflow and timing behaviour of `MPI_Bcast` with hardware support. After all participating nodes were marked as `ready` in the `Bit Array`, the root node sends out the multicast flits to the participant node(s). Generally, there are more participant nodes, but we included only one to keep the figure simple. Other participant nodes behave exactly the same way as the participant node in the figure. Boxes represent local code execution, continuous arrows between root, participant and Bit Array denote flits, arrows from the end of a box to the beginning of the same box illustrate loops. The dotted arrow at step (D) indicates the local access to the node's own `Bit Array`. Numbers inside of the boxes specify the WCET estimation of this code part. The meaning of the different steps is described in Table 5.3.

At the end of the first phase at step (D), the `ready` status of 64 nodes is checked via the `bnra` instruction. Thereby, the root node spins at the `bnra` instruction until these nodes are marked as ready. It takes $68 + 23 + 3 = 94$ cycles for the root node until the `bnra` instruction finished its execution for the first time. Thus, when all participant nodes have already marked their state in the `Bit Array`, 94 cycles at the root node represent the worst-case path. Otherwise, we have to wait for the last participant node to arrive, which takes $63 + WCTT(1,1)$ cycles. After the last participant node has arrived, the root node can continue with the rest of the

Table 5.3.: Description of structure and WCET estimates for the implementation of `MPI_Bcast` with hardware support as shown in Figure 5.4.

step	description	WCET estimate
(A)	The root node executes initialization code.	68
(B), (C)	At the end of the initialization of the participant nodes they send a `ready` flit to the root node, where the information that this participant is ready to receive data is stored in the `Bit Array`.	$63 + WCTT(1,1)$
(D)	For each 64 nodes, one iteration of a loop checking the status of the receiver nodes in the `Bit Array` has to be taken. In detail, the `bnra` instruction is called after 23 cycles. Since it requires three cycles and the following code takes 30 cycles, this code block takes 56 cycles in total.	$\lceil \frac{N}{64} \rceil \cdot (23 + 3 + 30) =$ $= 56 \cdot \lceil \frac{N}{64} \rceil$
(E)	After sending the `ready` flit and before handling incoming flits, some preparation code is executed at the participant node.	15
(F), (G)	The root node sends out multicast flits.	$\max(30 + WCTT(f,1),$ $30 \cdot f + WCTT(1,1))$
(H)	At the participant node, each flit except the last one is received and stored in a loop.	$\max(\text{sign}(f-1), 0) \cdot 28$
(I)	When all flits were sent out, the end of the function requires 13 cycles.	13
(J)	Finally, the last flit is received by the participant node and closing operations take place.	74

current iteration (30 cycles) and the remaining iterations of the loop at step (D): $(56 \cdot (\lceil \frac{N}{64} \rceil - 1))$. Altogether, the WCET estimate of the first phase is summarized as $WCET^{HwBcast}_{CollectReady}$ in Formula 5.1.

$$WCET^{HwBcast}_{CollectReady} = \max(94, 63 + WCTT(1,1)) + 30 + 56 \cdot (\left\lceil \frac{N}{64} \right\rceil - 1) \tag{5.1}$$

In the second phase, the root node sends out multicast flits, which are stored and received by the participant nodes. After the last flit was sent out, the worst-case path lies at the participating node, because step (J) is longer than step (I). However, first we have to consider f flits to be sent out at step (F). Like in Subsection 3.6.1, step (F) might be driven by the WCTT or the loop to be executed. Thus, we have to take the maximum of both: $\max(30 + WCTT(f,1), 30 \cdot f + WCTT(1,1))$. It should be noted that the second parameter for the WCTT is always set to 1 independent of the actual number of participants, because we only send out one *multicast* flit.

At the participating node at step (H) the third-last flit was already processed when the second-last flit arrives, because step (F) requires more cycles than step (H). Therefore, it is sufficient to add only the last iteration to the total WCET estimate. Because this step is not executed when only one flit is sent, we added the factor $\max(\text{sign}(f-1), 0)$. Finally, step (J) has to be added and we get $WCET^{HwBcast}_{Multicast}$ in Formula 5.2 for the second phase.

$$\begin{aligned} WCET^{HwBcast}_{Multicast} = {} & \max(30 + WCTT(f,1), 30 \cdot f + WCTT(1,1)) \\ & + \max(\text{sign}(f-1), 0) \cdot 28 + 74 \end{aligned} \tag{5.2}$$

Finally, we get the total WCET estimate $WCET^{HwBcast}_{Total}$ as result in Formula 5.3. Thereby, steps (E) and (I) can be safely ignored, because they do not lie on the worst-case path.

$$\begin{aligned} WCET^{HwBcast}_{Total} = {} & \max(31, WCTT(1,1)) + 56 \cdot \left\lceil \frac{N}{64} \right\rceil + \max(\text{sign}(f-1), 0) \cdot 28 \\ & + \max(30 + WCTT(f,1), 30 \cdot f + WCTT(1,1)) + 111 \end{aligned} \tag{5.3}$$

Timing Analysis of a Tree-Based `MPI_Bcast` Operation

Trees are current state-of-the-art to distribute data on a real-time network [SFMU18]. Our tree-based implementation works as follows: There is one root node, several intermediate nodes and a lot of leaf nodes. The message to be multicasted comes from the root node. It is received by the intermediate nodes who distribute it to the leaf nodes. There may be one or several layers of intermediate nodes. Our example will first focus on one layer of intermediate nodes and is afterwards extended to several layers during the analysis. At some points, the number of children is important. For keeping the analysis simple, we will assume that the tree is full, i.e.

the root node and all intermediate nodes will have the same number of children[6]. In the formulas, the number of children will be denoted as c. Furthermore, the number of flits will be represented as f and the number of *additional* layers as l[7].

Communication in the tree takes place in two phases: At the beginning, there is a phase for collecting `ready` information. It is propagated upwards the tree – when all nodes are ready, the root node is notified via its children. Then, the second phase starts, where data is multicasted to all nodes along the tree (propagation downwards). At the beginning, only the first data flit is sent out, then all but the last data flit and finally the last one[8]. The structure and timing behavior of the tree-based `MPI_Bcast` operation is illustrated in Figure 5.5, while the steps (A) to (K) which are displayed in the figure are explained in Table 5.4.

Now we consider the first phase: Collecting `ready` information (steps (A) to (D)). At step (A), we summarized a function for determining the node's own position in the tree together with a part of the initialization. This function was already analyzed by Bürger [Bür19] – we just copied his result here and adapted it. For the intermediate nodes, we have to consider two situations: When `ready` flits from leaf nodes have already arrived at the execution of step (C), Formula 5.4 holds. Otherwise, we have to wait for the `ready` flits at step (C) (Formula 5.5). For further estimations, we will need the maximum of both, which is assembled and simplified as $WCET^{TreeBcast}_{Ready1Intermediate}$ in Formula 5.6.

$$\begin{aligned} WCET^{TreeBcast}_{ReadyLeaf-1} &= 515 + 72 \cdot (log_2(N) + 1) + 100 \\ &\quad + 52 + 47 + 17 + (c-1) \cdot (47 + 17) \\ &\quad + 36 \end{aligned} \tag{5.4}$$

$$\begin{aligned} WCET^{TreeBcast}_{ReadyLeaf-2} &= 515 + 72 \cdot (log_2(N) + 1) + 100 \\ &\quad + 33 + WCTT(1,1) \\ &\quad + 17 + (c-1) \cdot (47 + 17) \\ &\quad + 36 \end{aligned} \tag{5.5}$$

$$\begin{aligned} WCET^{TreeBcast}_{Ready1Intermediate} &= \max(WCET^{TreeBcast}_{ReadyLeaf-1}, WCET^{TreeBcast}_{ReadyLeaf-2}) = \\ &= 637 + 72 \cdot (log_2(N) + 1) + \max(66, WCTT(1,1)) + c \cdot 64 \end{aligned} \tag{5.6}$$

[6] More cases are covered by Stegmeier [Ste19].

[7] This means when we have three layers (root, intermediate, leaf) in total, l will be 0. At 4 layers (root, intermediate, intermediate, leaf), l is 1.

[8] This structure is due to optimization to handle transfer of many flits fast.

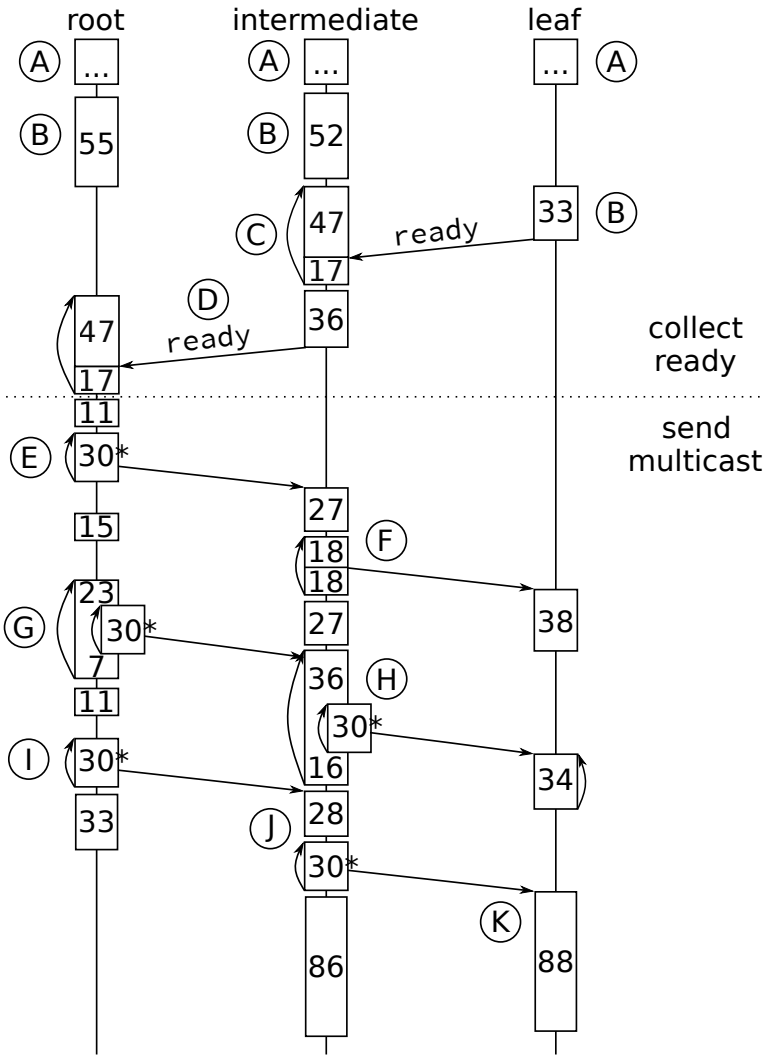


Figure 5.5.: Workflow and timing behaviour of our tree-based `MPI_Bcast` operation. After all leaf and intermediate nodes propagated that they are `ready`, data flits are sent out in three phases. Each time, the intermediate nodes get the data flits from the root node and forward them to their children. Generally, there are several intermediate and leaf nodes, but we included only one each to keep the figure simple. Other intermediate and leaf nodes behave exactly the same way as those in the figure. Boxes represent local code execution, arrows between root and intermediate nodes or intermediate and leaf nodes denote flits, arrows from the end of a box to the beginning of the same box illustrate loops. Numbers inside of the boxes specify the WCET estimation of this code part. The meaning of the different steps is described in Table 5.4.

Table 5.4.: Description of structure and WCET estimates for a tree-based implementation of `MPI_Bcast` as shown in Figure 5.5. c is the number of children of a node, while f is the number of flits to be multicasted.

step	description	WCET estimate
(A)	During the initialization, all nodes compute their position in the tree.	$515 + 72 \cdot (log_2(N) + 1) + 100$
(B), (C)	At the end of the initialization, the intermediate nodes have to wait for the `ready` flits from their children.	$\max(52 + 47 + 17 + (c-1) \cdot (47 + 17), 33 + WCTT(1,1) + 17)$
(D)	After the intermediate nodes received `ready` flits from their children, they send `ready` flits to the root node.	$\max(47 + 17 + (c-1) \cdot (47 + 17), 36 + WCTT(1,1) + 17)$
(E)	Now, the whole tree is ready to receive data. Thus, the root node starts with sending the first flit to all intermediate nodes.	$11 + 18 + (c-1) \cdot (18 + 12) + 12$
(F)	When the data flit arrived at the intermediate node, it is forwarded to the leaf nodes. $\max(\text{sign}(f-1), 0)$ ensures that this step is only considered when more than 1 flit is to be sent.	$WCTT(1,1) + \max(\text{sign}(f-1), 0) \cdot (27 + 18 + (c-1) \cdot (18 + 18) + 18 + 27)$
(G)	Should there be more than two data flits to be sent, it is iterated over the data flits (outer loop) and the children (inner loop). The inner loop works the same way as at step (E). This step is only executed when $f > 2$ via the factor $\max(0, f-2)$	$15 + \max(0, f-2) \cdot (23 + 18 + (c-1) \cdot (18 + 12) + 12 + 7)$
(H)	At the intermediate nodes, the flits from step (G) are received and forwarded to the leaf nodes. As in step (F) , the leaf node is not on the worst-case path and therefore not further respected. Like at step (G), $\max(0, f-2)$ ensures integration only for $f > 2$.	$\max(0, f-2) \cdot (36 + 18 + (c-1) \cdot (18 + 12) + 12 + 16)$
(I)	Finally, the last data flit is sent out at the root node.	$11 + 18 + (c-1) \cdot (18 + 12) + 12$
(J)	The intermediate nodes receive and forward the last data flit.	$28 + 18 + (c-1) \cdot (18 + 12) + 12$
(K)	At the leaf nodes, the last flit is received and the operation ends.	$WCTT(1,1) + 88$

Should we have several intermediate levels, then it is necessary to extend our formula $WCET^{TreeBcast}_{Ready1Intermediate}$. For each additional level, we have to consider the time it takes to collect all `ready` flits from the children at the next deeper level and to send one `ready` flit to the parent node. Because all intermediate nodes execute the same code, it is not necessary to respect steps (A), (B) and the beginning of (C) again – they are already respected at the next deeper intermediate node. An intermediate node between our current intermediate node and the root node will have to wait for its child intermediate node(s) – which have exactly the WCET estimate $WCET^{TreeBcast}_{Ready1Intermediate}$ from Formula 5.6. Thus, an additional intermediate node will be waiting for the `ready` flits from its children at step (C). Therefore, we only have to add the time for receiving and processing these `ready` flits: $17 + (c-1) \cdot (47+17)$. Moreover, we have to add the time to send the `ready` flit to the parent node at step (D): $36 + WCTT(1,1)$. When there are even more intermediate levels, the same times have to be added for each additional intermediate level. To integrate this situation, we have extended $WCET^{TreeBcast}_{Ready1Intermediate}$ from Formula 5.6 to $WCET^{TreeBcast}_{ReadyXIntermediates}$ in Formula 5.7, where l is the number of additional layers.

$$\begin{aligned} WCET^{TreeBcast}_{ReadyXIntermediates} =& 637 + 72 \cdot (log_2(N)+1) \\ &+ \max(66, WCTT(1,1)) + c \cdot 64 \\ &+ l \cdot (17 + (c-1) \cdot (47+17) + 36 + WCTT(1,1)) \end{aligned} \tag{5.7}$$

At the root node, we have to consider similar situations: When the root node reaches step (D) and the `ready` flits from intermediate nodes have already arrived, Formula 5.8 holds. Otherwise, we have to wait for the `ready` flits from the intermediate nodes at step (D) (Formula 5.9). However, because $WCET^{TreeBcast}_{ReadyRoot-1}$ from Formula 5.8 is smaller than $WCET^{TreeBcast}_{Ready1Intermediate}$ in Formula 5.6, $WCET^{TreeBcast}_{ReadyRoot-1}$ cannot lie on the worst-case path and will be ignored in the remainder[9].

$$\begin{aligned} WCET^{TreeBcast}_{ReadyRoot-1} =& 515 + 72 \cdot (log_2(N)+1) + 100 \\ &+ 55 + 47 + 17 + (c-1) \cdot (47+17) \end{aligned} \tag{5.8}$$

$$\begin{aligned} WCET^{TreeBcast}_{ReadyRoot-2} =& WCET^{TreeBcast}_{ReadyXIntermediates} \\ &+ WCTT(1,1) \\ &+ 17 + (c-1) \cdot (47+17) \end{aligned} \tag{5.9}$$

[9] We assume at least one intermediate layer in this thesis.

As result, the WCET estimate for the phase for collecting `ready` information is already given as $WCET^{TreeBcast}_{ReadyRoot-2}$ in Formula 5.9. We summarize it as $WCET^{TreeBcast}_{CollectReady}$ in Formula 5.10.

$$\begin{aligned} WCET^{TreeBcast}_{CollectReady} =& 662 + 72 \cdot log_2(N) \\ &+ \max(66, WCTT(1,1)) \\ &+ WCTT(1,1) \\ &+ c \cdot 128 \\ &+ l \cdot (c \cdot 64 - 11 + WCTT(1,1)) \end{aligned} \tag{5.10}$$

In the second phase data is multicasted. Like at the collect `ready` flits phase, we will first elaborate a formula with only one level of intermediate nodes and then extend it to more levels afterwards.

The send operation of the first data flit is seen at step (E). It contains a sending block with a WCET estimate of 30 cycles, which occurs several times in Figure 5.5 (at steps (G), (H), (I) and (J), denoted with $*$). It consists of two parts: After 18 cycles, the flit is sent out. Then, twelve cycles remain before the next iteration takes place. We will not go into further detail at the later occurrences, because always the same code is executed. Because it takes 18 cycles until the flit is sent in this sending block, the first flit from the root node is sent after $11 + 18$ cycles and the last one after $11 + 18 + (c - 1) \cdot 30$ cycles.

While the first data flit is processed by the intermediate nodes, the root node already continues with the next data flits. Because the operations at the root node are shorter than on the intermediate nodes[10], we do not need to consider the root node for the worst-case path anymore. The leaf nodes are also not on the critical path (only when receiving the last flit), because they only have to receive single flits, while the intermediate nodes have to distribute flits to their children. Thus, the WCET estimate for steps (F) and (H) can be directly taken from Table 5.4. It should be noted that the steps (F) to (H) are only executed when there are more than 1 (step (F)) or 2 (steps (G) and (H)) flits to be sent. Thus, their WCET estimates contain terms like $\max(\text{sign}(f-1), 0)$ or $\max(0, f-2)$. Thereby, *sign* is the sign function which returns -1 for negative inputs, 0 for input 0 and 1 for positive inputs. The factor $\max(0, f-2)$ works in a similar way – it prevents that the loop will have negative results when less than two flits are to be sent.

When reaching step (J), the last flit was already received, because the root node is faster than the intermediate nodes. Thus, program execution can immediately

[10] This is due to the root node only sending data, while the intermediate nodes have to receive and forward data.

proceed. The last flit from the intermediate node to the leaf node is sent out after $28 + 18 + (c-1) \cdot 30$ cycles[11]. After $WCTT(1,1) + 88$ cycles, the leaf node can finish execution. When this time is shorter than $12 + 86$ cycles, the intermediate node will be the last one to reach the end of the function. Altogether, we get $WCET^{TreeBcast}_{Multicast1Level-1}$ in Formula 5.11 as result.

$$\begin{aligned} WCET^{TreeBcast}_{Multicast1Level-1} = \\ =& 11 + 18 + (c-1) \cdot 30 \\ &+ WCTT(1,1) \\ &+ \max(\text{sign}(f-1), 0) \cdot (27 + 18 + (c-1) \cdot (18+18) + 18 + 27) \\ &+ \max(0, f-2) \cdot (36 + 18 + (c-1) \cdot (18+12) + 12 + 16) \\ &+ 28 + 18 + (c-1) \cdot 30 \\ &+ \max(12 + 86, WCTT(1,1) + 88) \end{aligned} \tag{5.11}$$

Furthermore, we have to extend $WCET^{TreeBcast}_{Multicast1Level-1}$ to respect the situation when the WCTTs are larger than the WCETs of the loop iterations. In this case, the WCET of the steps (F), (H) and (J) is WCTT-driven. Therefore, we have to take the maximum of the corresponding code parts and the WCTTs: At step (F), the first flit is sent after $WCTT(1,1) + 27 + 18$ cycles and the last one $(c-1) \cdot (18+18)$ cycles later. Afterwards, the next flit will be sent at step (H) after $18 + 27 + 36 + 18$ cycles. This flit transfer at step (H) cannot start when the flits from step (F) are still on their way. Thus, we have to take the maximum of the code execution time and flit transfer of *all* flits transmitted at step (F) as shown in $WCET^{TreeBcast}_{TimeFtoH}$ (Formula 5.12). When there are only 2 flits to send, steps (G) and (H) are not executed, but the term can still include the time at the beginning of step (H) ($36 + 18$ cycles), because it is longer than $28 + 18$ cycles at step (J). Then, step (F) directly integrates with step (J).

$$WCET^{TreeBcast}_{TimeFtoH} = \max((c-1) \cdot (18+18) + 18 + 27 + 36 + 18, WCTT(1,c)) \tag{5.12}$$

From step (H) to step (J), we have a similar situation. When code execution is faster than flit transmission, the WCET is WCTT-driven – we have to wait until the flit transfers are finished. For assembling the formula, we will first consider that 1 flit is sent at step (H) ($f = 3$) and then extend our result to more than 1 flit ($f > 3$).

The first flit can be sent after $36 + 18$ cycles at step (H). Then, it takes $(c-1) \cdot (18+12)$ cycles to send the flits, afterwards there remain $12 + 16$ cycles at step (H)

[11] Thereby, 18 cycles are the first part of the 30 cycle sending block as described above. Thus, we have respected the time for the flit to be sent to the first child. Flits to the other children are represented by the $(c-1) \cdot 30$ block. At this multiplication, we have included -1 because the flit sent to the first child was already considered at the 18 cycles.

and $28 + 18$ cycles at step (J) before the last flit is to be sent. At $WCET^{TreeBcast}_{TimeFtoH}$, we already respected the $36 + 18$ cycles at the beginning of step (H). The remainder works the same way as there: We need the maximum of the flit transfer of *all* flits transmitted at step (H) and the code execution time till the next flit transfer. Thus, we assemble formula $WCET^{TreeBcast}_{TimeHtoJ1}$ (Formula 5.13).

$$WCET^{TreeBcast}_{TimeHtoJ1} = \max(WCTT(1,c), (c-1)\cdot(18+12) + 12 + 16 + 28 + 18) \tag{5.13}$$

When we have more than 1 flit at step (H) ($f > 3$), the WCTT in $WCET^{TreeBcast}_{TimeHtoJ1}$ will have to be extended to $WCTT(f-2,c)$ and the second part has to be multiplied with the number of flits at step (H). However, we have to adapt the end of the term: $28 + 18$ represents the beginning of step (J), which only has to be included once. Furthermore, we have to include the beginning of step (H) with $36 + 18$ cycles for all further iterations. Therefore, we combine both by including $36 + 18$ cycles for all iterations. This leads to a small overestimation for the last iteration when the beginning of step (J) is executed instead of the beginning of step (H). However, we accept this overestimation of $36 - 28 = 8$ cycles to keep our formula manageable. Thus, we get $WCET^{TreeBcast}_{TimeHtoJX}$ in Formula 5.14 for step (H).

$$\begin{aligned} WCET^{TreeBcast}_{TimeHtoJX} = \max(&WCTT(f-2,c),\\ &(f-2)\cdot(c-1)\cdot[(18+12)+12+16+36+18]) \end{aligned} \tag{5.14}$$

The first part of step (J) ($28 + 18$ cycles) was already considered in $WCET^{TreeBcast}_{TimeHtoJX}$. When we have less than 2 flits, it may be represented by $36 + 18$ in $WCET^{TreeBcast}_{TimeFtoH}$. However, when only 1 flit is sent, we have to include it again, because then $WCET^{TreeBcast}_{TimeFtoH}$ and $WCET^{TreeBcast}_{TimeHtoJX}$ will not be utilized. This is done via the term $\max(\text{sign}(2-f),0)\cdot(28+18)$. For the rest of step (J), we have to take the maximum of the WCTT and the code execution like in $WCET^{TreeBcast}_{TimeFtoH}$ and $WCET^{TreeBcast}_{TimeHtoJX}$. The result is illustrated in $WCET^{TreeBcast}_{TimeJtoEnd}$ in Formula 5.15. Because no further flit transfer follows, we have to add the final 88 cycles at step (K) to the WCTT part of the max term.

$$\begin{aligned} &WCET^{TreeBcast}_{TimeJtoEnd} = \max(\text{sign}(2-f),0)\cdot(28+18)\\ &+\max(WCTT(1,c)+88, (c-1)\cdot 30 + \max(12+86, WCTT(1,c)+88)) \end{aligned} \tag{5.15}$$

We see that the first part of the second max term $WCTT(1,c) + 88$ is included in the second part of the same max term. Therefore, the second part of the max term will always be the larger term and we can ignore the first part and only respect

the second part. Altogether, we can adapt $WCET^{TreeBcast}_{Multicast1Level-1}$ from Formula 5.11 integrating $WCET^{TreeBcast}_{TimeFtoH}$ from Formula 5.12, $WCET^{TreeBcast}_{TimeHtoJX}$ from Formula 5.14 and $WCET^{TreeBcast}_{TimeJtoEnd}$ from Formula 5.15 and get $WCET^{TreeBcast}_{Multicast1Level-2}$ in Formula 5.16.

$$
\begin{aligned}
&WCET^{TreeBcast}_{Multicast1Level-2} = \\
&\quad =11 + 18 + (c-1) \cdot 30 \\
&\qquad + WCTT(1,1) \\
&\qquad + \max(\text{sign}(f-1), 0) \cdot (27 + 18 + \max((c-1) \cdot 36 + 99, WCTT(1,c))) \\
&\qquad + \max(\text{sign}(f-2), 0) \cdot \max(WCTT(f-2,c), (f-2) \cdot ((c-1) \cdot 30 + 82)) \\
&\qquad + \max(\text{sign}(2-f), 0) \cdot (28 + 18) \\
&\qquad + (c-1) \cdot 30 + \max(12 + 86, WCTT(1,c) + 88)
\end{aligned} \tag{5.16}
$$

When having several intermediate levels, the same considerations as at the collect `ready` part hold: The worst-case path is already covered with the first level of intermediate nodes as included in $WCET^{TreeBcast}_{Multicast1Level-2}$ (Formula 5.16). We now have to extend it with the time it takes for more intermediate layers to receive and forward flits. This is covered with the WCET of steps (F), (H) and (J) (now without the final 86/88 cycle blocks, because we are only interested in the point in time when the last flit is sent) for each intermediate layer again. When integrating these numbers in Formula 5.16 and summarizing some parts, we get the new Formula 5.17.

$$
\begin{aligned}
&WCET^{TreeBcast}_{MulticastXLevels} = \\
&\quad =11 + 18 + (c-1) \cdot 30 \\
&\qquad + WCTT(1,1) \\
&\qquad + 28 + 18 + (l+1) \cdot \max(\text{sign}(f-1), 0) \cdot \max(c \cdot 36 + 63, WCTT(1,c)) \\
&\qquad + (l+1) \cdot \max(\text{sign}(f-2), 0) \cdot \max(WCTT(f-2,c), (f-2) \cdot (c \cdot 30 + 52)) \\
&\qquad + (l+1) \cdot (c \cdot 30 - 30 + \max(10, WCTT(1,c))) + 88
\end{aligned} \tag{5.17}
$$

Finally, $WCET^{TreeBcast}_{CollectReady}$ from Formula 5.10 and $WCET^{TreeBcast}_{MulticastXLevels}$ from Formula 5.17 have to be added to get the total WCET estimate $WCET^{TreeBcast}_{Total}$ in Formula 5.18.

$$
\begin{aligned}
WCET_{Total}^{TreeBcast} &= WCET_{CollectReady}^{TreeBcast} + WCET_{MulticastXLevels}^{TreeBcast} = \\
&= 765 + 72 \cdot log_2(N) + c \cdot 188 + \max(66, WCTT(1,1)) + l \cdot (c \cdot 94 - 41) \\
&\quad + (l+2) \cdot WCTT(1,1) \\
&\quad + (l+1) \cdot \max(\text{sign}(f-1), 0) \cdot \max(c \cdot 36 + 63, WCTT(1,c)) \\
&\quad + (l+1) \cdot \max(\text{sign}(f-2), 0) \cdot \max(WCTT(f-2,c), (f-2) \cdot (c \cdot 30 + 52)) \\
&\quad + (l+1) \cdot \max(10, WCTT(1,c))
\end{aligned} \tag{5.18}
$$

5.8.2. Comparison of WCET Estimates

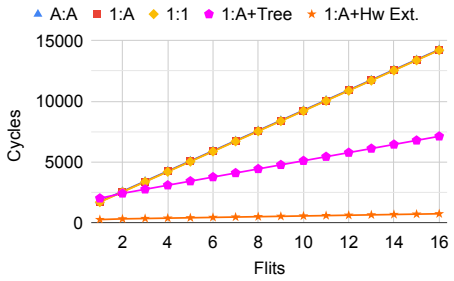
In this section, we compare the WCET estimates of the different implementations of `MPI_Bcast` from the last Subsection 5.8.1. For this, we combine them with the schedules 1:1, A:A and 1:A. Furthermore, we consider a simple implementation like it was described in Section 3.6. At this implementation, the root node iterates over all flits to be sent (outer loop) and participating nodes (inner loop). An implementation working this way was already analyzed by Bürger [Bür19]. His result is $WCET_{Total}^{SimpleBcast}$ in Formula 5.19. In this formula, f is the number of flits to be sent to χ nodes.

$$
\begin{aligned}
WCET_{Total}^{SimpleBcast} &= \max(f \cdot (3 + \chi \cdot 52) + WCTT(1,1), WCTT(f, \chi - 1) + 120) \\
&\quad + 76 + 41 \cdot \chi + WCTT(1,1)
\end{aligned} \tag{5.19}
$$

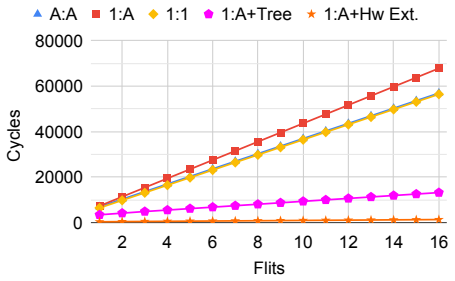
Thereby, we consider an RC/MC processor with 4x4 = 16 nodes, 8x8 = 64 nodes and 16x16 = 256 nodes. We evaluate how long it takes to broadcast/multicast varying numbers of flits to different parts of the chip. For communication between the nodes, we compare:

1. $WCET_{Total}^{TreeBcast}$ (Formula 5.18) configured as binary tree ($c = 2$), which is the most efficient way to broadcast/multicast flits through the NoC via the original 1:A schedule without hardware extensions [SFMU18].
2. $WCET_{Total}^{HwBcast}$ (Formula 5.3) running on our extended 1:A schedule with hardware support for multicasts and broadcasts.
3. $WCET_{Total}^{SimpleBcast}$ (Formula 5.19) utilizing the generic schedules 1:1, A:A and the original 1:A schedule without hardware extensions.

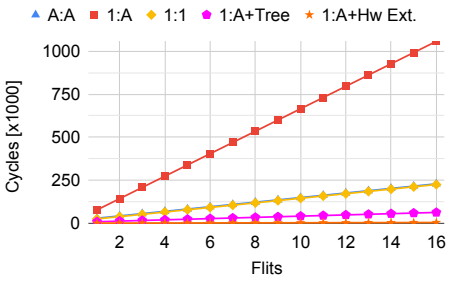
A tree-based algorithm makes only sense in combination with the 1:A schedule, because at the A:A schedule flits to different communication partners can all be sent in the same period and at the 1:1 schedule things are more complicated: Because of the condition that each node can receive at most one flit each period, `ready` flits



(a) Broadcast in a 4x4 node NoC



(b) Broadcast in a 8x8 node NoC



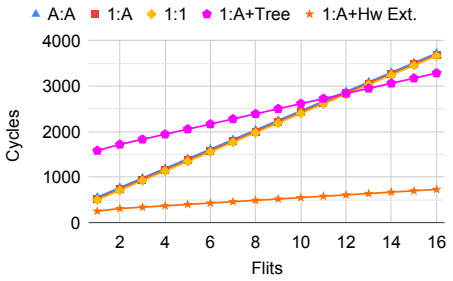
(c) Broadcast in a 16x16 node NoC

Figure 5.6.: WCET estimates for broadcasts in NoCs with 4x4, 8x8 and 16x16 nodes.

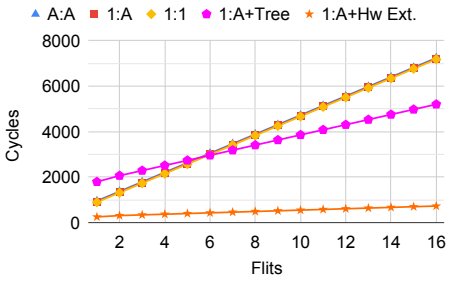
have to be sent via a dedicated `ready` NoC each time the communication partner changes. Thus, it is more efficient to utilize the short periods of the 1:1 schedule to send the data flits directly to all participating nodes.

Figure 5.6 displays results for broadcasts in 4x4 (a), 8x8 (b) and 16x16 (c) node NoCs. In all cases, our hardware supported broadcast operation (called *1:A+Hw Ext.* in the figures) has a WCET estimate far below the other WCET estimates. At our benchmarks, we did not utilize the `bnar` instruction, which would save a few cycles when checking the `ready` states. However, we decided to carry out our evaluation without `bnar`, because its hardware costs were considered to be quite expensive in Subsection 5.7.2. Since our results still convince without `bnar`, it may be cut out to decrease hardware costs.

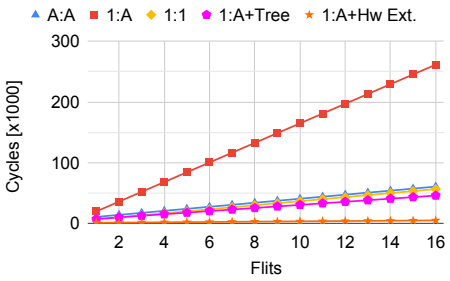
Our hardware supported broadcast operation is followed by the binary tree executed on the original 1:A schedule (*1:A+Tree*), which scales better than the simple broadcast operation (independent from the utilized schedule). At the simple broadcast operation, it can be noticed that the 1:1 and A:A schedules always have similar WCET estimates. For the broadcast operation in a 4x4 NoC (Figure 5.6a), the same holds for the simple broadcast operation combined with the original 1:A schedule. This is because in these cases the WCET of the simple broadcast operation is driven by the program code, i.e. we cannot use the full capacity of the NoC due to waiting



(a) Multicast to 4 of 16 nodes



(b) Multicast to 8 of 64 nodes



(c) Multicast to 64 of 256 nodes

Figure 5.7.: WCET estimates for multicasts in NoCs with 4x4 and 16x16 nodes.

for the execution of the next `snd` instruction. For the 1:1 and A:A schedules this stays the case in all scenarios. However, at the 1:A schedule, the WCTTs increase fast when the NoC size grows. Then, the WCET of the simple broadcast operation combined with the original 1:A schedule is driven by the WCTT as can be seen at the 8x8 and 16x16 node NoCs (Figures 5.6b and 5.6c).

At the WCET estimates for multicasts displayed in Figure 5.7, our hardware supported multicast remains also below all other multicast implementations. However, the binary tree does not always perform second-best for multicasts to 4 or 8 nodes in a 4x4 node NoC as we see in Figures 5.7a and 5.7b (par at 12 flits for 16 nodes and 6 flits for 64 nodes). This is because the tree needs some time to "warm up". Thus, trees are paying off especially when sending messages to a high number of nodes or sending a lot of flits. Like at the WCET estimates for broadcasting flits to all nodes in Figure 5.6, the WCET estimate of the simple broadcast operation is also driven by the code to be executed at the multicast operation. Only when employing the 1:A schedule and increasing the number of nodes, it is driven by the WCTT, as can be seen in Figure 5.7c.

5.8.3. Theoretical Comparison

The WCET analysis in Subsection 5.8.1 is performed with compiler optimizations disabled (compiler optimization level `-O0`) and may be improved. Moreover, at some parts of the comparison in the last Subsection 5.8.2, we already noticed that WCETs may be code-driven, meaning that the NoC is not fully utilized. Therefore, we carry out a theoretical comparison in this section to see the impact of our hardware extension when code would be optimized in an optimal way. For this, we consider only transportation times and assume code execution times to be zero.

Table 5.5.: How large is the WCTT for flit transfer for a broadcast/multicast operation with f flits to be transmitted to χ receiver nodes in a NoC with a dimension of n?

schedule	`ready` transm.	data transmission
One-to-One (1:1)	1 period 1:A $= n^2 + 2n$	$\chi \cdot f$ periods 1:1 $= n \cdot \chi \cdot f + 2n$
All-to-All (A:A)	1 period A:A $= \left(\frac{n^2(n-1)}{2} + 2\right) + \frac{n^2}{2} + 2n$	f periods A:A $= \left(\frac{n^2(n-1)}{2} + 2\right) \cdot f + \frac{n^2}{2} + 2n$
One-to-All (1:A) with binary tree	1 period 1:A $= n^2 + 2n$	$f \cdot (\lceil log_2(\chi + 1) \rceil - 1) \cdot 2$ periods 1:A $= n^2 \cdot f \cdot (\lceil log_2(\chi + 1) \rceil - 1) \cdot 2 + 2n$
One-to-All (1:A) with broadcast/ multicast extension	1 period 1:A $= n^2 + 2n$	f periods 1:A $= n^2 f + 2n$

Table 5.5 summarizes how long `ready` and data transmission take in the different schedules. In the first line of each schedule, it is stated how many periods are needed to transmit f flits to χ receiver nodes in a NoC with a dimension of n. Then, in the second line, the corresponding formula is given how many cycles are needed for transmission. Most of these formulas were already introduced in Subsection 3.2.2.

When employing the 1:1 schedule, a dedicated `ready` NoC is necessary, because in the 1:1 schedule there are no free time slots for `ready` coordination as described in Chapter 4. For small NoC sizes, a ring is sufficient, but since a ring does not scale well, a 2D NoC is better. Walter investigated how the 1:1 schedule can be utilized in an optimal way in his master's thesis [Wal19] and comes to the conclusion that the `ready` NoC should use the 1:A schedule. Therefore, transmission times in the 1:1 schedule require the 1:A schedule for `ready` synchronization, while the 1:1 schedule is employed for data transmission. For the latter, data flits are sent out in a `for` loop, iterating over flits and receiver nodes.

At the A:A schedule, we assume that each flit can be sent out to all nodes within one period. Therefore, one period of the A:A schedule is sufficient for `ready` synchronization and f periods for the transmission of all data flits.

We assume one period when transmitting `ready` flits via the 1:A schedule with binary tree: All nodes directly tell the root node that they are ready to receive data. Data transmission in the binary tree takes $\lceil log_2(\chi + 1) \rceil - 1$ periods for each flit[12]. A binary tree is the most efficient way to distribute data in a real-time NoC [SFMU18].

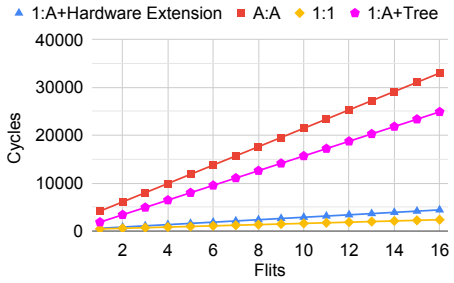
Finally, we consider the 1:A schedule with our hardware broadcast/multicast extension. `Ready` synchronization can be carried out in one single period. The data transmission time is now very short: Instead of several periods to transmit each flit through the NoC, only f periods of the 1:A schedule are required to send all data flits to all receivers.

Figure 5.8 visualizes the results of our theoretical comparison: It shows multicasts to varying numbers of receiver nodes in a 256 node NoC as well as multicasts in NoCs with 16 and 64 nodes.

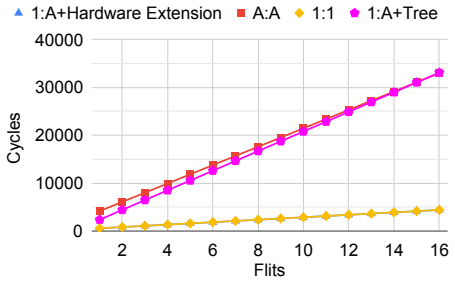
The first Figure 5.8a illustrates a multicast to 8 of 256 nodes. It can be seen that the A:A schedule needs the most cycles to distribute data, followed by the 1:A schedule with binary tree. 1:A with hardware-supported broadcast/multicast is a lot faster, but outperformed by the 1:1 schedule. When a multicast is sent to less than n nodes, then 1:1 is always better than our hardware extended 1:A schedule. This can be derived from the admission times from Subsection 3.2.2: For a multicast utilizing the 1:1 schedule, the admission time is $n \cdot \chi \cdot f$, while it is $n^2 \cdot 1 \cdot f$ for the 1:A schedule with broadcast/multicast hardware extension ($\chi = 1$ because only 1 multicast flit is to be sent out to reach all receivers). When $\chi < n$, the admission time for the 1:1 schedule is smaller than that of 1:A with hardware extensions. At $\chi = n$, both schedules coincide and for $\chi > n$, 1:A with hardware supported multicast is faster, because it remains constant.

The matching of the 1:1 schedule and the 1:A schedule with hardware supported multicast can also be recognized in Figure 5.8b, where a multicast is sent to exactly n (here 16) nodes. Here, the graph for the 1:A schedule with hardware-supported broadcast/multicast overlaps with the 1:1 schedule, which is why only the 1:1 schedule can be seen. Both schedules need exactly the same amount of cycles for data transmission. The A:A schedule is stable, because it is independent of the number of receiver nodes χ. However, the 1:A schedule with binary tree now partially overlaps with the A:A schedule, because the number of periods needed increases quite fast with the number of multicast receivers: For 8 receivers, the tree has a depth of 3 resulting in up to 6 periods, for 16 receivers the depth is 4, resulting in up to 8 periods. All data flits have to be propagated along the tree, needing 2

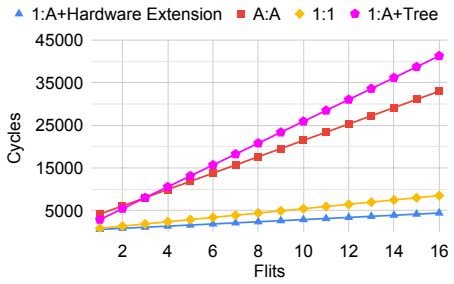
[12]The number of periods was explained in Section 5.1.



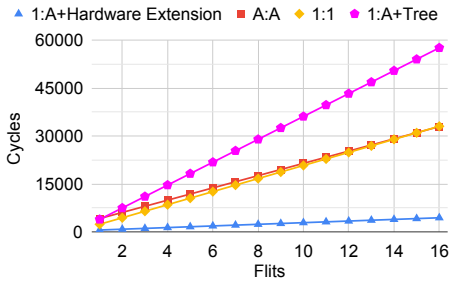
(a) Multicast to 8 of 256 nodes



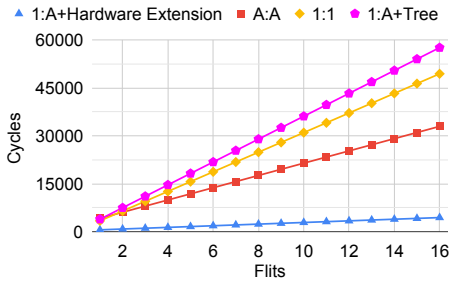
(b) Multicast to 16 of 256 nodes



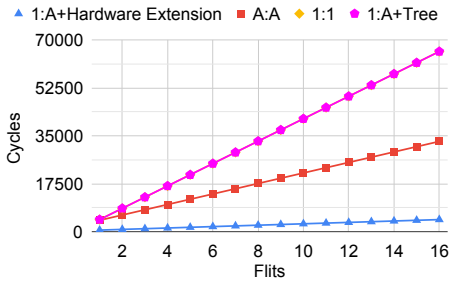
(c) Multicast to 32 of 256 nodes



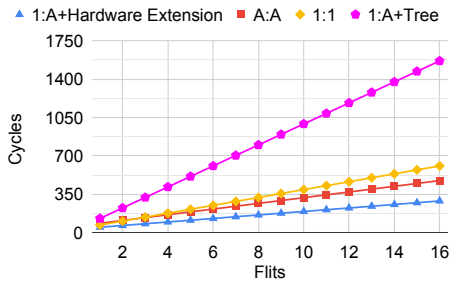
(d) Multicast to 128 of 256 nodes



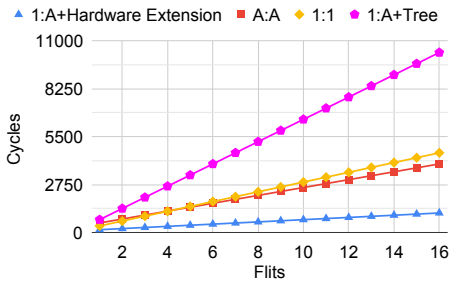
(e) Multicast to 192 of 256 nodes



(f) Broadcast to 256 nodes



(g) Multicast to 9 of 16 nodes



(h) Multicast to 35 of 64 nodes

Figure 5.8.: Multicast/broadcast in a NoC with 256 nodes and multicast in 16/64 node NoCs.

periods to arrive at the next level (the first period for the left child in the binary tree, the second period for the right child).

At the next Figure 5.8c, the 1:A schedule with binary tree passed the A:A schedule for $f > 3$. Both cycle counts of the 1:1 schedule and the 1:A with binary tree schedule steadily increase. In Figure 5.8d, the 1:1 schedule also passes the A:A schedule: It is faster than the A:A schedule when up to 15 flits are multicasted and slower for more flits. Afterwards, the 1:1 and 1:A schedule with binary tree go on to increase, while the other two schedules A:A and 1:A with multicast hardware extension remain at their positions, which is illustrated in Figure 5.8e.

Finally, we get the situation in Figure 5.8f, which presents a broadcast to 256 nodes: Although the 1:A schedule with binary tree never performed well in this theoretical comparison, it is on par with the 1:1 schedule, which was the best schedule for small numbers of receiver nodes. Both are outperformed by the A:A schedule as well as the 1:A schedule with hardware-supported broadcast/multicast, which remained at a very low level throughout all numbers of receivers.

After having a very close look on the schedule behavior with 256 nodes, we only consider two figures for configurations with 16 and 64 node NoCs. Basically these configurations have the same behavior like the 256 node NoC with a break-even point of the extended 1:A schedule at n receiver nodes (4 in a 16 node NoC and 8 for 64 nodes). In Figure 5.8g, a multicast is sent to 9 nodes in a 4x4=16 node NoC. Here the 1:1 schedule almost overtook the A:A schedule: 1:1 is only faster for one and two flits. It started to overtake the A:A schedule at $\chi = 7$: There, 1:1 was faster than A:A for up to 13 receivers. Altogether, the 1:A schedule with binary tree performs worst, while the 1:A schedule with hardware-supported multicast performs best. Figure 5.8h illustrates a similar situation for 35 receiver nodes in a 8x8=64 node NoC. Here, the 1:1 schedule is faster than the A:A schedule for small flit numbers ($f \leq 4$), but is slower for large flit numbers ($f \geq 5$).

From the view of the theoretic comparison, our 1:A schedule with hardware-supported broadcast/multicast always performs best except when a multicast is sent to very few nodes (less than n). Only in these cases, the 1:1 schedule is the better choice.

5.9. Conclusion

In this chapter, we presented the hardware broadcast/multicast extension to improve the 1:A schedule. It exploits reserved TDM slots of the 1:A schedule to broadcast flits to *all* nodes within one period. Thereby, flits are copied while travelling through the NoC. We integrated the ability in the nodes to check if a broadcast flit comes from a desired sender and count them. This enables us to realize multicast. For controlling our hardware broadcast/multicast extension, we added four new

assembly instructions: A broadcast/multicast specific `ready` and send instruction each and two status branches. Our hardware broadcast/multicast extension outperforms all other approaches in the worst-case and scales very well for increasing node numbers. It comes at the cost of around 10% more ALMs and 5% more registers. The hardware costs may be reduced by cutting out the second status branch, which has negligible impact on (worst-case) performance.

6

Hardware Barrier Extension to Improve Schedule One-to-All

Abstract. In parallel programs following the BSP model, barriers are a crucial coordination operation. They are required at the end of each superstep to separate the communication phase of a superstep and the begin of the next superstep, which starts with local computation. However, software implementations of barriers impose a lot of communication and coordination overhead. Therefore, our idea is to implement barriers in hardware. We employ the hardware-supported broadcast/multicast operation presented in the previous chapter to realize hardware-supported barriers.

6.1. Introduction

Barriers are an essential coordination operation in parallel programs, especially when considering BSP programs. Therefore, it is highly desirable to have an efficient implementation for them. In this regard, the Bruck Algorithm [BHK+97] is current state-of-the-art when realizing them in software for real-time systems [SFMU18]. Thereby, a node exchanges its knowledge about other nodes in several iterations with other nodes. By clever selection of communication partners, a node doubles its knowledge about other nodes every iteration. Thus, it only takes $\lceil log_2 \chi \rceil$ periods of a schedule for a node to know that χ participating nodes have arrived at a barrier.

Furthermore, a separation into distinct phases for collecting `ready` flits and sending out data (barrier release information) like in the previous Chapter 5 is not necessary anymore. However, this still means that realizing a barrier requires several periods of the 1:A schedule until all participating nodes can continue program execution.

An alternative would be to implement barriers with the hardware broadcast/multicast extension presented in the previous Chapter 5. This would work with all participating nodes sending a `ready` flit to the coordinator node, which sends out a `mcst` flit as soon as all participating nodes have arrived. However, this leads to problems when several barriers (with different participants) are coordinated by the same coordinator node, because multicasts are internally implemented as broadcasts.[1] Thus, there currently does not exist a safe way to distinguish several multicasts (with different receiver nodes) from the same sender node as already described in Section 5.4, where we already recommended to separate those multicasts with barriers.

To overcome this problem, we now extend the hardware-supported broadcast/multicast operation from the previous chapter to a hardware-supported barrier operation. Our basic idea is to have a dedicated *barrier control unit (BCU)*, which can be notified by all participating nodes when they have arrived at a barrier. When all participating nodes have arrived, a broadcast/multicast flit as implemented in Chapter 5 is sent out to inform the nodes that they can now continue their program execution. Thereby, our focus is on high efficiency providing low timing bounds and low hardware effort.

The remainder is structured as follows: In the next Section 6.2, related work is presented. Afterwards, we describe the concept and implementation considerations of hardware-supported barriers: In Section 6.3 for global barriers and in Section 6.4 for non-global barriers. The presented techniques are implemented in Section 6.5 and the programming model is illustrated in Section 6.6. After examining hardware costs in Section 6.7, the evaluation takes place in Section 6.8. Finally, the results are concluded in Section 6.9.

6.2. Related Work

Having hardware support for barriers is not a new concept. It is well researched in the field of high-performance systems and supercomputers. However, no literature was found on hardware barriers taking real-time aspects into account.

The IBM BlueGene/L [ABC+05] supercomputer employs several NoCs, each

[1] A software based solution for this would work like software `ready` synchronization (see Section 4.4) utilizing the payload of multicast flits to identify barriers. However, due to the experiences made with software `ready` synchronization, we will solve this in hardware in Subsection 6.4.2.

with a dedicated purpose. It implements a *global interrupt network*, which realizes hardware barriers for 32, 128, 512 and multiples of 512 nodes. Barriers for other node numbers may be implemented as special case of `MPI_Allreduce` utilizing the *collective network* at which each node injects some message into the network and waits for the response from the root node [AHA+05]. Altogether, the BlueGene/L implements a lot of communication capabilities, but at the cost of dedicated NoCs for each of them, which means improved hardware cost, energy consumption and coordination effort.

Another approach with a dedicated NoC comes from Abellán et al. [AFA10], who developed a hardware barrier based on the work of Krishna et al. [KKC+08]. They employ a many-core with a 2D-mesh NoC, where G-Lines are used to build up an additional network to carry out barrier synchronizations. Thereby, four cycles are required to carry out synchronization once all nodes have arrived at the barrier. However, their approach only works for barriers for the complete network, no barriers for subsets are possible.

Sartori and Kumar suggest three barrier implementations that are hybrids of software and dedicated hardware barriers [SK10]: Their first idea is a barrier network organized as virtual binary tree. This tree structure is stored in the NoC routers. For relieving barrier coordination effort from the PE, NoC routers are further extended with a state machine. It stores barrier notifications from the node where it is installed as well as from its children. When a node and its children have arrived at the barrier, this information is propagated upwards the tree to the parent node. This propagation continues until the root node is reached. Then, the root node propagates a *barrier release* notification downwards the tree. This way, the PE only has to deal with its own barrier arrival and release, the rest is done by the extended NoC routers.

Second, Sartori and Kumar improve the performance of their barrier tree by adding *barrier bolstering* [SK10]. This means to add shortcuts in the form of dedicated links between nodes to reduce the critical path of the barrier tree. For example, when the critical path in the barrier tree has a length of three hops, a dedicated link from the deepest node to the root node may be added to reduce the critical path to a length of two hops. This is just a performance improvement for cases where tasks can be mapped to nodes where these shortcuts can be utilized. In the worst-case, task mapping is not possible in this way and the performance is the same as for a platform without barrier bolstering. Moreover, the shortcuts may lead to long links, especially when having many nodes. This may result in increased latencies.

The motivation for the third idea of Sartori and Kumar [SK10] is that virtual links in virtual trees have a short distance, but on the physical layer a barrier notification may need to take multiple hops to take one virtual link. Thus, they suggest the use of *express virtual channels* as introduced by Kumar et al. [KPKJ07]. Thereby, all intermediate routers are spanned and the barrier notification only has to be routed

once although it takes several hops at the same time. However, this comes at the cost of increased router complexity and degradation of other network traffic as express virtual channels do not allow any other communication on intermediate routers.

Oh et al. [OPZ11] introduce *TLSync*, a hardware barrier utilizing the high frequency band of on-chip radio frequency transmission lines. All participating nodes send a "tone" at a barrier specific frequency as long as they have not arrived at the barrier. As long as this "tone" can be detected, all finished nodes have to wait until it disappears. When no "tone" is detected anymore, all nodes have arrived at the barrier and can continue their program execution. This approach supports many (tens) of barriers simultanously and is restricted to the high frequency band to leave the other frequencies for normal data transmission. However, it works only on architectures with on-chip radio frequency transmission lines and thus has to deal with their latencies, disturbance variables etc.

In his master's thesis, Walter implements hardware-barriers for the RC/MC processor utilizing the 1:1 schedule [Wal19]. Because each node is allowed to receive at most one flit per period, `ready` synchronization is done in a dedicated NoC, the *ready ring*. It allows simple broadcasting of status messages (e.g. `ready` status). Thereby, each node sends a status message each period. This status message is utilized for barriers. All participating nodes broadcast that they have arrived at a barrier and count how many barrier broadcasts they received. When they have received as many broadcasts as there are participants, they know that all other participating nodes have arrived at the barrier and they can now continue program execution.

Another master's thesis carried out by Auer [Aue18] investigated hybrid hard-/ software barriers for the 1:A schedule in the RC/MC processor. His idea is send `ready` flits to a coordinating node and interpret them as *barrier arrival* flits. This approach works well, we will use it for our evaluation. However, Auer focused only on global barriers. He also tried to implement an automatic hardware-only barrier, but it has quite a long delay due to high communication overhead. Thus, it performs worse than his hybrid approach.

A fast and hardware-efficient way to synchronize a multiprocessor system with a barrier register is described by Beckmann and Polychronopoulos [BP90]. We combine their approach with our hardware-supported broadcast operation to realize hardware-supported barriers on our platform, see details in Section 6.3, where we also explain their idea. Furthermore, we extend the approach to provide non-global barriers in Section 6.4.

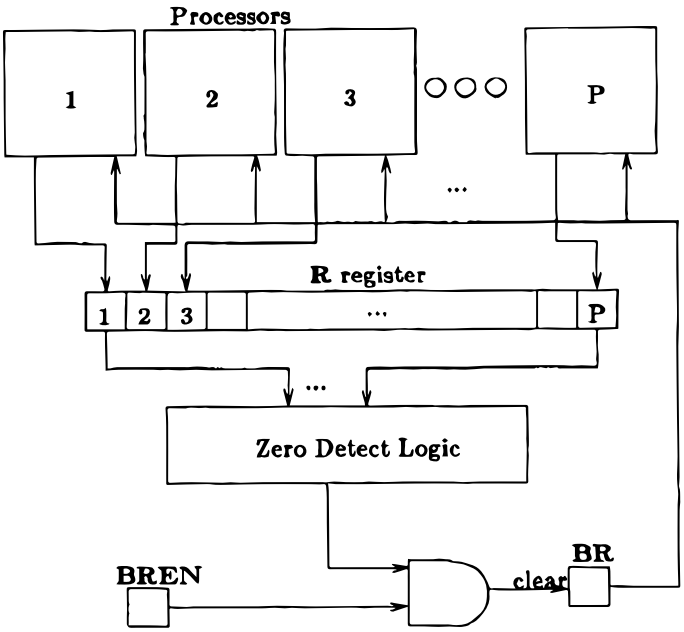


Figure 6.1.: Original single barrier register hardware as suggested by Beckmann and Polychronopoulos [BP90]: When arriving at the barrier, each processor stores 0 at its bit in the `R register` (1 otherwise). As soon as all processors have arrived, this is detected by the `Zero Detect Logic` and a broadcast is sent out via the `BR` register to inform all processors that they can continue their program execution. Figure taken from [BP90].

6.3. Concept for Global Hardware Barriers

For our hardware-supported barrier operation, we implement a variation of the *Fast Barrier Synchronization Hardware* from Beckmann and Polychronopoulos [BP90]. Their approach is illustrated in Figure 6.1. They realize a hardware barrier as register `R`, where each processor contributes one bit: It is set to 1 while the corresponding processor is working and reset to 0 when it has finished its work (and arrived at the barrier). Then, each processor is waiting for the other processors to arrive at the barrier. The register is connected to a zero detect logic, which checks if all bits are zero. When they are, all processors have arrived at the barrier. Then, a *barrier release* broadcast to all processors can be sent out via the single-bit register `BR`, so that they can continue program execution.

Instead of the `BR` register and the additional broadcast signal, we employ our hardware-supported broadcast operation as described in the previous Chapter 5. There is no call to the `mcst` instruction, the *barrier release* broadcast operation is directly initiated from the BCU to avoid wasting any time. At the same time as the broadcast is carried out, the barrier is reset. In the remainder, we will refer to the

`R` register as `barrWaiting` register. Furthermore, in our implementation we swap the meaning of bits: 0 means that a node is working and 1 that it has arrived at the barrier. Thus, our `barrWaiting` register is reset by resetting all bits to 0. After the BCU released and resetted the barrier, it can immediately be reused for the next coordination operation. A BCU in installed on each node. Therefore, it is possible to have several barriers for different groups of nodes at the same time.

For exchanging information about *barrier arrival* and *barrier release*, flits have to be sent between the coordinating node and participating nodes. Thereby, the coordinating node is the one collecting all *barrier arrival* flits of a specific barrier and initiating the final *barrier release* broadcast. The configuration of a barrier directly takes place by the coordinating node (details follow in the remainder of this and the next section). Thus, it is necessary that the coordinating node participates at each barrier which is carried out at its BCU.

At the NoC schedule, it has to be considered that the node coordinating the barrier has to wait one period until it can inject the next flit, because the send slot in the current period is occupied by the *barrier release* broadcast flit from the BCU. All nodes participating at the barrier have to wait for the *barrier release* broadcast flit to arrive before they can continue their program execution. They receive it at the end of the same period, which means that their next send operation can take place at the following period in the best case. The node coordinating the barrier can immediately continue its computations and send flits, because it does not have to wait for the *barrier release* broadcast flit. Instead, it is notified via a local signal which allows to continue program execution one cycle after *barrier release*. All other nodes participating at a barrier can continue their program execution as soon as the *barrier release* broadcast flit arrives.

Participation at barriers is usually implemented as one single call from the program, e.g. `pthread_barrier_wait` at POSIX Threads [IG18] or `MPI_Barrier` at MPI [Mes15]. Although these operations take place at a rather high level, we originally intended to implement hardware supported barriers with one single `barrier` assembly instruction. However, this would be a rather complex instruction as it should first send a *barrier arrival* flit and afterwards wait for the *barrier release* broadcast flit. Instead, a separation of arriving at a barrier and waiting for its release is closer to the *Reduced Instruction Set Computing (RISC)* concept:

Arriving at a barrier is implemented by sending a flit to the coordinating node. This could be realized with a specialized send instruction called `brrav (barrier arrival)`, see Table 6.1. Because `brrav` is a modified `snd` instruction raising an exception when the send buffer is full, `bsf` or `bsnf` has to be executed before executing `brrav`. At the coordinating node, `brrav` has a different behavior: The *barrier release* broadcast flit is put in the send buffer. Before it can be sent from there, a signal from the BCU has to indicate that all nodes have arrived at the barrier.

Waiting for the release of a barrier means to wait for the receipt of the corresponding *barrier release* broadcast flit at the participating nodes. Following the PIMP status branch concept, this could be implemented by an additional branch instruction `bbnr (branch if barrier not released)`, which checks if this flit was received. Explicitly receiving the flit afterwards via `rcvp` (`receive payload`, see Section 3.4) is not necessary, because it is processed by the dedicated BCU. Having two instructions for handling barriers gives the software developer the opportunity to execute code between arriving at a barrier and waiting for its release. However, we recommend to use `brrav` and `bbnr` always consecutively to keep barriers compact and avoid additional waiting times. As `brrav` already puts the *barrier release* multicast flit into the send buffer, the waiting time might also be used to put the next flits there. Due to the organization of the send buffer as FIFO buffer, they will all be sent out after the barrier release.

Table 6.1.: Overview on our RISC-V instruction set extension for barriers.

mnemonic	source register 1	source register 2	immediate value	function
`brrav`	*node*	*uid*		arrive at barrier which is coordinated by *node* and has unique id *uid*
`bbnr`			*address*	branch to *address* when the barrier where the node participates via `brrav` has not yet been released
`cbrr`	*part*	*nodes*		configure the *part* of the `barrExpected` register to set/reset bits for given *nodes*
`mbrr`	*mode*			when the content of *mode* is set (is non-zero), enter configuration mode (no barrier is released). When it is reset to 0, leave configuration mode.

6.4. Hardware Barriers for Subsets of Nodes

We now extend the concept from the previous Section 6.3 to barriers for subsets of all nodes in Subsection 6.4.1. Because multicasts are internally implemented as broadcasts, we introduce the *unique id* to distinguish *barrier release* multicast flits of consecutive barriers in Subsection 6.4.2. Finally, we illustrate the behavior and

communication flow for a complete hardware barrier operation in Subsection 6.4.3.

6.4.1. Concept for non-global Barriers

Although the BSP model relies on global barriers, barriers for subsets of all nodes are needed sometimes in parallel programs (cf. Figure 3.3 on page 17). Moreover, our platform may execute several applications at the same time, where each application is running on a dedicated group of nodes. Then we need barriers which only coordinate one of these groups.

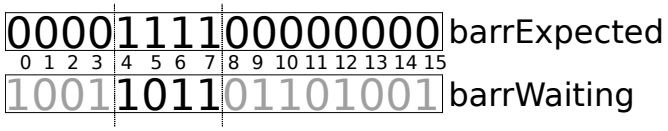


Figure 6.2.: Hardware barrier registers `barrExpected` and `barrWaiting`: The bits set to 1 in register `barrExpected` indicate which bits are currently respected in register `barrWaiting` (i.e. which nodes participate at the current barrier; each bit represents one node). When all of these bits are also set to 1 in `barrWaiting`, all participating nodes have arrived at the barrier and the *barrier release* multicast flit can be sent out. In this example, we currently only wait for the node on position 5 to arrive, then the barrier can be released.

To enable barriers for subsets of nodes, we need to define which nodes participate at a barrier. Therefore, we introduce a second register `barrExpected` working similar to the `barrWaiting` register. Its concept is illustrated in Figure 6.2. In both registers `barrWaiting` and `barrExpected` each bit represents one node. For the current barrier only the nodes selected in register `barrExpected` are respected in register `barrWaiting`.[2] When all bits set to 1 in `barrExpected` are also set to 1 in `barrWaiting`, all participating nodes have arrived at the current barrier and can be notified via a *barrier release* multicast flit that they can continue their program execution. In contrast to the global barrier, only the bits of the participating nodes are reset to 0 in the `barrWaiting` register while the *barrier release* multicast flit is sent out.

For changing the contents in the `barrExpected` register, we add an additional instruction `cbrr (configure barrier)` as described in Table 6.1. It sets/resets the bits representing the nodes which (do not) participate at the barrier. Since the number of bits in the registers `barrWaiting` and `barrExpected` matches with

[2] In `barrWaiting` 1 means that a node has arrived at a barrier and 0 that we still wait for it. In `barrExpected` 1 means that we respect this node at the current barrier and 0 that we currently ignore it.

the number of nodes on the chip, there may be more than 64 bits to be set/reset. Then, the contents of the `barrExpected` register cannot be changed at once, which is the same situation as with `bnra (branch if not ready array)` in Section 5.5. Thus, `cbrr` works the same way as `bnra`: When there are more than 64 nodes, the corresponding block has to be specified as parameter (cf. Figure 5.2 on page 58). Should there be up to 64 nodes, the block parameter always has to be zero. In the case when there is not a multiple of 64 nodes on the chip, the bits not representing nodes are generally ignored and treated like non-participants. An example would be a 4x4 NoC with 16 nodes. Then, `cbrr` can only set up to 16 participants in part 0.

While changing the contents of the `barrExpected` register on many-core processors with more than 64 nodes, it could happen that `barrWaiting` and `barrExpected` match although not intended and a *barrier release* multicast flit is sent out. Therefore, a configuration mode has to be implemented, controlled by the new instruction `mbrr (mode of barrier)`. While the BCU is in configuration mode, no comparisons between the registers `barrWaiting` and `barrExpected` take place and no barrier is released. When the content in the source register of `mbrr` is non-zero, configuration mode is entered and when it is zero, configuration mode is left.

To provide enough barriers, one BCU is installed at each node. Thus, there are always N barriers which should be enough to coordinate one large or several smaller distributed application(s) on a many-core chip. By installing two or more BCUs at each node, even more barriers could be provided, but since each node may only participate at one barrier at a time, one barrier at each node should be sufficient.

6.4.2. Distinction of Flits of Two Consecutive Barriers

The source node's id is not sufficient to distinguish several barriers when several barriers with different participants are coordinated by the same node[3]. When a node coordinates two consecutive barriers and the groups of nodes participating at these barriers are not identical, *barrier release* multicast flits might be interpreted in an ambiguous way:

Figure 6.3 illustrates the situation how it is intended to work. There are four nodes, half of them participating at the first barrier A and all are participating at the second barrier B. Both barriers are coordinated by node 0[4]. The corresponding communication flow is visualized in Figure 6.4.

Nodes 0 and 1 send their *barrier arrival* flits to the BCU of node 0, which then sends out a *barrier release* multicast flit. Next, all nodes send *barrier arrival* flits to the

[3] Restricting the software developer to have at most one hardware barrier on each node would be a too strong limitation.

[4] For better illustration, the BCU in Figures 6.4 and 6.5 has its own column. However, although this column makes the BCU look to be independent from the nodes, it belongs to node 0.

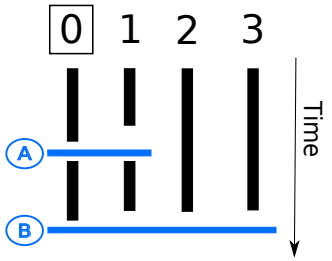


Figure 6.3.: Intended barrier participation

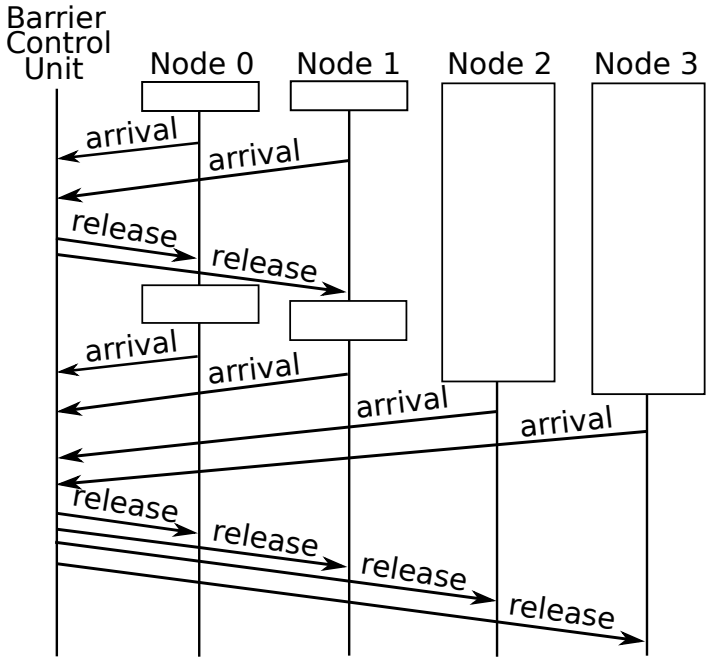


Figure 6.4.: Intended barrier-related communication: First, nodes 0 and 1 arrive at barrier A, which is then released. Second, all nodes arrive at barrier B, which is finally released.

BCU of node 0. Another *barrier release* multicast flit is finally sent out.

Figure 6.5 shows the situation that node 2 finishes its local computation early: Node 2 sends a *barrier arrival* flit to the BCU of node 0 before node 1 has arrived. The arrival status is stored in the `barrWaiting` register and not considered for barrier A due to the concept of the `barrExpected` register. After nodes 0 and 1 have signalized their arrival at barrier A, the BCU sends out a *barrier release* multicast flit. However, a multicast flit is internally implemented as a broadcast, which is only accepted by the nodes waiting for a multicast flit from the intended sender. Because node 2 waits for a *barrier release* multicast flit from node 0, this *barrier release* multicast flit is accepted by node 2 and is interpreted as release of barrier B there. As a result, node 2 continues its program execution too early.

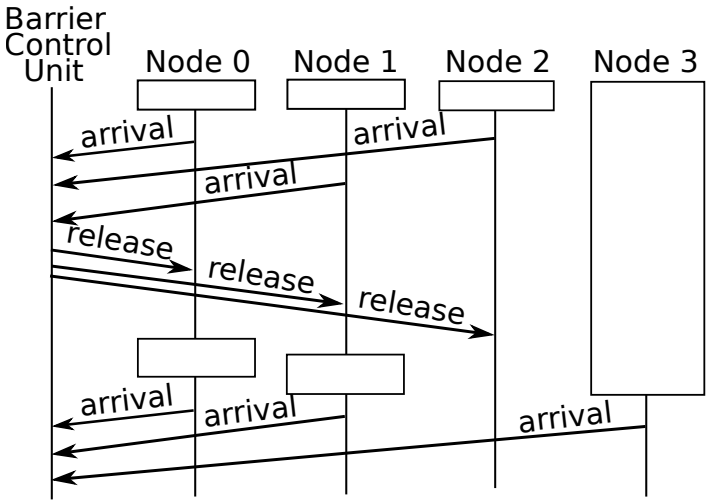


Figure 6.5.: Problem situation: Node 2 sends its barrier arrival flit too early. Because of the internal implementation of multicast flits, the *barrier release* multicast flit is then also received by node 2. As a result, node 2 continues its program execution too early.

To overcome this problem, we introduce a *unique id*, which has to be set for each barrier and is carried as payload of the *barrier release* multicast flit. Consequently, it ensures that a *barrier release* multicast flit is accepted only at the nodes participating at the current barrier. When arriving at a barrier via the `brrav` instruction, the node coordinating the barrier stores the *unique id* as payload of the *barrier release* multicast flit in the send buffer. On the receiver side, the *unique id* is implicitly stored in an additional register `awaitBarrID`, when `brrav` is executed. The nodes participating at a barrier compare the payload of each multicasted *barrier release* flit with the *unique id* stored in their register `awaitBarrID` and will only accept the one with the correct id. Choosing a *unique id* is up to the software developer. In an MPI implementation it may be integrated in the communicator.

The *unique id* allows to clearly distinguish different *barrier release* multicast flits: It enables to accept only the intended *barrier release* multicast flit. When coordinating a barrier, intermixed receipt of *barrier arrival* flits is also no problem due to the concept of the `barrWaiting` and `barrExpected` registers: All arriving *barrier arrival* flits set the bit of the corresponding sender node in the `barrWaiting` register. The coordinating node, where this BCU is installed, works off its different barriers in a fixed order. Thereby, it utilizes the `barrExpected` register to only respect the participating nodes in the `barrWaiting` register. Bits in the `barrWaiting` register which belong to other than the participating nodes are ignored. Furthermore, the 1:A schedule makes no restriction on the flits arriving at a node. Thus, it is not relevant if a node arrives at a barrier (too) early – *barrier arrival* flits can be handled

at any time.

6.4.3. Example for Complete Barrier Operation

The complete barrier initialization and communication flow as well as the state of the registers is visualized in Figure 6.6. In this example, there are only two nodes 0 and 1, where node 0 coordinates the barrier and node 1 participates at it. All further participants would behave the same way as node 1. Both nodes possess a PE and a BCU (besides other components which are omitted to keep the figure simple). In the columns of the PEs only the executed barrier-related instructions are shown and in the columns of the BCUs only the registers utilized by these nodes.

The figure is divided into three rows: In the first row *Init* only instructions initializing the required registers are illustrated. Thereby, the instruction `li t1, 0x214` loads a *unique id* `0x214` into register `t1` and `li t2, 0x3` prepares bits for the `barrExpected` register.

Configuration instructions are shown in the second row *Config*. They only have to be executed by the coordinating node. `cbrr zero,t2` selects the participating nodes in the `barrExpected` register. Should the NoC comprise more than 64 nodes, several calls to `cbrr` are necessary to configure the different parts of `barrExpected`. To avoid random matches between `barrWaiting` and `barrExpected` resulting in unwanted *barrier release* flits, `mbrr` should be employed. It enters the configuration mode before the `cbrr` calls and leaves it afterwards. Meanwhile, no comparison between `barrWaiting` and `barrExpected` takes place.

The last row *Arrival/Release* contains barrier arrival and release operations and the corresponding states of the registers. Before the instruction `brrav` can be executed, it is checked if the send buffer has free slots via `bsf self`, because `brrav` puts a new flit in it and would raise an exception when it is full. At the participating node 1, `brrav zero, t1` sends the *barrier arrival* flit[5] to the `barrWaiting` register of node 0 and stores the *unique id* implicitly in the local `awaitBarrID` register. At the coordinating node 0, the execution of `brrav` also puts a flit into the send buffer. However, here it is a *barrier release* multicast flit[6]. Moreover, the coordinating nodes' bit in the `barrWaiting` register is also set. When the comparison of the `barrExpected` and the `barrWaiting` registers indicates that all participating nodes have arrived and the BCU is not in configuration mode[7], the `barrReleasePermission` signal is set. It

[5] Internally, the `messageType` of a *barrier arrival* flit is called `barrival`, see details in Section 6.5 and Appendix B.1.

[6] The `messageType` of a *barrier release* multicast flit is called `barrelease` and further explained in Section 6.5 and Appendix B.1.

[7] A BCU is in configuration mode when `barrConfig` is 1. Then, no comparison between the `barrExpected` and `barrWaiting` registers takes place. Configuration mode is entered and left via the `mbrr` instruction, see Table 6.1.

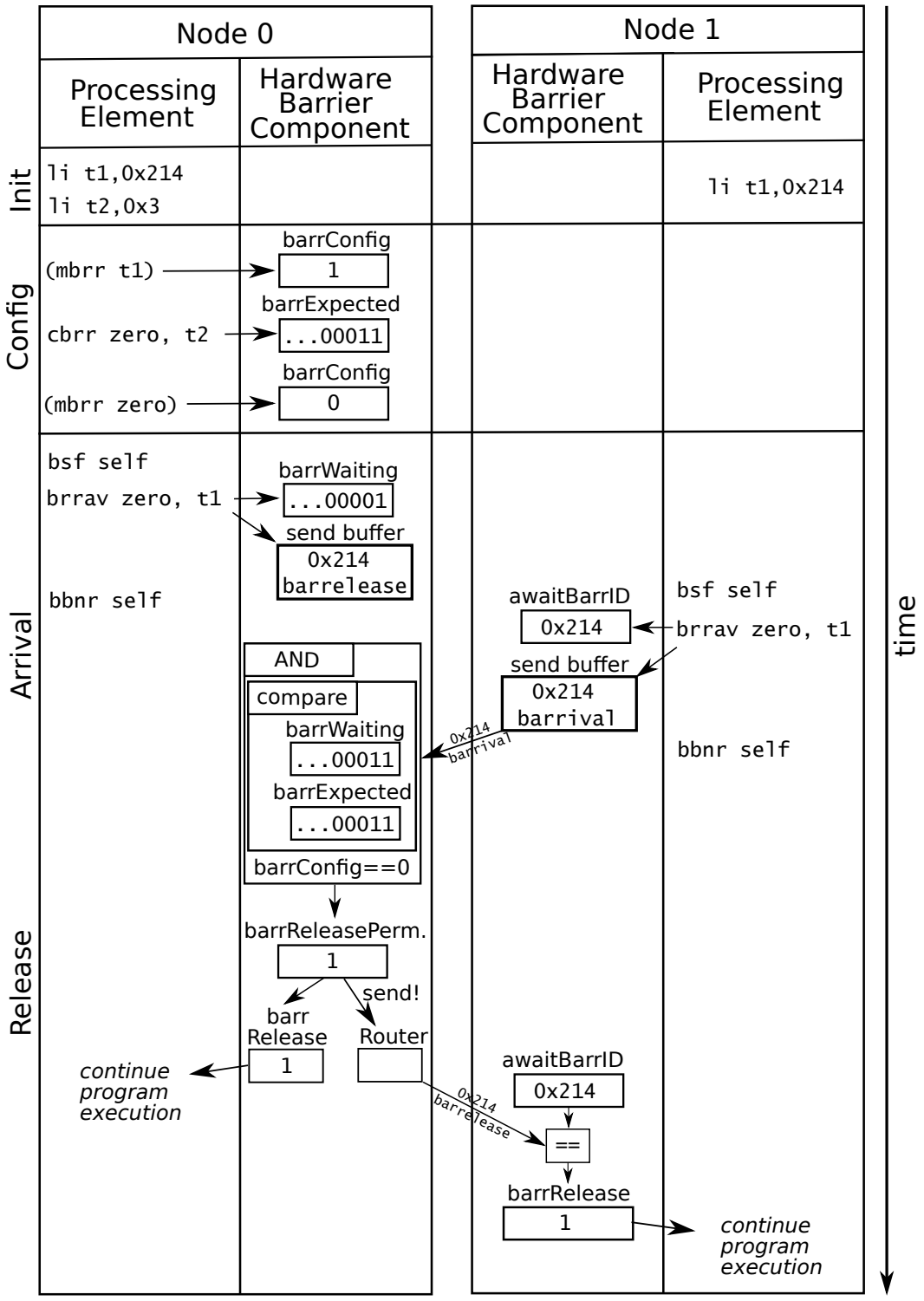


Figure 6.6.: Complete barrier initialization and communication flow: Node 0 coordinates the barrier, both nodes participate at it. Further participating nodes would behave the same way as Node 1.

tells the router to send out the *barrier release* multicast flit and sets the `barrRelease` register. After the *barrier release* multicast flit has arrived at the participating node 1 and its payload was successfully compared to the `awaitBarrID`, the `barrRelease` register is also set there. All nodes participating at a barrier (including the coordinating node) check the status of the `barrRelease` register via the `bbnr` instruction. When `barrRelease` is 0, the `bbnr` branch is taken (in our examples, it always jumps to itself). Otherwise, when `barrRelease` is 1, the barrier is released and program execution can continue. At the same time, `barrRelease` is reset to 0.

6.5. Hardware Implementation

In our hardware implementation, global barriers are implemented as special case of barriers for subsets of nodes as described in Section 6.4 to have a consistent programming model and behavior without too many distinctions. The actual implementation was carried out and refined by Bitterlich and Unte [BU19].

All barrier-related communication fits into single flits. Their transfer takes place via the 1:A NoC, which is also utilized for `ready`, `mcst` and data flits. For a clear separation of concerns, we introduce two new `messageTypes`: `barrival` for *barrier arrival* flits and `barrelease` for *barrier release* multicast flits. All flits with these `messageTypes` are handled by the BCU, while all other flits are handled as before. Flits of `messageType` `barrival` are sent from the participating nodes when executing the `brrav` instruction. They indicate that a participating node has arrived at a barrier. When a `barrival` flit arrives at the coordinating node, the node who sent the `barrival` flit is marked as arrived in the `barrWaiting` register of the BCU. When the coordinating node executes the `brrav` instruction, it puts a flit with the `messageType` `barrelease` and the *unique id* as payload into the send buffer. Moreover, it sets its own bit in the `barrWaiting` register. Because `brrav` is a variation of the `snd` instruction, it raises an exception when the send buffer is full. Thus, the status of the send buffer always has to be checked with the `bsf` or `bsnf` instruction before executing the `brrav` instruction.

Figure 6.7 illustrates the BCU and how it is integrated in an RC/MC node. Like the `ready` and hardware broadcast/multicast extension, the BCU is integrated in the execute stage of the pipeline. The send logic has to support the two new `messageTypes` `barrival` and `barrelease` (not seen in the figure) and is further extended with the new signal `barrReleasePermission`, which connects the BCU with the router. When a `barrelease` multicast flit is in the send buffer, it is only sent out when the `barrReleasePermission` signal is set. As long as the `barrReleasePermission` signal is not set and the `barrelease` flit is on the first position in the send buffer, no flit is sent out. The rest of the router is not changed except supporting the additional `messageTypes`. When flits are received, they are handled like before

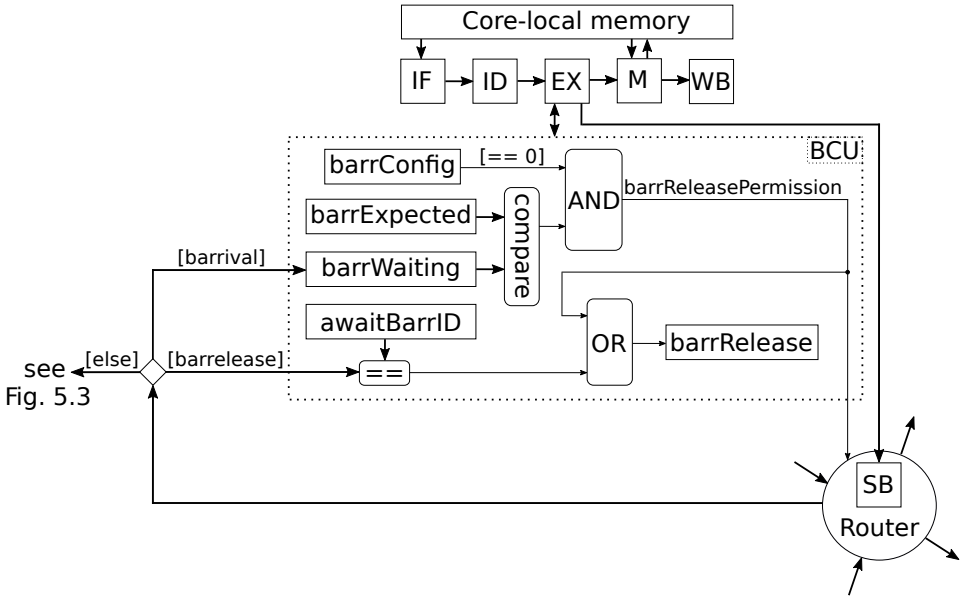


Figure 6.7.: Hardware structure and integration of the BCU in the node: When incoming flits have the `messageType` `barrival` or `barrelease`, they are forwarded to the BCU. There, incoming `barrival` flits set the corresponding bit of the sender node in the `barrWaiting` register or are compared to the *unique id* stored in the `awaitBarrID` register, respectively. When this comparison matches, the barrier is released. A barrier coordinated by the current node is released when the content of the `barrExpected` and `barrWaiting` registers match and the barrier is not in configuration mode (set via the `barrConfig` register). Arriving flits not having the `messageType` `barrival` or `barrelease` are handled like before in one of the cases seen in Figure 5.3. Thick arrows indicate the transport of several bits, while thin arrows represent only one bit.

except when they have one of the new `messageTypes` `barrival` or `barrelease`. In these cases, they are handed over to the BCU. Besides the already described signal `barrReleasePermission` to the router and the acceptance of `barrival` and `barrelease` flits, the BCU is only connected to the execute stage – more precisely, it is part of the execute stage to allow fast access to its registers.

Inside of the BCU, the registers `barrExpected` and `barrWaiting` work as described in Subsection 6.4.1. They are connectd via a `compare` logic module. In the same Subsection 6.4.1, the concept of the configuration mode was explained. It is implemented via the register `barrConfig`. When a barrier is not in configuration mode and both the `barrExpected` and `barrWaiting` registers match, the `barrRelease-Permission` signal releases a barrier coordinated by the current node. The barrier release is locally recognized via the `barrRelease` register. It is alternatively triggered

when a barrelease multicast flit is received and its payload matches with the *unique id* stored in the awaitBarrID register (see Subsection 6.4.2).

Table 6.2.: Encoding of barrier related instructions in our RISC-V instruction set extension.

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	100	00000	1011011	brrav
imm[12|10:5]	00000	00000	110	imm[4:1|11]	1111011	bbnr
0000000	rs2	rs1	111	00000	1011011	cbrr
0100000	00000	rs1	111	00000	1011011	mbrr

To control the BCU, the instructions described in Table 6.1 are used. Their encoding is illustrated in Table 6.2. They allow direct access to the following registers: (i) barrRelease via the bbnr instruction[8], (ii) barrConfig via the mbrr instruction, (iii) barrExpected via the cbrr instruction and (iv) awaitBarrID via the brrav instruction (only for participating nodes). At the barrWaiting register, only the nodes' own bit can be directly accessed via the brrav instruction. All other bits are set by the other nodes by sending a barrival flit to the BCU of the coordinating node also via the brrav instruction. The bits in the barrWaiting register are reset when the barrelease multicast flit is sent out (not further illustrated in Figure 6.7). Thereby, only the bits set in the barrExpected register are respected. Due to the generic implementation of our VHDL model, it is not ensured that not-existing nodes cannot be configured via the cbrr instruction (e.g. a configuration of nodes 17–63 would be possible on a 16 node RC/MC processor).

As already described in the previous Section 6.4, the BCU internally works as following: When the nodes set in the barrExpected register are also set in the barrWaiting register (checked with the compare unit), all participants have arrived at the current barrier. It has to be further checked that the BCU is not in configuration mode. Only when all of these preconditions are fulfilled, the barrReleasePermission signal is set. Then, the router is allowed to send out the barrelease multicast flit which was put into the send buffer when the coordinating node executed the brrav instruction. Always when the barrelease multicast flit arrives at the other nodes, its payload is compared to the stored awaitBarrID to ensure that only the barrelease multicast flit from the corresponding barrier is accepted. When they match, the barrRelease register is set and the node can con-

[8] Like at bnr and the Ready Bit Array, the barrRelease signal is automatically cleared when it is read by bbnr and was set before (i.e. when the PE gets the information that it can continue program execution, barrRelease is automatically reset).

tinue its program execution. The `barrRelease` register is also set when the barrier coordinated by the local BCU is released.

After all participating nodes walked through a barrier, the barrier may be reused without the need of re-configuration when the same nodes meet again. Should the group of participating nodes change, the *unique id* has to be changed and the barrier reconfigured. For the capability of the barrier to be reused, the register `barrExpected` should remain unchanged, while the `barrRelease` register and the corresponding bits in the `barrWaiting` register of the coordinating node need to be reset. Both are generally reset automatically at all barriers: At the `barrWaiting` register, the corresponding bits are reset when the barrier is released, i.e. when the `barrrival` multicast flit is sent out. The `barrRelease` signal is reset when the `bbnr` instruction gets the barrier release status and the PE knows that it can continue its program execution. Actually, there is no need to clear the `awaitBarrID` register, but it might be useful for easier debugging.

6.6. Programming model

Figure 6.6 already illustrated the communication flow when nodes process a barrier, but also some instructions executed by the different nodes. The corresponding perspective of the software developer is visualized in the Code examples 6.1 (participating nodes) and 6.2 (coordinating node).

Code example 6.1 Code executed by the participating nodes
(Node 1 at example illustrated in Figure 6.6).

```
      ...                  # code executed before barrier
      li    t1, 0x214      # init: unique barrier id
w4sb: bsf   w4sb           # wait for send buffer before sending flit
      brrav zero, t1       # arrive at barrier with unique id 0x214, ...
                           # ... which is coordinated by node 0
self: bbnr  zero, t1, self # busy waiting for barrier release
      ...                  # code executed after barrier
```

At the participating nodes (Code example 6.1), only the *unique id* of the barrier has to be specified (`li`) and sent to the coordinating node (`brrav`). Afterwards, the branch instruction `bbnr` is utilized for busy waiting until the barrier is released. It might also jump to some other code to be executed while waiting, which we usually avoid to ensure good timing behavior (low WCET and good predictability) and keep code examples simple.

Code example 6.2 representing coordinating nodes works the same way as Code example 6.1 besides two additional instructions between `li` and `bsf`. This is due to the node being participating and coordinating node at the same time. The additional

Code example 6.2 Code executed by the coordinating node
(Node 0 at example illustrated in Figure 6.6).
`brrav` and `bbnr` are exactly the same as for participating nodes (Code example 6.1)

```
      ...                  # code executed before barrier
      li    t1, 0x214      # init: unique barrier id
      li    t2, 0x3        # init: participating nodes 0 and 1
      cbrr  zero, t2       # store participating nodes in barrExpected
w4sb: bsf   w4sb           # wait for free slot in send buffer
      brrav zero, t1       # arrive at barrier, set unique id 0x214
                           #   (barrier is coordinated at own HBC;
                           #      same call as for participating nodes)
self: bbnr  zero, t1, self # busy waiting for barrier release
      ...                  # code executed after barrier
```

instructions are for barrier coordination: A second `li` instruction prepares the bits to be written into the `barrExpected` register to select the participating nodes. The `barrExpected` register is configured via the `cbrr` instruction. As Code example 6.2 is intended to work with up to 64 nodes, one single call to `cbrr` is sufficient. An example with more than 64 nodes is illustrated later in Code example 6.3.

Code example 6.3 brings the code for participating and coordinating nodes together into one single function `barrier`, which can be called by any node. It is tailored to work with 256 nodes, but might be adapted for larger architectures by replacing the registers `a2` to `a5` with the address of an array which holds the information which nodes participate. Before the function is called, the parameter registers `a0` to `a5` are set for a global barrier with the coordinating node 0 and the *unique id* `0x214`. At the beginning of the function `barrier`, a `csrr` and a branch are executed to check if the current node is the coordinating node (for details on the `csrs`, see Section 3.4). When it is, the PE jumps to the configuration code at `cfg`. Otherwise the PE executes the following lines, where the code for arriving at a barrier is located.

Because the code for barrier arrival is almost the same as in Code example 6.1 (the registers are changed due to the implementation as function and the jump label is adapted), we now have a closer look on the code for the coordinating node: As the *unique id* and the participating nodes are now passed as parameters, there is now only one `li` instruction, which is used to address the different parts of the `barrExpected` register[9]. Since the `barrExpected` register cannot be changed at once, but requires several calls to the `cbrr` instruction, comparisons between the `barrWaiting` and `barrExpected` registers are temporarily disabled via the `mbrr` instruction. Thereby, it is sufficient to re-enable the comparison via `mbrr zero` before jumping to `brav`,

[9]The `barrExpected` register works the same way as the `bnra` instruction in Figure 5.2 on page 58. Thus, its different parts have to be addressed with their number 0, 1, 2 or 3.

because the barrier will not yet be released due to the coordinating node *barrier arrival* taking place with the `brrav` instruction later. The rest of the function works the same way as Code example 6.2, besides the jump from configuration code to barrier arrival (at `barv`) and the `ret` instruction at the end, which was added due to the implementation as function.

From the timing perspective, Code example 6.3 brings together all aspects of our hardware-supported barrier to achieve a good WCET analyzability: All instructions have a fixed latency, the only dynamic behavior is found in the branch instructions. When executing the `beq` instruction after the `csrr` instruction, the branch delay only has to be added for the coordinating node, while the participating nodes can immediately proceed as `bsf` and `brrav` are already in the next pipeline stages. Although the coordinating node has more code to execute, we optimized Code example 6.3 for faster barrier arrival of the participating nodes, because the transport of *barrier arrival* flits over the NoC lies on the worst-case path[10]. The number of iterations of our new `bbnr` instruction is dependent how long the participating nodes need to arrive at the barrier. This can be computed based on the NoC schedule and the WCETs of the sequential code parts, see Section 3.1. Altogether, we have good timing analyzability: Only the execution times of the instructions have to be added and the waiting time for the participating nodes has to be integrated.

[10]Configuration code at `cfg` requires fixed twelve cycles (1 cycle per instruction and two cycles branch delay), while flit transfer takes already 24 cycles in a 4x4 node 1:A NoC.

Code example 6.3 Function `barrier` for 256 node NoCs. Here, it is called with parameters for a global barrier with *unique id* `0x214` coordinated by node 0.

```
set_parameters_for_barrier:
# global barrier with coordinator node 0 and unique id 0x214
      li    a0, 0           # init: set coordinator node
      li    a1, 0x214       # init: unique barrier id
      li    a2, -1          # init: all nodes should participate
      li    a3, -1          #           (-1: set all bits)
      li    a4, -1          #
      li    a5, -1          #
      jal   barrier         # call function barrier

.globl barrier
barrier:                    # barrier function for 256 nodes
                            # expects coordinating node in a0 ...
                            # ... unique id in a1 and ...
                            # ... node selection in a2 - a5
      csrr t0, cid          # get own core id
      beq  a0, t0, cfg      # when node is coordination node ...
                            # ... jump to code for configuration

      # barrier arrival code for participants including coordinator node
barv: bsf   barv            # wait for free slot in send buffer
      brrav a0, a1          # arrive at barrier with unique id in a1
                            #       (coordinated by node in a0)

      # barrier release code for participants including coordinator node
self: bbnr  a0, a1, self    # busy waiting for barrier release
      ret                   # barrier released, continue prg. execution

      # configuration code only for coordinator node
cfg:  li    t0, 1           # init: for addressing parts of barrExpected
      mbrr  t0              # config: enter configuration mode
                            # (no comp. of barrWaiting and barrExpected)
      cbrr  zero, a2        # config: store part. nodes in barrExpected#0
      cbrr  t0, a3          # config: store part. nodes in barrExpected#1
      addi  t0,t0,1         # t0 should now address pt. 2 of barrExpected
      cbrr  t0, a4          # config: store part. nodes in barrExpected#2
      addi  t0,t0,1         # t0 should now address pt. 3 of barrExpected
      cbrr  t0, a5          # config: store part. nodes in barrExpected#3
      mbrr  zero            # config: leave configuration mode
      j     barv            # jump to code for arriving at barrier
```

6.7. Hardware Costs

We now estimate the hardware costs of our implementation in Subsection 6.7.1 and compare our expectations with the actual hardware costs in Subsection 6.7.2.

6.7.1. Expected Hardware Costs

As all RC/MC components, our hardware barrier extension is designed with the goal of low additional hardware effort. At hardware level, the following extensions have to be implemented (cf. Figure 6.7):

First, the new `messageTypes` `barrival` and `barrelease` require 1 additional bit between the nodes. Altogether, three bits are required to support the `messageTypes` `none`, `data`, `ready`, `mcst`, `barrival` and `barrelease`. An overview on them is given in Appendix B.1. These `messageTypes` also have to be supported in the decode and execute stages and the send buffer. Moreover, we require additional circuitry to distinguish between these `messageTypes` at a receiver node and forward the relevant data to the corresponding hardware components (BCU for `barrival` and `barrelease` flits). Each node comprises a BCU with five additional registers: `barrConfig` and `barrRelease` are one bit registers, `awaitBarrID` is 64 bits wide[11] and the width of `barrExpected` and `barrWaiting` directly correlates with the number of nodes on the chip. Thereby, `barrExpected` and `barrWaiting` are coupled tightly together to enable the comparison releasing the barrier. In Figure 6.7 we illustrated this with the `compare` unit, which should contain an `AND` logic gate and a comparator (when `barrExpected AND barrWaiting == barrExpected` then set output bit of `compare`). It is followed by another `AND` logic gate, which additionally takes the negation of `barrConfig` as input and sets the `barrReleasePermission` signal when both inputs are set. The `barrReleasePermission` signal is one of two ways to set the `barrRelease` register. Another way is when an arriving `barrelease` multicast flit is successfully compared to the value in the `awaitBarrID` register via a comparator. Both ways set one bit of the `OR` logic gate which controls the `barrRelease` register. For setting bits in the `barrWaiting` register, a multiplexer is required. Automatic reset of the `barrWaiting` register takes place via a `XOR` logic gate (`barrWaiting xor barrExpected`). The registers `barrExpected`, `barrConfig` and `awaitBarrID` are directly written by the execute stage.

Beyond that, the instructions from Tables 6.1 and 6.2 have to be implemented: As suggested in Section 6.3, we provide a modified send instruction `brrav (barrier arrival)`, which behaves like the `snd` instruction with the following differences: First, it sets the `barrival` `messageType`. Second, it stores the *unique id* in the `awaitBarrID` register. At the coordinating node, it places a `barrelease` flit in the

[11] 16 bits would also be enough, but we decided to have 64 bits, because we have a 64 bit architecture.

send buffer, having the *unique id* as payload. Thereby, the `barrelease` flit behaves like a `mcst` flit except having another type. Thus, it does not impose further overhead at the routing in the NoC. Our additional branch instruction `bbnr (branch if barrier not released)` works similar to the `bnr` instruction from Subsection 4.5.2 – instead of checking a bit in the `Bit Array` it checks the `barrRelease` register. The remaining instructions `cbrr (configure barrier)` and `mbrr (mode of barrier)` directly write to their corresponding registers `barrExpected` and `barrConfig`.

These extensions add up as following: Our additional `messageTypes` `barrival` and `barrelease` require one bit and add to the other signals which connect nodes: 64 bit for data, 8+8 bit for sender + receiver information (in a 16x16 NoC), two bits for the already existing `messageTypes`. Altogether, these are bit lines with a width of 82 bit and we add one bit, which is close to 1 %. For handling `barrival` and `barrelease` flits, the decode and execute stages as well as the send buffer have to be extended with the same costs. The additional register `awaitBarrID` is 64 bits wide, while the width of the registers `barrWaiting` and `barrExpected` correlates with the number of nodes on the chip, i.e. 1 bit for each node. To connect these registers, two comparators and a `AND` logic module are required. Furthermore, one bit is needed each for the `barrRelease` and `barrConfig` registers as well as the `barrReleasePermission` signal. For connecting these registers and signals, we require another `AND` logic module as well as an `OR` logic module. Furthermore, some multiplexers are needed (i) to distinguish `messageTypes` and (ii) to set the corresponding bit in the `barrWaiting` register. For resetting the `barrWaiting` register, a `XOR` logic gate is also required. The complete BCU and its integration into the node was already illustrated in Figure 6.7. For working together, our new instructions have to be integrated into the decode and execute stages, which should only be minor changes.

When looking at the complete BCU and comparing it with the other extensions, it should roughly cost the same as the broadcast/multicast extension.

6.7.2. Actual Hardware Costs

Bitterlich and Unte integrated the hardware supported barrier into the existing VHDL model and estimated hardware costs as described in Section 3.5 [BU19]. Their result are around 130 ALMs and 220 registers per node (for an RC/MC processor with 2x2 and 4x4 nodes). Compared with the hardware broadcast/multicast extension, we need less than half of the ALMs, but 50% more registers. This may come from reduced management effort (e.g. no need to count flits at the barrier), but more information to store: Especially `awaitBarrID` and the two new hardware bit arrays require a lot of registers. The total cost of a node including all of our hardware extensions is 2615 ALMs and 2929 registers on a 2x2 RC/MC processor or 3350 ALMs and 3777 registers for 4x4 nodes. Thus, the overhead of our hardware barrier extension is around 4-5% at the ALMs and 6-7% at the registers. Since the

total size of a node increases with larger core numbers and the required ALMs and registers remain very slow, the overhead imposed by our hardware barrier extension decreases for larger node numbers.

6.8. Evaluation: Worst-Case Performance

For the evaluation, we carry out timing analyses for different implementations of `MPI_Barrier`. In this operation, one node coordinates a barrier, where several nodes participate. It is defined in the MPI standard [Mes15] and was already introduced in Subsection 3.3.2. `MPI_Barrier` is a crucial operation in parallel programs, e.g. that all participants wait until the initialization or some computation is finished before further data exchange between nodes takes place.

In the next Subsection 6.8.1, we conduct timing analyses of different implementations of `MPI_Barrier`. Afterwards, we compare their worst-case performance in Subsection 6.8.2. Because the timing analysis might have introduced pessimism and overestimation, we assume optimal code optimization at the theoretical comparison in Subsection 6.8.3.

6.8.1. Timing Analysis of Different `MPI_Barrier` Implementations

We first analyze our implementation of `MPI_Barrier` with hardware support as it was elaborated in this chapter. Afterwards, an analysis for `MPI_Barrier` implementing the Bruck Algorithm [BHK+97] takes place.

Timing Analysis of `MPI_Barrier` with Hardware Support

`MPI_Barrier` with hardware support works as illustrated in Figure 6.8 and described in Table 6.3. Bascially, it follows the principles described in Subsection 6.4.3 and already displayed in Figure 6.6. First, the root node has to configure its BCU. Then, it arrives at the barrier. Thereby, the root node does not have to send a *barrier arrival* flit via the NoC like the participant node(s) do. Instead, it directly tells its BCU that it has arrived. When all nodes have arrived, the BCU sends out a *barrier release* flit and all nodes can continue their program execution. We illustrated only one participant node – all others behave exactly the same way and have the same timing.

For easier handling, the workflow of `MPI_Barrier` with hardware support may be divided into two phases: First, we have *barrier arrival* at steps (A) to (E): Configuring the BCU (only for the root node) and arriving at the barrier. Second, when all nodes (including the root node) have arrived, we have *barrier release* at step (F). The WCTT for barrier arrival is the maximum of the processing time of the root node (steps (A), (D) and (E)) and the time the participant node(s) require to deliver their *barrier arrival* flit at the BCU (steps (B) and (C)). For the phase of *barrier release* at step (F),

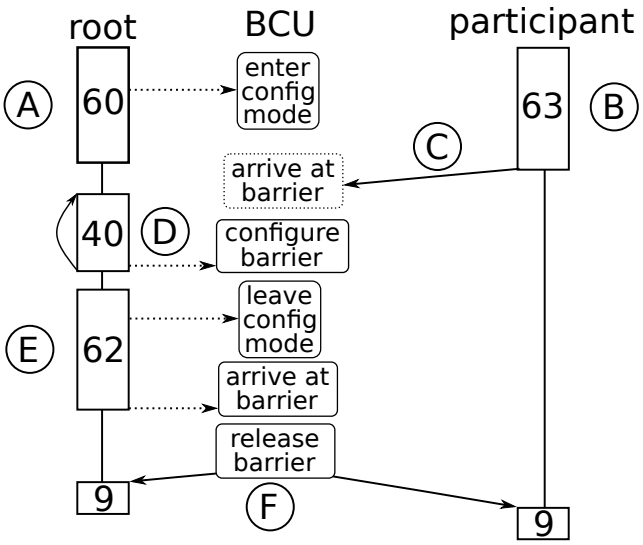


Figure 6.8.: Workflow and timing behavior of `MPI_Barrier` with hardware support as already illustrated in Figure 6.6. While the root node configures the barrier, the participant node(s) may already arrive. Then, they wait until the barrier is released by the BCU. There may be more participant nodes, but we included only one to keep the figure simple. Other participant nodes would behave exactly the same way as the participant node in the figure. Boxes represent local code execution, the continuous arrows at steps (C) and (F) denote flits, dotted arrows represent local interaction at the root node and its BCU, arrows from the end of a box to the beginning of the same box illustrate loops. The box at step (C) is dotted, because barrier arrival from participant node(s) does not necessarily take place at this point in time – nodes may arrive at the barrier at an arbitrary point in time. Numbers inside of the boxes specify the WCET estimation of this code part. The meaning of the different steps is described in Table 6.3.

Table 6.3.: Description of structure and WCET estimates for the implementation of `MPI_Barrier` with hardware support as shown in Figure 6.8.

step	description	WCET estimate
(A)	During the initialization, the root node activates the configuration mode of its BCU.	60
(B), (C)	After determining the root node, each participant node sends a *barrier arrival* flit there.	$63 + WCTT(1,1)$
(D)	The root node tells its BCU which nodes participate. Thereby, the configuration for 64 nodes can be set at once. Thus, this step has to be executed in a loop for each 64 nodes (e.g. 4 times when 256 nodes are present).	$\lceil \frac{N}{64} \rceil \cdot 40$
(E)	After the BCU was configured, the root node disables the configuration mode and arrives at the barrier.	62
(F)	When all nodes have arrived at the barrier, it is released by the BCU and all nodes execute the end of the barrier function. Due to the concept of the integrated hardware broadcast, only the WCTT of one single flit has to be considered.	$\max(9, WCTT(1,1) + 9)$

we take the maximum of the time it takes for the BCU to notify the root node and to notify the participant node(s). When we add the corresponding WCET estimates from Table 6.3, we get $WCET_{Table}^{HwBarrier}$ in Formula 6.1.

$$\begin{aligned} WCET_{Table}^{HwBarrier} = \; & \max(60 + \left\lceil \frac{N}{64} \right\rceil \cdot 40 + 62, 63 + WCTT(1,1)) \\ & + \max(9, WCTT(1,1) + 9) \end{aligned} \tag{6.1}$$

Simplifying both max terms and summarizing the rest, we get the total WCET estimate $WCET_{Total}^{HwBarrier}$ in Formula 6.2.

$$WCET_{Total}^{HwBarrier} = \; 72 + \max(59 + \left\lceil \frac{N}{64} \right\rceil \cdot 40, WCTT(1,1)) + WCTT(1,1) \tag{6.2}$$

Timing Analysis of `MPI_Barrier` implementing the Bruck Algorithm

As we have already seen in Subsection 5.8.1, the management of a tree imposes a high overhead. For barriers, a more efficient implementation is available realizing the Bruck Algorithm [BHK+97] as already described in the introduction in Section 6.1. It is illustrated in Figure 6.9 and the steps in the figure are explained in Table 6.4.

Table 6.4.: Description of structure and WCET estimates for a implementation of `MPI_Barrier` implementing the Bruck Algorithm as shown in Figure 6.9.

step	description	WCET estimate
(A)	Initialization code is executed.	32
(B)	After determining the communication partners, a `ready` flit is sent out.	73
(C)	The parameters for the `branch if not ready` instruction are set.	2
(D)	When the `ready` flit from the corresponding communication partner has arrived, we can check for the next loop iteration.	12
(E)	Steps (B) to (D) are executed in a loop $\lceil log_2 \chi \rceil$ times, where χ is the number of nodes to arrive at the barrier.	$\lceil log_2 \chi \rceil$ times
(F)	After the last `ready` flit was received, the operation can terminate.	7

After the initialization at step (A), the next communication partner is determined and a flit is sent there in step (B). Two cycles later (C), the node waits for the flit from some other node (D). Steps (B) to (D) are repeated $\lceil log_2 \chi \rceil$ times. Thereby, χ is

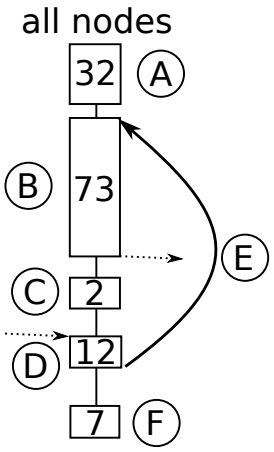


Figure 6.9.: Workflow and timing behavior of `MPI_Barrier` implementing the Bruck Algorithm. A node sends a `ready` flit to one other node and waits for the `ready` flit from another node. Each iteration the communication partner is changed to one which has knowledge about more other nodes. When the `ready` flit in the last iteration arrives, the node can be sure that all other nodes are ready for barrier release. Because there is no distinction between root and participant nodes, all nodes behave exactly the same way and execute the same code. Therefore, we only displayed one node in the figure. Boxes represent local code execution, dotted arrows denote `ready` flits, arrows from the end of a box to the beginning of a box illustrate loops. Numbers inside of the boxes specify the WCET estimation of this code part. The meaning of the different steps is described in Table 6.4.

the number of nodes arriving at the barrier. Each time communication partners are changed in a clever way, doubling the information about other nodes. Finally, the node has the information that all other nodes are ready to release the barrier and can continue program execution after step (F).

For formulating a WCET estimation, we sum up all steps from Table 6.4 and also respect the loop iteration at step (E). Then, we get $WCET_{Table}^{BarrierBruck}$ in Formula 6.3 as result.

$$WCET_{Table}^{BarrierBruck} = 32 + \lceil log_2 \chi \rceil \cdot (73 + \max(2, WCTT(1,1)) + 12) + 7 \tag{6.3}$$

Because we always have to wait for the `ready` flit to arrive, we have to respect the WCTT and the code execution time at *all* iterations. However, because the WCTT is always larger than two cycles, we do not need the term $\max(2, ...)$. Altogether, we can summarize $WCET_{Table}^{BarrierBruck}$ to $WCET_{Total}^{BarrierBruck}$ in Formula 6.4.

$$WCET_{Total}^{BarrierBruck} = 39 + \lceil log_2 \chi \rceil \cdot (85 + WCTT(1,1)) \tag{6.4}$$

6.8.2. Comparison of WCET Estimates

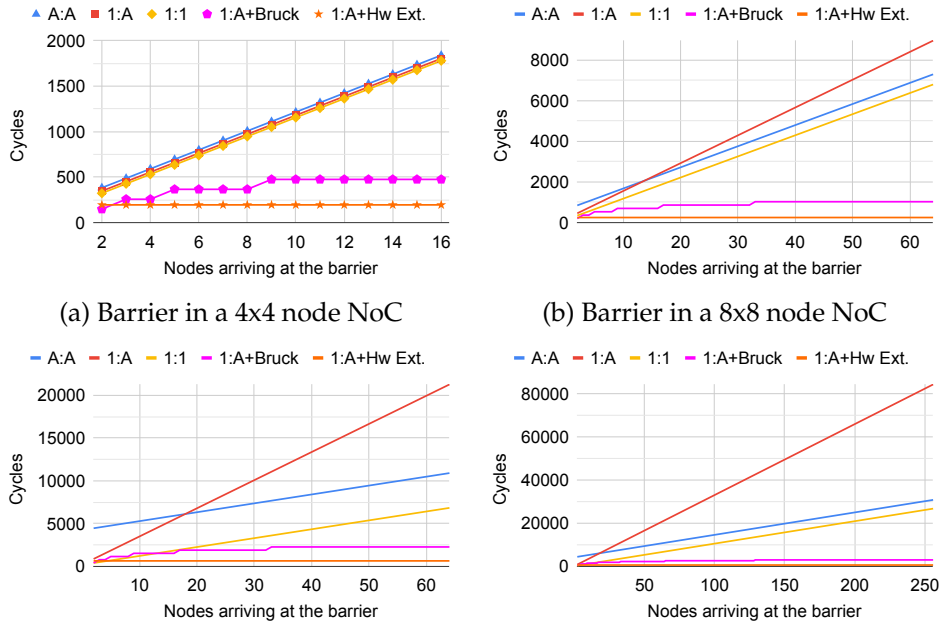
After estimating the WCETs of different implementations of `MPI_Barrier`, we now compare their worst-case performance. For this comparison, we also consider the simple implementation from Section 3.6. To achieve better comparability, we assemble Formula 3.6 as $WCET^{SimpleBarrier}$ in Formula 6.5. Thereby, χ represents the number of nodes arriving at the barrier like at $WCET_{Total}^{BarrierBruck}$ (Formula 6.4).

$$\begin{aligned} WCET^{SimpleBarrier} = &\max(9, WCTT(1,1)) + 73 \cdot \chi + 122 \\ &+ \max(31 + WCTT(1, \chi - 1), 31 \cdot \chi - 31 + WCTT(1,1)) \end{aligned} \tag{6.5}$$

For our evaluation, we consider an RC/MC processor with 4x4 = 16 nodes, 8x8 = 64 nodes and 16x16 = 256 nodes. We evaluate how long it takes when different numbers of nodes participate at a barrier. Thereby, we combine the WCET estimates with the generic schedules 1:1, A:A, the original 1:A schedule without hardware extensions and our extended 1:A schedule with hardware support for barriers in the following way:

1. $WCET_{Total}^{HwBarrier}$ (Formula 6.2) utilizes the 1:A schedule with hardware support for barriers.

2. $WCET_{Total}^{BarrierBruck}$ (Formula 6.4) can only be combined with the original 1:A schedule, because the Bruck implementation relies completely on `ready` flits



(a) Barrier in a 4x4 node NoC

(b) Barrier in a 8x8 node NoC

(c) Up to 64 nodes in a 16x16 node NoC

(d) Barrier in a 16x16 node NoC

Figure 6.10.: WCET estimates for barriers in NoCs with 4x4, 8x8 and 16x16 nodes.

and therefore cannot run on the 1:1 schedule. A:A also does not make sense, because it is able to send out flits to all nodes in the same period.

3. $WCET^{SimpleBarrier}$ (Formula 6.5) utilizes the generic schedules 1:1, A:A and the original 1:A schedule without hardware extensions.

Our results are displayed in Figure 6.10. At the 4x4 node NoC in Figure 6.10a, the simple barrier implementation has similar WCET estimates for all schedules, because the WCET is code-driven. For the Bruck algorithm, we can clearly see the "steps" each time we pass powers of two at the number of nodes arriving at the barrier. It is noticeable that the Bruck algorithm's WCET estimate is below the WCET estimate of the 1:A schedule with our hardware barrier extension when only two nodes arrive at the barrier. This is because of the very short code of `MPI_Barrier` implementing the Bruck algorithm. In all other cases, the 1:A schedule with our hardware extensions is faster than all other implementations. It remains at a constant low level due to low communication effort and only few code to be executed.

Figure 6.10b illustrates barrier participation in a 8x8 node NoC. Our 1:A schedule with hardware barrier extension is still at a low level, only outperformed by the 1:A schedule with Bruck algorithm in the case when two nodes arrive at the barrier. At the Bruck algorithm, we can see the continuation of the "steps" after 16 and 32 nodes arrive at the barrier. When considering the simple barrier implementation, 1:1 and A:A perform similar to the case of the 4x4 node NoC from Figure 6.10a: Their

WCET estimates are code-driven and thus remain parallel to each other. However, the WCET estimate of the simple barrier implementation utilizing the 1:A schedule increases faster, because it is driven by the WCTT of the 1:A schedule.

Finally, barriers in a 16x16 node NoC are displayed in Figures 6.10c and 6.10d. Thereby, Figure 6.10d visualizes the complete range from 2 to 256 nodes arriving at the barrier, while Figure 6.10c shows only the range from 2 to 64 nodes. Our 1:A schedule with hardware barrier extension can only be seen in the left picture, because it stays at its very low level all time, which is too low to be recognized in Figure 6.10d. The same holds for the "steps" of the Bruck algorithm utilizing the 1:A schedule. At the simple barrier implementation, the A:A and 1:1 schedules are code-driven again. However, their distance between each other has increased due to their WCTTs. The simple barrier implementation running on the 1:A schedule is WCTT driven like in the 8x8 node NoC. In the case of 256 nodes participating at the barrier, its WCET is almost 27 times the WCET of the Bruck algorithm and 130 times the WCET of the 1:A schedule with hardware support for barriers.

Altogether, the WCET of our 1:A schedule with barrier hardware extension almost always performs best. Only in the case of two nodes arriving at a barrier, the Bruck algorithm utilizing the 1:A schedule is better. Furthermore, the Bruck algorithm has the lowest WCET estimates when not having hardware support for barriers.

6.8.3. Theoretical Comparison

The WCET analysis in Subsection 6.8.1 is performed with compiler optimizations disabled (compiler optimization level `-O0`) and may be improved. Therefore, we carry out a theoretical comparison in this section to see the impact of our hardware extension when code would be optimized in an optimal way. For this, we consider only transportation times and assume code execution times to be zero.

Table 6.5.: How large is the WCTT for a barrier operation where χ nodes arrive at a barrier in a NoC with a dimension of n?

schedule	barrier arrival	barrier release
One-to-One (1:1)	1 period 1:A $= n^2 + 2n$	$(\chi - 1)$ periods 1:1 $= n \cdot (\chi - 1) + 2n$
One-to-All (1:A) with Bruck	$\lceil log_2 \chi \rceil$ periods 1:A $= n^2 \cdot \lceil log_2 \chi \rceil + 2n$	
All-to-All (A:A)	1 period A:A $= \left(\frac{n^2(n-1)}{2} + 2\right) + \frac{n^2}{2} + 2n$	1 period A:A $= \left(\frac{n^2(n-1)}{2} + 2\right) + \frac{n^2}{2} + 2n$
One-to-All (1:A) with hardware barrier extension	1 period 1:A $= n^2 + 2n$	1 period 1:A $= n^2 + 2n$

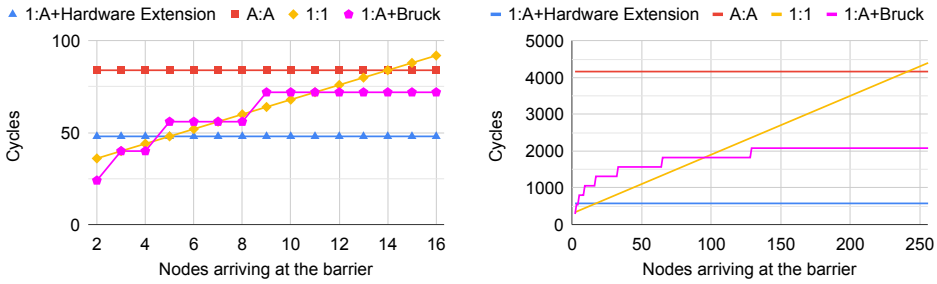
Table 6.5 summarizes how many periods are required for *barrier arrival* and *barrier release* at the different schedules. Following the investigations of Auer [Aue18], we assume barrier arrival to be implemented as sending a `ready` flit when no explicit hardware support is present. In the first line of each schedule, it is stated how many periods are required when χ nodes arrive at a barrier. Then, in the second line, the corresponding formula is given how many cycles are needed for transmission of `ready` and data flits. Similar formulas were already utilized for the theoretical comparison of our hardware broadcast/multicast extension in Subsection 5.8.3. They were also described in Subsection 3.2.2 except the formula for the Bruck algorithm, which was explained in the introduction of this chapter in Section 6.1.

As already mentioned in Subsection 5.8.3, a dedicated `ready` NoC is necessary when employing the 1:1 schedule. Therefore, the 1:A schedule is required for *barrier arrival*, while the 1:1 schedule is employed for *barrier release*. Thereby, *barrier release* flits are sent out in a `for` loop, iterating over arriving nodes.

At the 1:A schedule with Bruck algorithm, *barrier arrival* and *barrier release* are integrated into one single formula, because the nodes only exchange information about the state of the barrier via `ready` flits.

We assume that *barrier release* information can be sent out in one period at the A:A schedule. Therefore, one period of the A:A schedule is sufficient each for arriving at and releasing a barrier.

Finally, the 1:A schedule including our new hardware barrier extension is presented. It requires one period for *barrier arrival* and one for *barrier release*.



(a) Barrier in a 4x4 node NoC

(b) Barrier in a 16x16 NoC

Figure 6.11.: Barriers in 4x4 and 16x16 node NoCs.

Figure 6.11 displays the results of our theoretical comparison. Thereby, Figure 6.11a illustrates a small 4x4 NoC with 16 nodes, while Figure 6.11b shows a large NoC with 256 nodes. In both figures, we see that *1:A+Hardware Extension* remains at a constant low level, while A:A remains at a constant high level. The 1:1 schedule starts with a lower WCTT than our 1:A schedule with hardware barrier extension for few nodes arriving at a barrier. Then, the 1:1 schedule increases steadily and ends with a WCTT higher than A:A when many nodes arrive at the barrier. At the 4x4

node NoC in Figure 6.11a, the 1:A schedule with Bruck algorithm performs similar to the 1:1 schedule, but with the typical "steps" at the powers of two and staying better than A:A. In a larger NoC, its performance is worse for small to medium numbers of nodes arriving at a barrier, but better for larger numbers. Altogether, our hardware supported barrier is not most efficient when only few cores arrive at a barrier. But for larger barriers, our approach always performs best. 1:A with Bruck algorithm is often the second best solution, but for low numbers of nodes arriving at a barrier, it is outperformed by the 1:1 schedule.

6.9. Conclusion

Originally, the 1:A schedule has the same shortcomings for barriers as it has for broadcasts and multicasts. They can be overcome with an additional BCU, which is an extension of our hardware supported broadcast/multicast operation. Thereby, we install a dedicated BCU at each node. It handles all incoming flits where nodes tell that they arrived at the barrier managed by that node. When all nodes have arrived, the BCU automatically sends out a barrier release flit, which is a special kind of multicast flit. For controlling the BCU, we added four new assembly instructions: One for arriving at a barrier, a new status branch and two instructions for configuring which nodes participate in a barrier. Moreoever, barriers for subsets of nodes can be clearly distinguished via unique ids. The worst-case performance of our approach is very good, especially for large node numbers. Finally, the hardware supported barrier comes at low hardware costs, imposing an overhead of only 4-5% at the ALMs and 6-7% at the registers.

7

Case Studies: Impact on Communication in Benchmarks

We now investigate the impact of our hardware extensions. Thereby, we compare the WCET estimates for the communication of three benchmarks when they are running on the RC/MC processor with the A:A, 1:1 and 1:A schedules. For the latter, we have one variant with the original 1:A schedule, one with optimal algorithms running on it and one variant with our hardware extensions. However, no further evaluation of `ready` synchronization is carried out. We just take the hardware implementation to have a simple timing analysis and no problems with buffer overflows. Our benchmarks come from the NAS Parallel Benchmark Suite[1] [BBB+91b, BBB+91a]. They were originally designed with the goal to evaluate highly parallel systems if they are able to simulate an entire aerospace vehicle system [BBDS93]. This means their kernels may be utilized to compose applications running on cyber-physical systems implementing physical simulation. These are typical use cases for future embedded hard real-time many-core systems, e.g. when they are installed in cars, trains, aeroplanes etc. This chapter is divided into three sections, one for each benchmark: In Section 7.1, we will present our case study on the CG benchmark. Afterwards, the worst-case behavior of the MG benchmark running on our platform is investigated in Section 7.2. Finally, we evaluate the communication and computation of the LU benchmark in Section 7.3.

[1] Homepage: `http://www.nas.nasa.gov/Software/NPB/`

7.1. Case study: CG Benchmark

In [FSMU16], we already carried out a timing analysis for the *conjugate gradient (CG)* benchmark. It is taken from the NAS Parallel Benchmark Suite. Bailey et al. describe CG as following: *A conjugate gradient method used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long-distance communication, using unstructured matrix-vector multiplication* [BBB+91b]. For our analysis, we decided to analyze a class S CG benchmark, which is the smallest class: The matrix size is $1400 \cdot 1400$, there are 7 nonzero values per row. In the benchmark, the matrix is divided into equal sized blocks and each block is assigned to one node for computation. We employ 4x4=16 nodes – thus, the matrix is divided into 16 blocks of $350 \cdot 350$ values.

7.1.1. Timing Analysis of the CG Benchmark

For the evaluation, we reuse the timing analysis from [FSMU16]. Thereby, we created a formula which is composed of the sum of the WCET estimates of all sequential code parts and the communication parts. Now, we exchange the parts of this formula which represent `MPI_Bcast` with those from Chapter 5. This allows us to evaluate the impact of our hardware broadcast/multicast extension. The total WCET estimate of one iteration of the CG benchmark is assembled as $WCET^{CG}_{Iteration}$ in Formula 7.1 (based on [FSMU16]).

$$\begin{aligned} WCET^{CG}_{Iteration} &= 1\,896\,959 + 16 \cdot WCET^{SendRecv}(351) \\ &\quad + WCET^{Reduce}(2,15) + 17 \cdot WCET^{Reduce}(1,3) + 16 \cdot WCET^{Reduce}(351,3) \\ &\quad + WCET^{Bcast}(2,15) + 17 \cdot WCET^{Bcast}(1,3) + 16 \cdot WCET^{Bcast}(351,3) \end{aligned} \quad (7.1)$$

Thereby, we did not analyze the initialization of the benchmark, but the parts which are intended for benchmarking a system. Due to problems in the timing analysis, the sequential program parts were analyzed for integer variables instead of doubles, resulting in shorter computation times [FSMU16]. Moreover, the result $WCET^{CG}_{Iteration}$ in Formula 7.1 represents only one iteration. For a complete execution of the CG benchmark, it has to be multiplied with 15.

$WCET^{CG}_{Iteration}$ is composed of the following parts: First, 1 896 959 is the sum of the sequential code parts of the benchmark. Second, $WCET^{SendRecv}(f)$ represents the WCET estimate of `MPI_SendRecv`, which is an operation where nodes send and receive data at the same time (f flits are to be sent and another f flits to be received). Thereby, sender and receiver do not necessarily coincide. In the second line, we find several $WCET^{Reduce}$ estimates. They belong to the operation `MPI_Reduce`, which is a reduce operation (e.g. find global minimum or maximum or calculate a global sum).

Thereby, f flits are collected from χ nodes (see Formula 7.3). This means that $\chi + 1$ nodes work together at these operations (one root node and χ participant nodes). In the last line, we have $WCET^{Bcast}$ estimates, where we evaluate our hardware broadcast/multicast extension from Chapter 5.

At each iteration, parts of the matrix and one additional flit are exchanged between the nodes. Based on the WCET estimations [Brü19, FSMU16], we assembled Formula 7.2 for $WCET^{SendRecv}$. It covers the simplified case when the numbers of flits to be sent and received are equal and are processed in same periods. 250 cycles is the sum of the sequential code parts, $2 \cdot WCTT(1,1)$ represents exchange of `ready` and header flits and $\max(WCTT(f,1), f \cdot 34)$ the send/receive loop.

$$WCET^{SendRecv}(f) = 250 + 2 \cdot WCTT(1,1) + \max(WCTT(f,1), f \cdot 34) \tag{7.2}$$

For the estimate $WCET^{Reduce}$, we adapted the numbers from [FSMU16] in Formula 7.3. Thereby, S has to be set depending on the schedule, because this WCTT describes the receipt of flits from view of the root node, while we usually describe it from the sender's view. At the 1:A and A:A schedules, each period $n^2 - 1$ flits can be received. Thus, it takes $f - 1$ periods to receive flits from all nodes S. This is already expressed by $WCTT(f-1, ...)$. In this case, S can be set to 1. At the 1:1 schedule, each period at most 1 flit can be received. Therefore, it takes $\chi \cdot (f-1)$ periods to receive flits from all nodes. $(f-1)$ is already included in the WCTT term, but to express χ periods, we need to set S to χ.

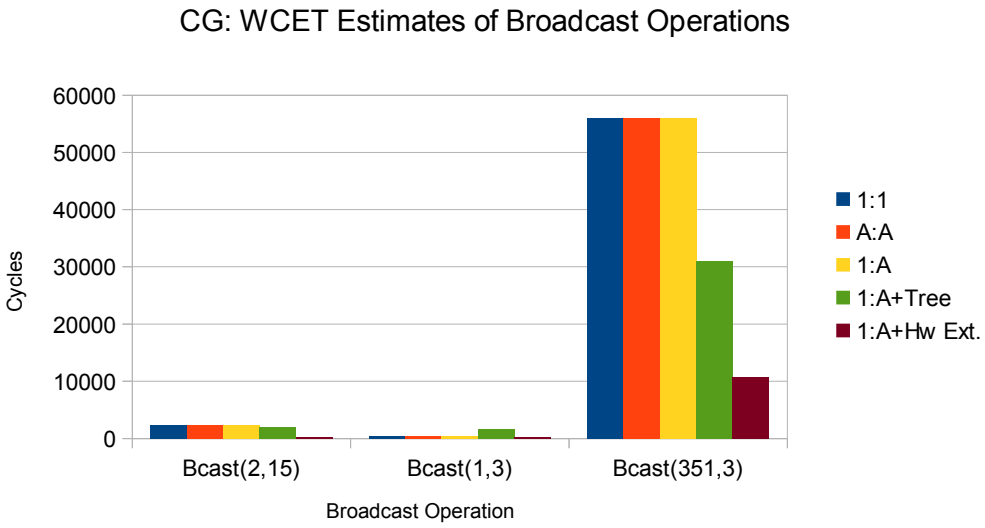
$$\begin{aligned} WCET^{Reduce}(f,\chi) =& 224 + 141 \cdot \chi + \max(23 + 17 \cdot \chi, 24 + 2 \cdot WCTT(1,1)) + 55 \cdot f \\ &+ 23 \cdot f \cdot \chi + \max(35 \cdot \chi \cdot (f-1), WCTT(f-1, S)) \end{aligned} \tag{7.3}$$

For our evaluation, we assume that `ready` flits have the same WCTT as data flits at the 1:1 schedule (usually, they have to be sent via a dedicated `ready` NoC with larger WCTTs).

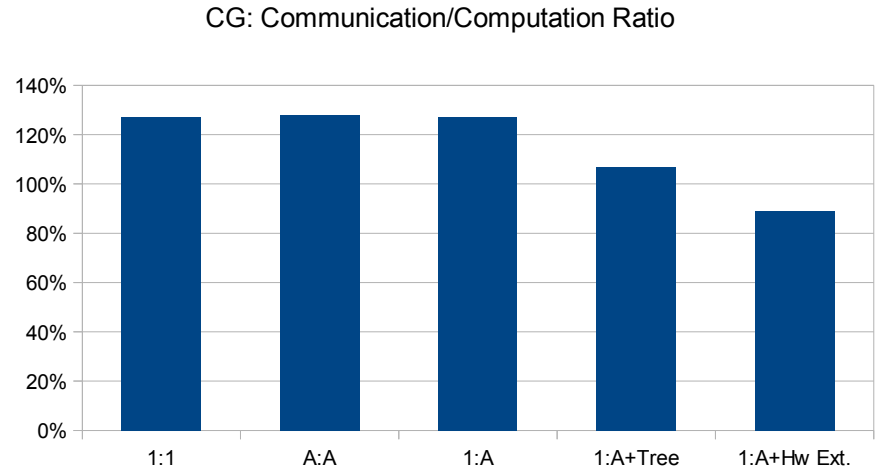
7.1.2. Impact of Hardware Extensions on Worst-Case Performance

As already mentioned in the last Subsection 7.1.1, the CG benchmark allows us only to evaluate our hardware broadcast/multicast extension, because it does not employ any barriers. Therefore, we compare the variants of `MPI_Bcast` we already presented in Section 5.8 in the context of the CG benchmark. Our results are displayed in Figure 7.1.

First, the WCET estimates of the broadcast/multicast operations are compared in Figure 7.1a. We see that the 1:A schedule with hardware broadcast/multicast extension always performs best. Notable is the 1:A schedule combined with a binary



(a) Total WCET estimate for all broadcast/multicast operations executed at CG.



(b) Ratio between worst-case communication times and WCET estimate of computations for CG.

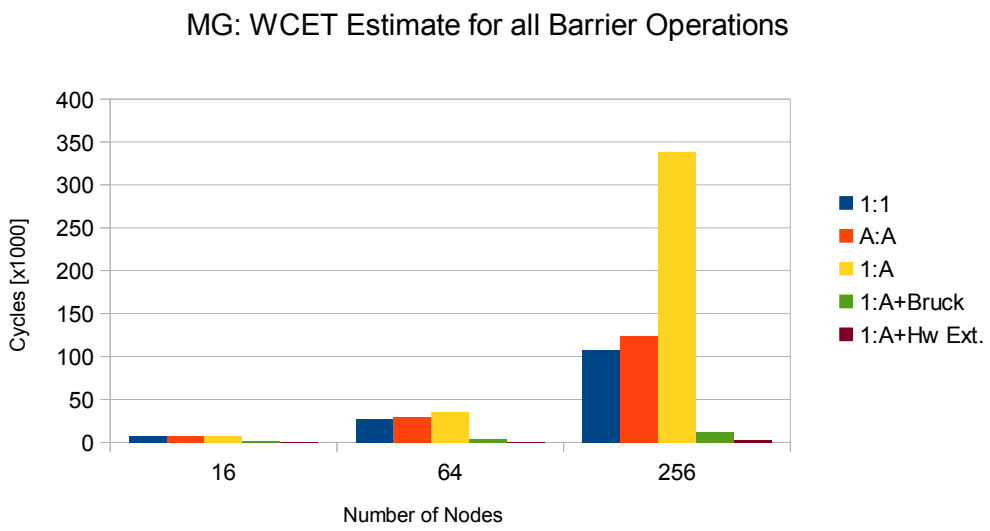
Figure 7.1.: Sum of all WCET estimates for broadcasts/multicasts during execution of CG benchmark and communication/computation ratio.

tree: It has the highest WCET estimate at `Bcast(1,3)` due to the high effort for preparing the binary tree. At `Bcast(351,3)`, the binary tree is found between 1:A with hardware broadcast/multicast extension and the other schedules. This means that the effort for creating the tree pays off at larger broadcasts/multicast. We did not include results for $WCET^{SendRecv}(351)$, because there we have very similar results for all schedules. Because our node count is very small, the variances between the schedules are also only very small. The same holds for $WCET^{Reduce}$.

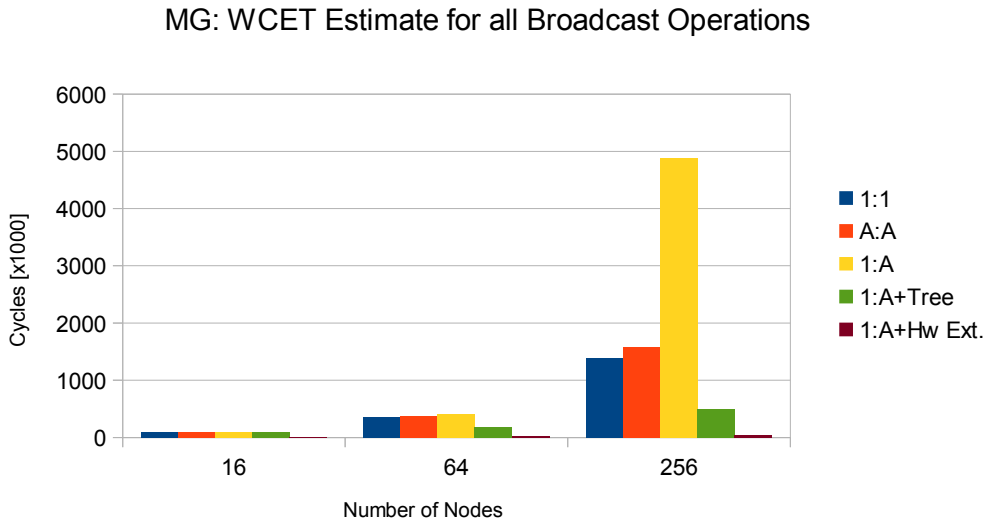
The total WCET estimate of the CG benchmark is roughly double of the sequential code part. Depending on the schedule and algorithm/hardware support, it is a little bit more or less. This can be seen at the communication/computation ratio in Figure 7.1b. Thereby, we divided the sum of the communication times through the WCET estimate of the sequential code parts (1 896 959). Only the 1:A schedule with hardware extension is below 100%, all other variants between 107% and 127%. At this benchmark, communication times are very large compared to the sequential program parts. This is caused by the communication intensive program structure, mainly influenced by the operations where 351 values are exchanged. These operations are executed in loops. We carried out a few experiments with larger matrices and node counts. The CG benchmark is highly dominated by these communication times. Thus, there is no (WCET) speedup on our platform and we investigated two more benchmarks.

7.2. Case study: MG benchmark

Our second benchmark *MultiGrid (MG)* also comes from the NAS Parallel Benchmark Suite [BBB+91b, BBB+91a]. It approximates the solution of a three-dimensional discrete Poisson equation using the V-cycle multigrid method [BBDS93]. This benchmark is very complex – it is the largest of the benchmarks from the NAS Parallel Benchmark Suite, with over 3 000 lines of code. The code contains lots of loops and case distinctions, making a characterization and timing analysis complex. However, a timing analysis was carried out by Brügmann [Brü19]. For communication, the operations `MPI_Bcast`, `MPI_Barrier`, `MPI_Sendrecv`, `MPI_Reduce` and pairs of `MPI_Send` and `MPI_Recv` are employed. Thereby, barriers and broadcasts always target all nodes. For `MPI_Reduce`, Brügmann took the timing analysis from Bürger [Bür19]. However, it seems to have a large overestimation, because Bürger's implementation is organized as tree, but performs worse than the simple implementation we utilized at the WCET analysis of the CG benchmark. Thus, we reused $WCET^{Reduce}$ (Formula 7.3) from the last Section 7.1. For the benchmark, we chose a grid size of $2048 \cdot 2048 \cdot 2048$ elements, which are computed over 4 iterations. The total WCET estimate for all barrier and broadcast operations is compared in Figure 7.2.



(a) Total WCET estimate for all barrier operations executed at MG.



(b) Total WCET estimate for all broadcast/multicast operations executed at MG.

Figure 7.2.: Sum of all WCET estimates for barriers and broadcasts/multicasts during execution of MG benchmark.

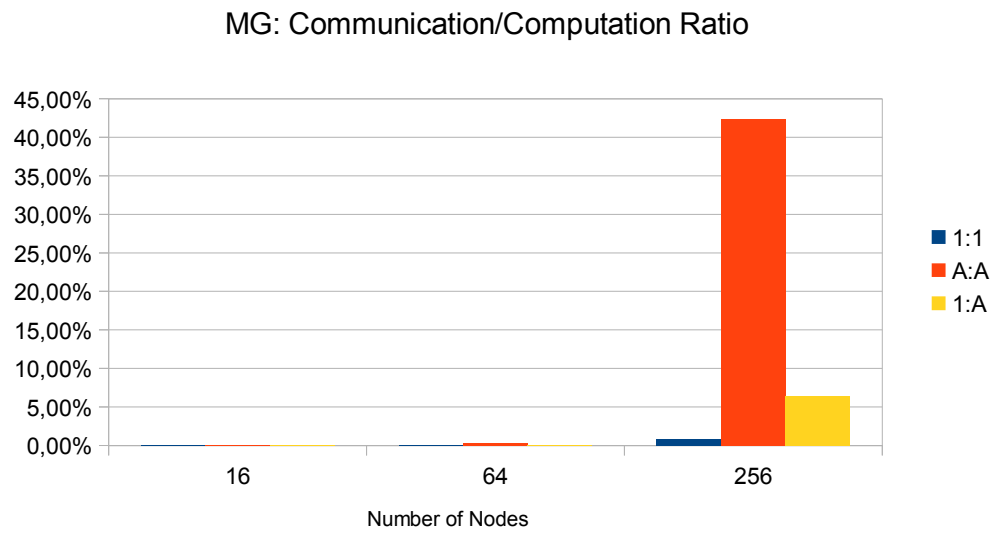


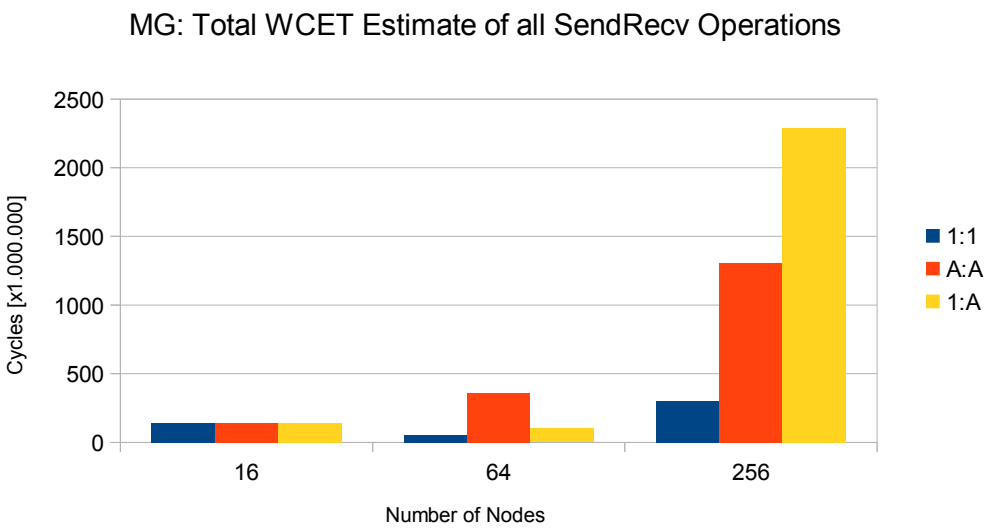
Figure 7.3.: Ratio between worst-case communication times and WCET estimate of computations for MG.

At a barrier with 16 nodes (Figure 7.2a), all variants with a simple implementation perform nearly same, maybe their WCET estimate is code-driven. The 1:A schedule with Bruck algorithm and with hardware barrier extension is much faster. When employing 64 nodes we see the tendency that the original 1:A schedule performs worst, followed by the A:A and 1:1 schedules. The 1:A schedule with Bruck algorithm is very fast, only outperformed by 1:A with hardware barrier extension. The results with 256 nodes are similar to 64 nodes, but the original 1:A schedule now takes more than double the time of the second-worst A:A schedule. In Figure 7.2b, the WCET estimates for all broadcast operations are summarized. At 16 nodes all except the 1:A schedule with hardware broadcast/multicast extension have nearly the same WCET estimate – here, the effort to build a binary tree does not seem to pay off for small node counts. When considering 64 and 256 nodes, the situation looks similar to barriers: The original 1:A schedule performs worst, followed by the A:A and 1:1 schedules. 1:A with hardware broadcast/multicast extension performs best, followed by 1:A with binary tree. However, at 64 nodes, the difference between between the original 1:A schedule and the A:A/1:1 schedules is very small. On the other hand, at 256 nodes the original 1:A schedule takes around three times of the A:A schedule.

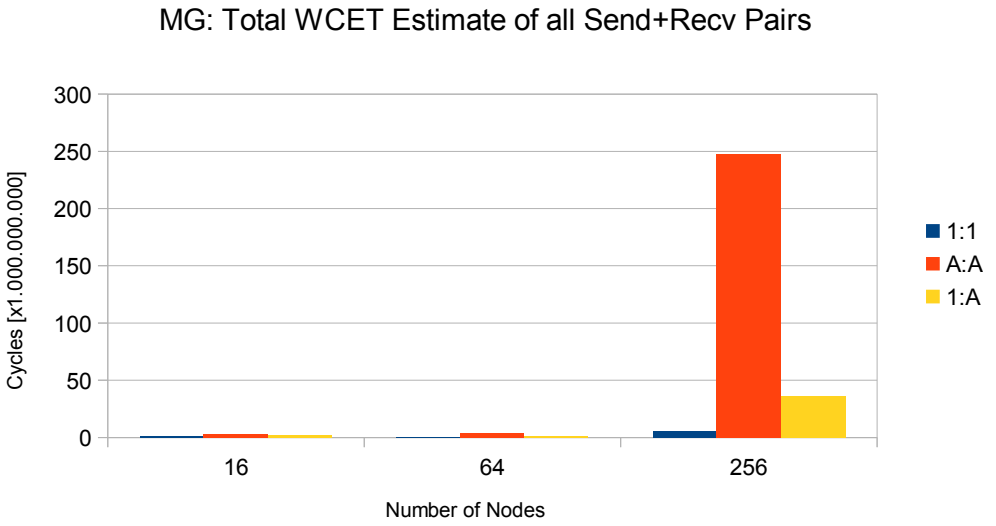
After comparing broadcasts/multicasts and barriers, we have a look on the communication/computation ratio in Figure 7.3. Thereby, we do not distinguish between different variants of the 1:A schedule, because they nearly have the same result here. At 16 nodes we see nothing in the figure, because of the small portion of

communication in the MG benchmark: Communication is 0.04% (1:1) to 0.06% (A:A) of the computation part (1:A is in-between with 0.05%). This already sets the trend for all larger node numbers. For 64 nodes, the range is between 0.04% 1:1 and 0.29% (A:A). When we employ 256 nodes, the result can be seen most clearly: A:A performs worst with 42.3%, followed by 1:A with 6.5% and 1:1 with 0.8%. Altogether, the 1:1 schedule performs best at the WCET estimation of the complete MG benchmark, followed by the 1:A schedule. We will now investigate the reasons.

The MG benchmark does not only consist of barriers and broadcasts – they are only a small part of the communication. Most of the communication is dominated by `MPI_SendRecv` and pairs of `MPI_Send` and `MPI_Recv`. At these operations, two (at the pair) or up to three (`MPI_Sendrecv`) nodes exchange data with each other. We calculated their total WCET estimates in Figure 7.4. Thereby, it is most important to note that the y-axis does not represent *cycles [x1.000]* like in Figure 7.2, but *cycles [x1.000.000]* in Figure 7.4a and even *cycles [x1.000.000.000]* in Figure 7.4b. This means their contribution is one thousand to one million times larger than that of barriers and broadcasts. At `MPI_Sendrecv` in Figure 7.4a, the (worst-case) performance of all schedules is similar for 16 nodes, because their WCTTs are quite similar for small node counts. However, at 64 nodes we already see the magnitudes of the different schedules: As already described in Subsection 3.2.2, periods of A:A grow with $O(n^3)$, 1:A with $O(n^2)$ and 1:1 with $O(n)$. Thus, A:A performs worst and is followed by 1:A – 1:1 is best. For 256 nodes, 1:A and A:A swap places. When a lot of flits are to be sent to different nodes, then 1:A may perform worse than A:A for large numbers of nodes. This is because A:A can send flits to different nodes within the same period, while 1:A always has to wait for the next period. At the pairs of `MPI_Send` and `MPI_Recv` in Figure 7.4b, the situation is different. For 16 and 64 nodes, they behave like the `MPI_Sendrecv` operation. When employing 256 nodes, the A:A schedule again performs worst. At these pairs each node sends or receives data to/from exactly one other node. This means the shorter the periods of a schedule, the better its (worst-case) performance. Again, we have the strong impact from the growing periods from Subsection 3.2.2. Moreover, the pairs of `MPI_Send` and `MPI_Recv` have the highest dominance at the total WCET estimate of MG, because of their high cycle contribution *[x1.000.000.000]*.



(a) Total WCET estimate for all `MPI_SendRecv` operations executed at MG.



(b) Total WCET estimate for all pairs of `MPI_Send` and `MPI_Recv` executed at MG.

Figure 7.4.: Sum of all WCET estimates for direct node-to-node communication during execution of MG benchmark.

7.3. Case Study: LU Benchmark

The *Lower-Upper symmetric Gauß-Seidel (LU)* benchmark is part of the NAS Parallel Benchmark Suite [BBB+91b, BBB+91a] as well as the SPLASH-2 benchmark suite [WOT+95]. Woo et al. describe it as following: *The LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix* [WOT+95]. Thereby, the matrix is divided into equal sized blocks to be distributed on several nodes.

7.3.1. Workflow of the LU Benchmark

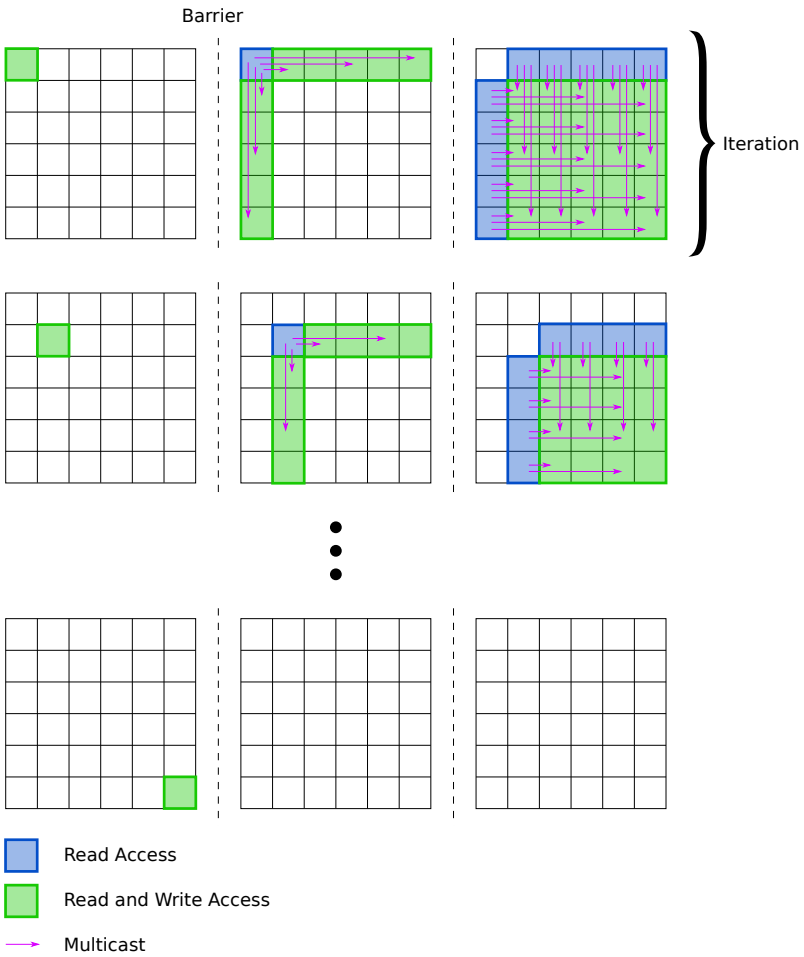


Figure 7.5.: Workflow of the LU benchmark, figure based on [Ste19]. Each row represents one iteration. The iterations are divided into three steps. After the first and second step, all nodes wait at a barrier.

In Figure 7.5, we visualize the workflow of the LU benchmark. It expects a

quadratic matrix[2] with m rows and columns ($m \cdot m$ floating point numbers) as input, e.g. when $m = 120$, 14 400 floating point numbers are stored in the matrix. For distributed computation, the matrix is divided into equal sized blocks of $e \cdot e$ elements, e.g. when we set $e = 20$, each block comprises 400 floating point numbers. Thereby, we have b blocks in each row and each column, e.g. for $m = 120$ and $e = 20$, there are $b = 6$ blocks in each row and column and a total of 36 blocks. Because the number of blocks is usually larger than the number of nodes, each node manages several blocks. They are distributed in a block-cyclic way [MSM04], see an example for $b = 6$ and $n = 2$ in Table 7.1, where we have $n \cdot n = 4$ nodes in total.

Table 7.1.: 6x6 block-cyclic distribution to 4 nodes. The number in a cell specifies its assignment to a node.

1	2	1	2	1	2
3	4	3	4	3	4
1	2	1	2	1	2
3	4	3	4	3	4
1	2	1	2	1	2
3	4	3	4	3	4

We see that there are always $n = 2$ different nodes in the same row or column. This is important for the multicasts during the LU benchmark, because they take place row-wise or column-wise.

The LU benchmark is executed for b iterations. Each iteration is represented by a row in Figure 7.5 and consists of three steps, which are represented by the columns:

1. One single block is computed by one single node, while the other nodes wait at a barrier.
2. A multicast to the blocks in the same row and same column is sent out. Then, computations on these blocks take place. Afterwards, the step is finished with a second global barrier.
3. Multiple multicasts are sent out: One for each row and one for each column. Finally, computations on all blocks where data was received take place.

At steps 2 and 3 nodes do not need to send data to themselves, because they already have it. Then, data only has to be multicasted to the other nodes. The number of considered blocks decreases at each iteration. Thus, we introduce p for the number of participating blocks in the same row at step 2 (green blocks in the same row as the blue block). At the example in Figure 7.5, $p = 5$ in the first iteration and $p = 4$ in the second iteration. As can be seen in the last row of Figure 7.5, the last iteration walks

[2]The benchmark allows matrices to be non-quadratic. However, we focus only on quadratic matrices to keep our estimations manageable.

through all steps, but only at the first step something happens. This is because there are no receivers at the multicasts – thus, they are not executed anymore. Moreover, there are no computations at steps 2 and 3. Therefore, $p = 0$ in the last iteration. However, the nodes still meet at the barriers.

7.3.2. Characterization of the (Worst-Case) Execution Behaviour

We did not carry out a complete timing analysis of the LU benchmark, but of its computation parts for the Gauß–Seidel method. Therefore, we now characterize which computation parts are executed how often and how much communication takes place in between.

At step 1, only one node carries out a computation called `lu0` in our implementation. All other nodes wait at a barrier for this node to finish its computation. The WCET of step 1 is estimated in Formula 7.4

$$WCET^{LU}_{Step1} = WCET^{lu0} + WCET^{Barrier} \tag{7.4}$$

Step 2 starts with a multicast to all participating blocks p in the same row and another p in the same column as the block computed in step 1. Because the node who owns this block does not need to send it to itself, $n-1$ nodes are receivers in the same row and another $n-1$ nodes in the same column. When the row or column contains less than $n-1$ participating blocks p, then there are only p nodes receiving the multicast. After the blocks were received, each node has to compute at most $\lceil \frac{p}{n} \rceil$ blocks in the same row (`bdiv` in our implementation) and another $\lceil \frac{p}{n} \rceil$ blocks in the same column (`bmodd` in our implementation). Finally, the step finishes with a barrier. Its complete WCET estimation is summarized in Formula 7.5.

$$\begin{aligned} WCET^{LU}_{Step2} =& WCET^{Bcast}_{2 \cdot min(p,n-1)} \\ &+ \left\lceil \frac{p}{n} \right\rceil \cdot WCET^{bdiv} + \left\lceil \frac{p}{n} \right\rceil \cdot WCET^{bmodd} \\ &+ WCET^{Barrier} \end{aligned} \tag{7.5}$$

At the last step 3, all blocks computed in step 2 are to be multicasted to their row or column, respectively. This means p multicasts to $n-1$ nodes to reach all rows and the same to reach all columns. When the participating blocks p are less than $n-1$, then all multicasts are sent to p receivers. Like at step 2, all blocks where data was exchanged, are to be re-computed (`bmod` in our implementation). Because there are $p \cdot p$ blocks distributed to $n \cdot n$ nodes, at most $\lceil \frac{p \cdot p}{n \cdot n} \rceil$ blocks are to be computed by the same node (in the worst-case). The WCET of step 3 is estimated in Formula 7.6.

$$WCET^{LU}_{Step3} = (p+p) \cdot WCET^{Bcast}_{min(p,n-1)} + \left\lceil \frac{p \cdot p}{n \cdot n} \right\rceil \cdot WCET^{bmod} \tag{7.6}$$

Because the number of participating blocks p changes at each iteration, the total WCET estimate has to be formulated as sum: In the first iteration $p = b - 1$, in the second iteration $p = b - 2$ and so on. In the last iteration $p = 0$. Altogether, we get Formula 7.7 as result for the total WCET estimate of the LU benchmark.

$$WCET^{LU}_{Total} = \sum_{p=0}^{b-1} (WCET^{LU}_{Step1} + WCET^{LU}_{Step2} + WCET^{LU}_{Step3}) \tag{7.7}$$

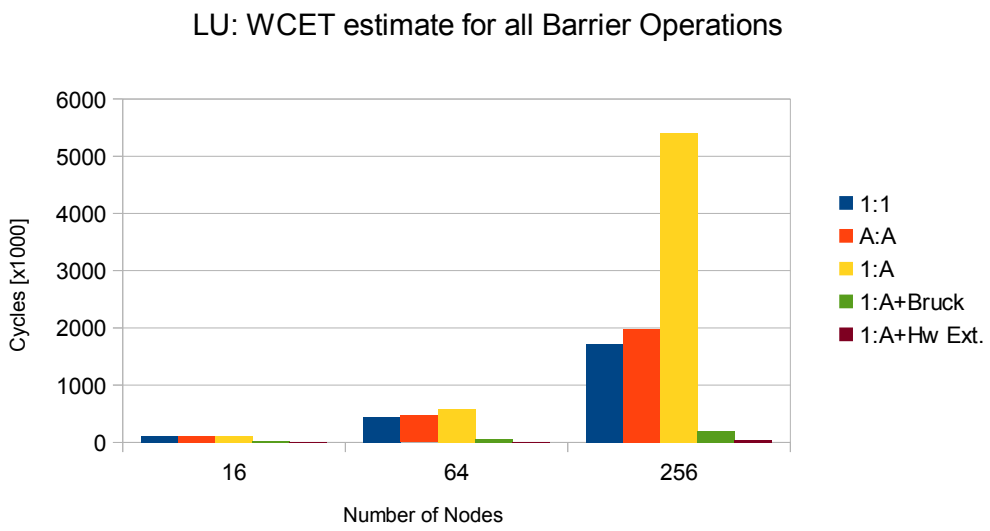
Moreover, we add 5% of the WCET estimates of communication times for the overhead to prepare communication and other situations not covered by our characterization.

7.3.3. WCET estimates for LU

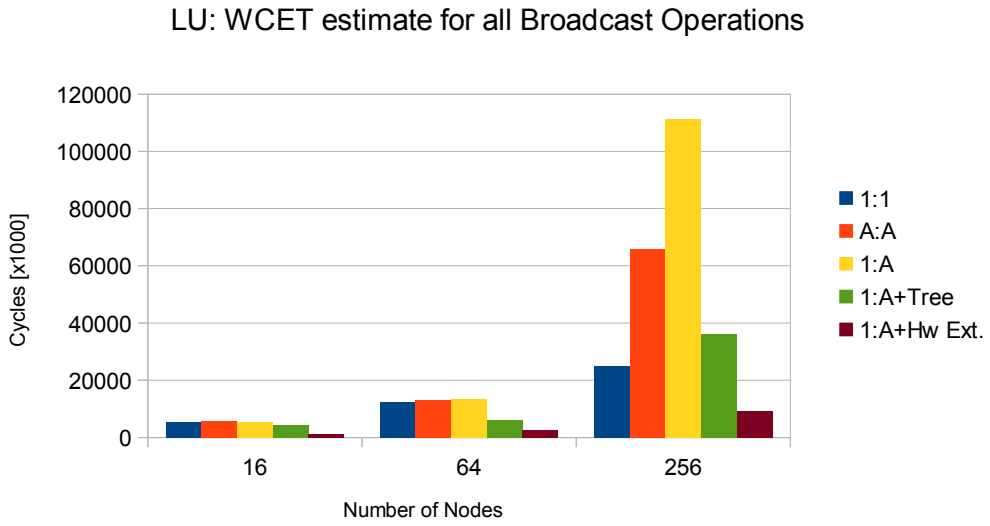
To illustrate the (WCET) speedup potential of the RC/MC and the impact of our hardware extensions, we take a $1024 \cdot 1024$ matrix ($m = 1024$), divided into $32 \cdot 32$ blocks, where each block has $32 \cdot 32$ values ($e = 32$). Then, we estimate WCETs for 1 node, 4x4=16 nodes, 8x8=64 nodes and 16x16=256 nodes. At the sequential execution, we assume that the matrix is divided into equal-sized blocks as shown in Figure 7.5, but all blocks are computed by one single node and no communication takes place. The total WCET estimate for this sequential execution is 16.7 billion cycles. At the parallel versions, we calculate $WCET^{LU}_{Total}$ as assembled in Formula 7.7 and add 5% of the WCET estimates of communication as described in the last Subsection 7.3.2. Like at the evaluation of the CG and MG benchmarks, we assume that `ready` flits have the same WCTT as data flits at the 1:1 schedule.

Figure 7.6 displays the sum of all WCET estimates for broadcasts/multicasts and barriers. At the WCET estimates of barriers in Figure 7.6a, we see similar numbers for small node counts at all schedules. This is because of similar WCTTs at small node counts. The 1:A schedule with Bruck algorithm and hardware barrier extension exhibits very low WCET estimates. On the other hand, the 1:A schedule without Bruck and hardware extensions shows very high WCET estimates when the number of nodes increases. At the broadcast/multicast WCET estimates in Figure 7.6b, we see a situation similar to barriers: The WCET estimates are very similar for small node numbers. However, the 1:A schedule becomes very large for increasing node numbers, but performs quite good when employing a binary tree – and even better with hardware broadcast/multicast extension. Altogether, the 1:A schedule is very scalable with hardware extensions.

In Figure 7.7a, we illustrate WCET speedups of the LU benchmark. They depend on the schedule: We have a WCET speedup between 54 for 1:A without broadcast/-multicast and barrier extensions and 87 with hardware extensions. Differences at the speedup can only be seen at the large node count 16x16=256 nodes, because for small node numbers, communication operations only have a small impact. This

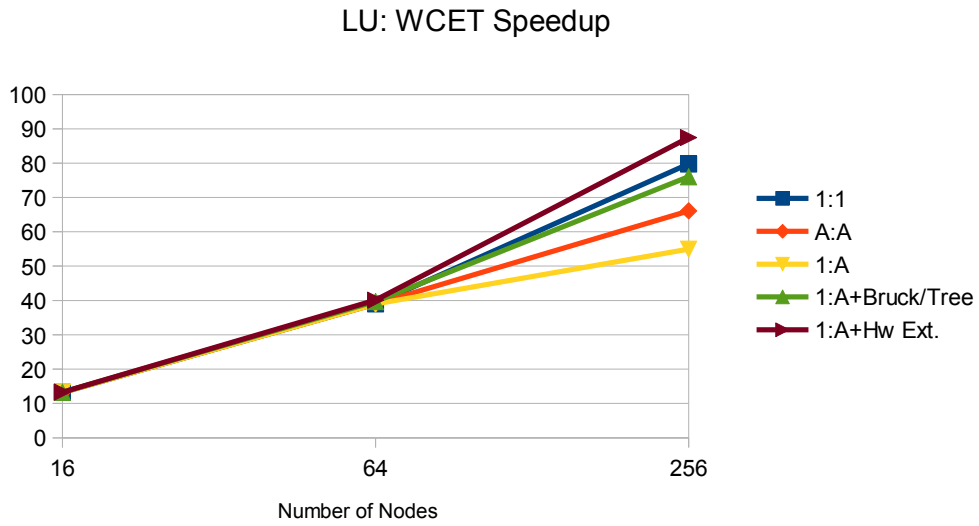


(a) Total WCET estimate for all barrier operations executed at LU.

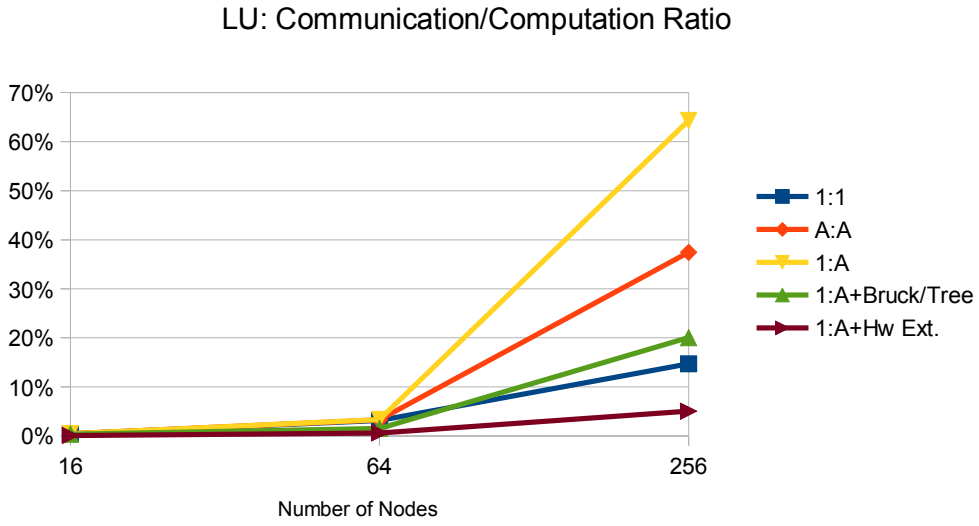


(b) Total WCET estimate for all broadcast/multicast operations executed at LU.

Figure 7.6.: Sum of all WCET estimates for barriers and broadcasts/multicasts during execution of LU benchmark.



(a) WCET speedup of the LU benchmark.



(b) Ratio between worst-case communication times and WCET estimate of computations for LU.

Figure 7.7.: WCET speedup and communication/computation ratio for LU.

is also seen at the communication/computation ratio in Figure 7.7b. For 16 nodes, communication is less than 1% of the computation time. At 64 nodes, it stays below 1% with hardware extensions, while the other cases have low percentages. Finally, at 256 nodes, the different scenarios can clearly be distinguished: The 1:A schedule with hardware extensions has a communication/computation ratio of 5%, while it is 65% without hardware extensions. Another factor at the communication/-computation ratio is the increasing communication demand when more nodes are employed – more nodes need to communicate more with each other. At the same time, the computation time is decreased when the application can be distributed better. Altogether, the scalability can be seen from the speed how fast the communication/computation ratio increases – the slower, the better. Moreover, we see that although the communication portion of the benchmark is not too large it has an important impact on the WCET speedup. However, the LU benchmark *exhibits a somewhat limited amount of parallelism* [BBB+91b]. This is demonstrated best at step 1, where all nodes have to wait for one node and have to be idle meanwhile.

8

Conclusion and Outlook

In our thesis, we extended the RC/MC processor with three hardware extensions at the network interface of its nodes to improve timing predictability and worst-case performance. The RC/MC is a many-core processor, where each node consists of a simple processing element, a local scratchpad memory and a network interface. All nodes are connected via a predictable NoC. Our platform is intended to execute hard real-time applications similar to the Bulk Synchronous Parallel (BSP) model. In software, this is realized via the Message Passing Interface (MPI). A timing analysis for the RC/MC can be carried out by combining a WCET analysis of the sequential code executed on the nodes with a Worst-Case Transportation Time (WCTT) analysis of the NoC.

To increase the timing predictability of the network traffic arriving at a node and to avoid buffer overflows, we introduced `ready` synchronization: Before two nodes can communicate with each other, the receiving node sends a `ready` flit to the sender node to indicate that it is ready to receive data flits. The sender node waits for the `ready` flit before it starts sending. This way it is ensured that sender and receiver are synchronized and the receiver is able to handle all incoming flits. We extended all nodes with hardware logic to distinguish `ready` and data flits and sort out `ready` flits before they reach the processing element. Instead, the sender of a `ready` flit is marked as ready in a hardware bit array. With our hardware extension, the processing element is disencumbered from handling `ready` flits and can focus on handling desired data flits. Before data is sent to some other node, its status can be retrieved from the hardware bit array via a new assembly instruction. Our evaluation shows that this simplifies timing analysis and increases (worst-case)

performance. Moreover, the hardware costs for `ready` synchronization are low – it costs less than one receive buffer slot and may save many of them.

Our next two hardware extensions focus on improving the (worst-case) performance of the One-to-All (1:A) time-division multiplexing (TDM) schedule. This schedule defines when each node is allowed to send a flit and thereby avoids collisions and conflicts on the NoC. However, its (worst-case) performance is low when a flit is to be sent to several nodes. Thus, we exploited its reserved time slots to send flits to all nodes at once. This is possible by copying a flit while it travels through the NoC. As a result, our first extension for the 1:A schedule is a hardware-supported broadcast operation. By adapting the receive logic of nodes, it is also capable to send out multicasts to a subset of all nodes. Thereby, nodes store the number of the node whom they sent a `ready` flit to and sort out all broadcast/multicast flits from other senders. When a broadcast/multicast flit from the desired sender passes by, it is copied into the receive buffer. Additionally, it is counted how many broadcast/multicast flits are received to stop the receive operation when all desired flits have arrived. We carried out a timing analysis of our new hardware-supported broadcast/multicast operation and compared it with two other implementations. Our hardware broadcast/multicast operation always performs best in both theoretical and practical evaluations. It comes at 5–10% hardware overhead, which might be reduced by omitting an optional part of the extension which only brings minor performance improvements.

A hardware-supported barrier operation is our second hardware extension for the 1:A schedule. Basically, we install a new Barrier Control Unit (BCU) at all nodes, which handles all barrier-related flits. This means when a node wants to participate in a barrier, it sends a *barrier arrival* flit to the coordinating node. There, the barrier arrival flit does not reach the processing element, but is handled by the BCU. The BCU is configured by the node where it is installed. When all nodes have arrived (including the node where the BCU is installed), the BCU sends out a *barrier release* broadcast flit indicating that all participating nodes can continue their program execution. For distinguishing several barriers taking place at the same time, all barriers have a *unique id*. Altogether, our platform can handle as many barriers at the same time as there are nodes on the chip. We evaluated our hardware-supported barrier operation in a similar manner to our evaluation of the hardware-supported broadcast/multicast operation. Thereby, our hardware barrier extension always performs best when more than two nodes participate. It imposes only low hardware costs of around 5% overhead. Most of it is already paid with the hardware broadcast/multicast extension. All of our hardware extensions are realized as VHDL prototype and synthesized for an FPGA to evaluate their hardware costs. For functional tests, we also integrated them in our many-core simulator (MacSim). Our hardware extensions are all designed in a way to be

scalable when the number of nodes increases.

For our case studies which are taken from the NAS Parallel Benchmark (NPB) Suite, the scalability was also one important factor to be investigated. These benchmarks were chosen by NASA and represent scenarios for physical simulation, which are relevant for applications like autonomous driving and other use cases for future embedded hard real-time many-core processors. Therefore, we carried out a timing analysis of three benchmarks and then evaluated the impact of our hardware extensions. The first benchmark Conjugate Gradient (CG) implements *a conjugate gradient method used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix* [BBB+91b]. We only evaluated CG for 4x4=16 nodes, because it does not scale well due to a lot of communication. But at this small scenario, our hardware broadcast/multicast extension performed very well, being the only variant to bring the communication/computation ratio below 100%. Our next benchmark MultiGrid (MG) approximates the solution of a three-dimensional discrete Poisson equation [BBDS93]. It is the largest benchmark of the NPB Suite and utilizes five different communication operations. While our hardware broadcast/multicast and barrier extensions improve the results of the benchmark, two non-accelerated communication operations have a considerably stronger impact on the total WCET estimate of the benchmark. Unfortunately, they perform second-worst with the 1:A schedule and therefore decrease the total worst-case performance of the MG benchmark. Thus, the 1:1 schedule performs better at the MG benchmark. Finally, we investigated a third benchmark: Lower-Upper symmetric Gauß-Seidel (LU). It *factors a dense matrix into the product of a lower triangular and an upper triangular matrix* [WOT+95]. Thereby, it only uses multicasts and barriers for communication. We carried out a WCET analysis of its computation kernels and analyzed its communication. This allowed us to give rough estimates for the total WCET and calculate a WCET speedup. Our hardware broadcast/multicast and barrier extensions always perform best. In addition, LU revealed that for a high number of nodes our hardware extensions have a high positive impact on the (WCET) speedup. Although the `ready` hardware extension was not explicitly evaluated with the NPB benchmarks, it was an important part of the timing analyses of the communication operations – there, it simplified timing analysis a lot. Our evaluation showed that the 1:A schedule with hardware extensions is the best solution for some cases, but not always – this depends on the characteristics of the benchmark. But when communication in a benchmark is well suited for the 1:A schedule, our hardware extensions have a positive impact. Then, even the best algorithms are outperformed by our hardware extensions. Moreover, our hardware extensions impose a high scalability.

As future work, hardware support for more complex operations might be introduced, for example operations that collect or distribute data from/to other nodes (`MPI_Gather`, `MPI_Scatter`). Thereby, flits might be collected in-order or out-of-

order. Since the 1:A schedule was outperformed at a situation where a lot of flits were sent to the same node, this might be improved, e.g. by integrating Direct Memory Access (DMA), like it is already installed at the T-CREST architecture [SAA+15]. Another solution might be to integrate support for neighbourhood communication. Walter [Wal19] extended the `ready` NoC of the 1:1 schedule to a status NoC, where each node broadcasts its status every period. This allows to send a `not-ready` flit to tell all nodes that the receive buffer of a node is running full. Our evaluation also revealed that the various schedules A:A, 1:1 and 1:A have different properties. It should be investigated which algorithms are the best solution for each of them.

Bibliography

[ABB+07] Anant Agarwal, Liewei Bao, John Brown, Bruce Edwards, Matt Mattina, Chyi-Chang Miao, Carl Ramey, and David Wentzlaff. Tile Processor: Embedded Multicore for Networking and Multimedia. In *2007 IEEE Hot Chips 19 Symposium (HCS)*, pages 1–12. IEEE, August 2007.

[ABC+05] Narasimha R. Adiga, Matthias A. Blumrich, Dong Chen, Paul Coteus, Alan Gara, Mark E. Giampapa, Philip Heidelberger, Sarabjeet Singh, Burkhard D. Steinmacher-Burow, Todd Takken, et al. Blue Gene/L torus interconnection network. *IBM Journal of Research and Development*, 49(2/3):265–276, 2005.

[ADG16] Roman Atachiants, Gavin Doherty, and David Gregg. Parallel performance problems on shared-memory multicore systems: Taxonomy and observation. *IEEE Transactions on Software Engineering*, 42(8):764–785, Aug 2016.

[AESF16] Hamdi Ayed, Jérôme Ermont, Jean-luc Scharbarg, and Christian Fraboul. Towards a unified approach for worst-case analysis of Tilera-like and KalRay-like NoC architectures. In *IEEE World Conference on Factory Communication Systems (WFCS)*, pages 1–4, May 2016.

[AFA10] José L. Abellán, Juan Fernández, and Manuel E. Acacio. Efficient and scalable barrier synchronization for many-core cmps. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, pages 73–74, New York, NY, USA, 2010. ACM.

[AHA+05] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili Zheng. Optimization of MPI Collective Communication on BlueGene/L Systems. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 253–262, New York, NY, USA, 2005. ACM.

[AIS09] Ankur Agarwal, Cyril Iskander, and Ravi Shankar. Survey of Network on Chip (NoC) Architectures & Contributions. *Journal of Engineering, Computing and Architecture*, 3(1):21–27, 2009.

[Alt06] Altera/Intel. FPGA Architecture White Paper. Whitepaper, August 2006. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01003.pdf`.

[Aue18] Dominik Johannes Ruslan Auer. Implementierung und Evaluierung von Hardware-Broadcast und Hardware-Barrieren im One-to-All-Schedule des RC/MC-Prozessors. Master's thesis, University of Augsburg, September 2018.

[AUT14] AUTOSAR. Guide to multi-core systems. Technical report, AUTOSAR, Frankfurter Ring 224, 80807 Munich, Germany, March 2014. `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-1/AUTOSAR_EXP_MultiCoreGuide.pdf`.

[Bau18] Maximilian Bauer. Modellierung des Energieverbrauchs der RC/MC-Architektur. Master's thesis, University of Augsburg, June 2018.

[BBB+91a] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks – Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[BBB+91b] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[BBB+10] Armelle Bonenfant, Ian Broster, Clément Ballabriga, Guillem Bernat, Hugues Cassé, Michael Houston, Nicholas Merriam, Marianne de Michiel, Christine Rochange, and Pascal Sainrat. Coding guidelines for WCET analysis using measurement-based and static analysis techniques. Technical Report IRIT/RR-2010-8-FR, IRIT-Institut de recherche en informatique de Toulouse, March 2010.

[BBDS93] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. Nas parallel benchmark results. *IEEE Parallel Distributed Technology: Systems Applications*, 1(1):43–51, February 1993.

[BCRS11] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, volume

6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer Berlin Heidelberg, 2011.

[BdMS08] Armelle Bonenfant, Marianne de Michiel, and Pascal Sainrat. oRange: A tool for static loop bound analysis. In *Workshop on Resource Analysis, University of Hertfordshire, Hatfield, UK*, volume 9, September 2008.

[BEA+08] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, David Wentzlaff, Walker Anderson, Ethan Berger, Nat Fairbanks, Durlov Khan, Froilan Montenegro, Jay Stickney, and John Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, page 88, February 2008.

[BHK+97] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, November 1997.

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

[Bor10] Shekhar Borkar. Future of interconnect fabric: A contrarian view. In *Workshop on System Level Interconnect Prediction*, SLIP '10, pages 1–2, 2010.

[BP90] Carl J. Beckmann and Constantine D. Polychronopoulos. Fast barrier synchronization hardware. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 180–189, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[Brü19] Jakob Brügmann. WCET-Analyse des parallelen Multi-Grid Benchmarks für den RC/MC-Prozessor. Bachelor's Thesis, University of Augsburg, April 2019.

[BU19] Martin Bitterlich and Tilmann Unte. Implementierung von Hardware-Broadcasts/Multicasts und Hardware-Barrieren im RC/MC-Prozessor. Project Module, University of Augsburg, May 2019.

[Bür19] Martin Bürger. WCET-Analyse eines parallelen Programms zur Simulation von Ozean-Strömungen auf dem RC/MC-Prozessor. Bachelor's Thesis, University of Augsburg, May 2019.

[CM07] Jason Cong and Kirill Minkovich. Optimality study of logic synthesis for LUT-based FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):230–239, February 2007.

[CMR+06] Martijn Coenen, Srinivasan Murali, Andrei Ruadulescu, Kees Goossens, and Giovanni De Micheli. A buffer-sizing algorithm for networks on chip using TDMA and credit-based end-to-end flow control. In *Hardware/Software Codesign and System Synthesis. CODES+ISSS'06. Proceedings of the 4th International Conference*, pages 130–135. IEEE, 2006.

[dD15] Benoît Dupont de Dinechin. Kalray MPPA®: Massively Parallel Processor Array – Revisiting DSP Acceleration with the Kalray MPPA Manycore Processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–27. IEEE, August 2015.

[dD16] Benoît Dupont de Dinechin. Kalray MPPA®: Massively Parallel Processor Array – Engineering a Manycore Processor Platform for Mission-Critical Applications. Keynote at the *IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-16)*, September 2016. `http://mcsoc-forum.org/2016/wp-content/uploads/2015/07/Keynote-De-Dinechin.pdf`.

[dDAB+13] Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoît Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, and Thierry Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, September 2013.

[DT04] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[FJO+16] Martin Frieb, Ralf Jahr, Haluk Ozaktas, Andreas Hugl, Hans Regler, and Theo Ungerer. A parallelization approach for hard real-time systems and its application on two industrial programs. *International Journal of Parallel Programming*, 44(6):1296–1336, 2016.

[FSMU16] Martin Frieb, Alexander Stegmeier, Jörg Mische, and Theo Ungerer. Employing MPI Collectives for Timing Analysis on Embedded Multi-Cores. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess*

Series in Informatics (OASIcs), pages 10:1–10:11, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[FSMU18] Martin Frieb, Alexander Stegmeier, Jörg Mische, and Theo Ungerer. Lightweight hardware synchronization for avoiding buffer overflows in network-on-chips. In Mladen Berekovic, Rainer Buchty, Heiko Hamann, Dirk Koch, and Thilo Pionteck, editors, *31st International Conference on Architecture of Computing Systems (ARCS)*, pages 112–126. Springer, Springer International Publishing, 2018.

[GCH11] Gernot Gebhard, Christoph Cullmann, and Reinhold Heckmann. Software Structure and WCET Predictability. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18, pages 1–10, Dagstuhl, Germany, 2011.

[GDR05] Kees Goossens, John Dielissen, and Andrei Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5):414–421, 2005.

[GH04] James R. Goodman and Herbert H. J. Hum. MESIF: A two-hop cache coherency protocol for point-to-point interconnects. Technical report, University of Auckland, 2004. `http://hdl.handle.net/2292/11593`.

[GKN+17] Kees Goossens, Martijn Koedam, Andrew Nelson, Shubhendu Sinha, Sven Goossens, Yonghui Li, Gabriela Breaban, Reinier van Kampenhout, Rasool Tavakoli, Juan Valencia, Hadi Ahmadi Balef, Benny Akesson, Sander Stuijk, Marc Geilen, Dip Goswami, and Majid Nabi. *NoC-Based Multiprocessor Architecture for Mixed-Time-Criticality Applications*, pages 1–40. Handbook of Hardware/Software Codesign. Springer Netherlands, Dordrecht, 2017.

[Gor18] Roman Marek Alexander Kermit Gorlo. Optimierung der Netzwerkschnittstelle des PaterNoster-NoC. Master's thesis, University of Augsburg, May 2018.

[GSLD11] Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David. ΣC: A Programming Model and Language for Embedded Manycores. In Yang Xiang, Alfredo Cuzzocrea, Michael Hobbs, and Wanlei Zhou, editors, *Algorithms and Architectures for Parallel Processing*, pages 385–394, Berlin, Heidelberg, October 2011. Springer Berlin Heidelberg.

[GvMPW02] Kees Goossens, Jef van Meerbergen, Ad Peeters, and Paul Wielage. Networks on silicon: combining best-effort and guaranteed services. In *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 423–425, March 2002.

[Hem94] Rolf Hempel. The MPI standard for message passing. In Wolfgang Gentzsch and Uwe Harms, editors, *High-Performance Computing and Networking*, pages 247–252, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[HGBH09] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):2:1–24, January 2009.

[HJK+00] Ahmed Hemani, Axel Jantsch, Shashi Kumar, Adam Postula, Johnny Öberg, Mikael Millberg, and Dan Lindqvist. Network on a Chip: An architecture for billion transistor era. In *Proceedings of the IEEE NORCHIP Conference*. IEEE, November 2000.

[IG18] IEEE and The Open Group. IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008). The Open Group Base Specifications Issue 7, 2018.

[Int19a] Intel. Quartus Prime Design Software. Whitepaper, April 2019. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/br/br-quartus-prime-software.pdf`.

[Int19b] Intel. Stratix V Device Handbook Volume 1: Device Interfaces and Integration. Whitepaper, April 2019. `https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/stratix-v/stx5_core.pdf`.

[Int19c] Intel. Stratix V Device Overview. Whitepaper, April 2019. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-v/stx5_51001.pdf`.

[JGU+14] Ralf Jahr, Mike Gerdes, Theo Ungerer, Haluk Ozaktas, Christine Rochange, and Pavel G. Zaykov. Effects of structured parallelism by parallel design patterns on embedded hard real-time systems. In *IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, Aug 2014.

[KJS+02] Shashi Kumar, Axel Jantsch, Juha-Pekka Soininen, Martti Forsell, Mikael Millberg, Johny Öberg, Kari Tiensyrjä, and Ahmed Hemani. A network on chip architecture and design methodology. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002*, pages 117–124. IEEE, April 2002.

[KKC+08] Tushar Krishna, Amit Kumar, Patrick Chiang, Mattan Erez, and Li-Shiuan Peh. Noc with near-ideal express virtual channels using global-line communication. In *16th IEEE Symposium on High Performance Interconnects, 2008. HOTI'08*, pages 11–20. IEEE, August 2008.

[KM95] H. T. Kung and Robert Morris. Credit-based flow control for ATM networks. *IEEE Network*, 9(2):40–48, 1995.

[KPKJ07] Amit Kumar, Li-Shiuan Peh, Partha Kundu, and Niraj K. Jha. Express Virtual Channels: Towards the ideal interconnection fabric. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 150–161, New York, NY, USA, 2007. ACM.

[KQnBS15] Sebastian Kehr, Eduardo Quiñones, Bert Böddeker, and Günther Schäfer. Parallel Execution of AUTOSAR Legacy Applications on Multicore ECUs with Timed Implicit Communication. In *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, San Francisco, USA, 2015.

[KR12] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Pearson, 6th edition, 2012.

[KS14] Evangelia Kasapaki and Jens Sparsø. Argo: A Time-Elastic Time-Division-Multiplexed NOC Using Asynchronous Routers. In *20th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 45–52, May 2014.

[KSS+16] Evangelia Kasapaki, Martin Schoeberl, Rasmus Bo Sørensen, Christoph Müller, Kees Goossens, and Jens Sparsø. Argo: A real-time network-on-chip architecture with an efficient gals implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):479–492, 2016.

[Lis12] Björn Lisper. Towards Parallel Programming Models for Predictability. In *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23, pages 48–58, Dagstuhl, Germany, 2012.

[LN91] Xiaola Lin and Lionel M. Ni. Deadlock-free multicast wormhole routing in multicomputer networks. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, pages 116–125, New York, NY, USA, 1991. ACM.

[May83] David May. Occam. *ACM SIGPLAN Notices*, 18(4):69–79, April 1983.

[McC10] Michael D. McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[Mes15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1*. High Performance Computing Center Stuttgart (HLRS), 2015.

[MFSU17] Jörg Mische, Martin Frieb, Alexander Stegmeier, and Theo Ungerer. Reduced complexity many-core: Timing predictability due to message-passing. In Jens Knopp, Wolfgang Karl, Martin Schulz, Inoue Koji, and Thilo Pionteck, editors, *30th International Conference on Architecture of Computing Systems (ARCS)*, pages 139–151. Springer, Springer International Publishing, 2017.

[MFSU19] Jörg Mische, Martin Frieb, Alexander Stegmeier, and Theo Ungerer. PIMP my Many-Core: Pipeline-Integrated Message Passing. In Dionisios N. Pnevmatikatos, Maxime Pelcat, and Matthias Jung, editors, *2019 International Conference on Embedded Computer Systems: Architectures, Modelling, and Simulation (SAMOS)*, pages 199–211, Cham, July 2019. Springer International Publishing.

[MMU11] Stefan Metzlaff, Jörg Mische, and Theo Ungerer. A real-time capable many-core model. In *Proceedings of 32nd IEEE Real-Time Systems Symposium: Work-in-Progress Session*, pages 21–24, Vienna, Austria, 2011.

[MSM04] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

[MU12] Jörg Mische and Theo Ungerer. Low power flitwise routing in an unidirectional torus with minimal buffering. In *Proceedings of the Fifth International Workshop on Network on Chip Architectures*, NoCArc '12, pages 63–68, New York, NY, USA, 2012. ACM.

[MU14] Jörg Mische and Theo Ungerer. Guaranteed service independent of the task placement in nocs with torus topology. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 151:151–151:160, New York, NY, USA, 2014. ACM.

[NM93] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, February 1993.

[NPB+14] Jan Nowotsch, Michael Paulitsch, Daniel Bühler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *26th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 109–118, July 2014.

[Olo16] Andreas Olofsson. Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip. Whitepaper, October 2016. `https://www.parallella.org/docs/e5_1024core_soc.pdf`.

[Olo17] Andreas Olofsson. EPIPHANY-V: A TFLOPS scale 16nm 1024-core 64-bit RISC Array Processor. In *Hot Chips 29 Poster Session – Symposium on High Performance Chips*, August 2017.

[ONUA14] Andreas Olofsson, Tomas Nordström, and Zain Ul-Abdin. Kickstarting high-performance energy-efficient manycore architectures with Epiphany. In *48th Asilomar Conference on Signals, Systems and Computers*, pages 1719–1726, November 2014.

[OPZ11] Jungju Oh, Milos Prvulovic, and Alenka Zajic. TLSync: support for multiple fast barriers using on-chip transmission lines. In *38th Annual International Symposium on Computer Architecture (ISCA)*, pages 105–115. IEEE, 2011.

[PSK99] Dhabaleswar K. Panda, Sanjay Singal, and Ram Kesavan. Multidestination message passing in wormhole k-ary n-cube networks with base routing conformed paths. *IEEE Transactions on Parallel and Distributed Systems*, 10(1):76–96, 1999.

[Rat10] Justin Rattner. Single-chip Cloud Computer – An experimental many-core processor from Intel Labs. Presentation, 2010. `http://download.intel.com/pressroom/pdf/rockcreek/SCC_Announcement_JustinRattner.pdf`.

[RBS+10] Christine Rochange, Armelle Bonenfant, Pascal Sainrat, Mike Gerdes, Julian Wolf, Theo Ungerer, Zlatko Petrov, and Frantisek Mikulu. WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-Core. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15, pages 90–100, Dagstuhl, Germany, 2010.

[RCS09] Tahiry Ratsiambahotra, Hugues Cassé, and Pascal Sainrat. A versatile generator of instruction set simulators and disassemblers. In *International Symposium on Performance Evaluation of Computer Telecommunication Systems*, volume 41, pages 65–72, July 2009.

[RH90] Michel Raynal and Jean Michel Helary. *Synchronization and control of distributed systems and programs*. Wiley series in parallel computing. Wiley, Chichester, 1990. Translation of : Synchronisation et contrôle des systèmes et des programmes réparties. Paris, Eyrolles.

[SAA+15] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449 – 471, 2015.

[SBSK12] Martin Schoeberl, Florian Brandner, Jens Sparsø, and Evangelia Kasapaki. A Statically Scheduled Time-Division-Multiplexed Network-on-Chip for Real-Time Systems. In *Sixth IEEE/ACM International Symposium on Networks on Chip (NoCS)*, pages 152–160, May 2012.

[Sew18] Ingo Sewing. Implementierung von Ready-Synchronisation und One-to-All + All-to-All-Routing im RC/MC-Prozessor. Project Module, University of Augsburg, March 2018.

[Sew19] Ingo Sewing. Evaluierung und Optimierung des One-to-All-Schedules im echtzeitfähigen RC/MC-Prozessor. Master's thesis, University of Augsburg, March 2019.

[SFMU16] Alexander Stegmeier, Martin Frieb, Jörg Mische, and Theo Ungerer. WCTT bounds for MPI Collectives in the Paternoster NoC. In *14th International Workshop on Real-Time Networks (RTN)*, Toulouse, France, July 2016.

[SFMU18] Alexander Stegmeier, Martin Frieb, Jörg Mische, and Theo Ungerer. Analysing real-time behaviour of collective communication patterns in MPI. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems (RTNS)*. ACM, 2018.

[SGC+16] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, Mar 2016.

[SHM97] David B. Skillicorn, Jonathan M. D. Hill, and William F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6:249–274, 1997.

[SK10] John Sartori and Rakesh Kumar. Low-overhead, high-speed multi-core barrier synchronization. In Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell, editors, *High Performance Embedded Architectures and Compilers*, pages 18–34, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[SMG14] Radu Andrei Stefan, Anca Molnos, and Kees Goossens. dAElite: A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-Up. *IEEE Transactions on Computers*, 63(3):583–594, March 2014.

[SPG97] Scott Shenker, Craig Partridge, and Roch Guerin. Specification of Guaranteed Quality of Service. RFC 2212, RFC Editor, September 1997.

[SSP+11] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18, pages 11–21. OASICS, 2011.

[Ste19] Alexander Stegmeier. *Real Time Analysis of MPI Programs for NoC-based Manycores using Time Division Multiplexing (working title)*. PhD thesis, University of Augsburg, expected to appear 2019.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[Taf16] Thomas Tafertshofer. Taktgenaue Simulation und FPGA-Emulation eines nachrichtengekoppelten Manycores. Master's thesis, University of Augsburg, August 2016.

[TVS07] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, second edition, 2007.

[TW10] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Pearson, 5th edition, 2010.

[UBF+16] Theo Ungerer, Christian Bradatsch, Martin Frieb, Florian Kluge, J. Mische, Alexander Stegmeier, Ralf Jahr, Mike Gerdes, Pavel Zaykov, Lucie Matusova, Zai Jian Jia Li, Zlatko Petrov, Bert Böddeker, Sebastian Kehr, Hans Regler, Andreas Hugl, Christine Rochange, Haluk Ozaktas, Hugues Cassé, Armelle Bonenfant, Pascal Sainrat, Nick Lay, David George, Ian Broster, Eduardo Quiñones, Milos Panić, Jaume Abella, Carles Hernandez, Francisco Cazorla, Sascha Uhrig, Mathias Rohde,

and Arthur Pyka. Parallelizing Industrial Hard Real-time Applications for the parMERASA Multi-core. *Transactions on Embedded Computing Systems (TECS)*, 15(3), 2016.

[Unt18] Tilmann Unte. Analyse und Optimierung der WCET eines parallelen Programms zur Lösung von Blocktridiagonalmatrizen. Bachelor's Thesis, University of Augsburg, March 2018.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[vdWMH11] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Operating Systems Review*, 45(1):73–83, February 2011.

[VHR+08] Sriram R. Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, Vasantha Erraguntla, Clark Roberts, Yatin Hoskote, Nitin Borkar, and Shekhar Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, 2008.

[VLKJ17] Pedro Valero-Lara, Ezhilmathi Krishnasamy, and Johan Jansson. Towards HPC-embedded. Case study: Kalray and message-passing on NoC. *Scalable Computing: Practice and Experience*, 18(2):151–160, June 2017.

[WAE17] Andrew Waterman and Krste Asanović (Editors). The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2. Technical report, RISC-V Foundation, May 2017.

[Wal19] Dominik Walter. Implementierung und Evaluierung des One-to-One-Schedules im echtzeitfähigen RC/MC-Prozessor. Master's thesis, University of Augsburg, February 2019.

[WEA+08] Reinhard Wilhelm, Jakob Engblom, Ermedahl Aandreas, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution Time Problem – Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.

[WGH+07] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *Micro, IEEE*, 27(5):15–31, September-October 2007.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, New York, NY, USA, June 1995. ACM.

[WS85] Colin Whitby-Strevens. The transputer. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA '85, pages 292–300, Los Alamitos, CA, USA, June 1985. IEEE Computer Society Press.

List of Figures

List of Tables

List of Code Examples

Acronyms

1:1 One-to-One. 15, 55, 76, 77, 79, 80, 82, 88, 112, 113, 114, 115, 117, 119, 121, 123, 124, 129, 135

1:A One-to-All. 15, 19, 46, 51, 53, 56, 76, 77, 78, 79, 80, 82, 85, 88, 95, 98, 103, 112, 113, 114, 115, 116, 117, 119, 121, 123, 124, 129, 134, 135, 151, 152, 170

A:A All-to-All. 14, 15, 14, 15, 25, 76, 77, 79, 80, 82, 112, 113, 114, 115, 117, 119, 121, 123, 124, 135

ACET average-case execution time. 52

ALM Adaptive Logic Module. 23, 49, 63, 82, 106, 116

BCET best-case execution time. 61

BCU barrier control unit. 86, 89, 90, 93, 94, 95, 96, 98, 99, 100, 105, 106, 107, 116, 134, 151, 167

BE best effort. 13, 32

BSP Bulk Synchronous Parallel. 16, 18, 32, 56, 62, 85, 92, 133, 151

CG conjugate gradient. 117, 118, 119, 121, 129, 135, 151

CoMPSoC Composable and Predictable Multi-Processor System on Chip. 7

CSR Control and Status Register. 22, 153, 154, 165, 168

DMA direct memory access. 6, 7, 8, 135

ECU Electronic Control Unit. 1, 13

FIFO first in first out. 12, 14, 32, 37, 90

FPGA Field-Programmable Gate Array. 23, 63, 134

G-Line global interconnection line. 53, 87

GHDL G Hardware Design Language. 23

GS guaranteed service. 13, 16

ISA instruction set architecture. 7, 10, 20, 22, 23, 24, 161

LU Lower-Upper symmetric Gauß-Seidel. 126, 127, 128, 129, 135, 152

LUT look-up table. 23

MacSim many-core simulator. 23, 24, 48, 56, 134

MG MultiGrid. 121, 123, 124, 129, 135, 151, 152

MPI message passing interface. 18, 19, 20, 25, 27, 29, 64, 94, 107, 133, 153

MPPA Massively Parallel Processor Array. 8

NI network interface. 2, 5, 7, 12, 22, 40, 41, 54, 55, 56, 60, 161, 167

NoC network-on-chip. 2, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 19, 22, 23, 24, 27, 33, 34, 35, 36, 45, 46, 51, 52, 53, 76, 77, 76, 77, 78, 77, 78, 79, 80, 82, 86, 87, 88, 90, 92, 96, 98, 103, 102, 105, 106, 107, 113, 114, 115, 119, 133, 135, 151, 155, 170

OTAWA Open Tool for an Adaptive WCET Analysis. 24, 25

PE processing element. 10, 11, 18, 22, 33, 35, 40, 41, 43, 46, 87, 96, 100, 101, 102

PIMP pipeline-integrated message passing. 12, 22, 43, 46, 61, 90

RC/MC Reduced Complexity Many-Core. 2, 7, 8, 9, 10, 11, 12, 15, 16, 20, 22, 23, 24, 31, 32, 33, 36, 45, 50, 63, 88, 98, 100, 105, 106, 117, 129, 133, 151, 153, 154, 165, 168

RISC Reduced Instruction Set Computing. 90

sawp stop-and-wait protocol. 34, 36

SCC Single-chip Cloud Computer. 7

SoC System-on-Chip. 34

TDM time-division multiplexing. 6, 13, 31, 32, 45, 46, 50, 82, 134

TDMA time-division multiple access. 34

VHDL Very High Speed Integrated Circuit Hardware Description Language. 22, 23, 24, 42, 49, 56, 63, 100, 106, 134

WCET worst-case execution time. 1, 2, 9, 11, 16, 18, 19, 24, 25, 27, 29, 61, 65, 67, 68, 71, 72, 73, 75, 76, 77, 78, 77, 78, 101, 103, 107, 110, 112, 113, 114, 117, 118, 119, 121, 123, 121, 123, 124, 128, 129, 133, 135, 151, 152, 153

WCTT worst-case transportation time. 13, 14, 18, 19, 24, 25, 27, 53, 61, 67, 73, 74, 77, 78, 79, 107, 112, 113, 114, 115, 119, 124, 129, 133, 153

A

Overview on RISC-V Instruction Set Extensions

In the publications containing work of the RC/MC project (e.g. [MFSU17]) including this thesis, we introduced several instruction set extensions to the RISC-V ISA [WAE17]. We give an overview on them in this appendix chapter.

A.1. Overview on our Instructions

Table A.1.: Overview on our RISC-V instruction set extension for sending and receiving flits with pipeline-integrated message-passing (copy of Table 3.3) [MFSU19, Gor18].

mnemonic	destination register	source register 1	source register 2	function
`snd`		*receiver*	*payload*	send *payload* to *receiver*
`rcvp`	*payload*			store the *payload* from the oldest flit in the receive buffer in the destination register and remove this flit from the receive buffer
`rcvn`	*node id*			store the *node id* of the sender from the oldest flit in the receive buffer in the destination register

Table A.2.: Overview on our RISC-V instruction set extension checking the status of pipeline-integrated message-passing (copy of Table 3.4) [MFSU19, Gor18].

mnemonic	source register 1	source register 2	immediate value	function
`bsf`			*address*	when send buffer is full, jump to *address*
`bsnf`			*address*	when there is space left in the send buffer, jump to *address*
`bre`			*address*	when receive buffer is empty, jump to *address*
`brne`			*address*	when there is a flit waiting in the receive buffer, jump to *address*

Table A.3.: Overview on our RISC-V instruction set extension for `ready` synchronization (copy of Table 4.1, originally published in [FSMU18]).

mnemonic	source register 1	source register 2	immediate value	function
`srdy`	*sender*			send `ready` flit to *sender*
`bnr`	*receiver*		*address*	check if *receiver* is ready; when it is not, jump to *address*

Table A.4.: Overview on our RISC-V instruction set extension for broadcasts and multicasts (copy of Table 5.1.

mnemonic	source register 1	source register 2	immediate value	function
`mcst`	*message*			broadcast/multicast a 64 bit *message*
`bnar`			*address*	branch to *address* when not *all* nodes are ready yet
`bnra`	*part*	*nodes*	*address*	branch to *address* when the nodes in the indicated *part* of the `Ready Bit Array` are not yet `ready` *part* has to be 0 when up to 64 nodes are present
`mrdy`	*sender*	*number*		send `ready` flit to *sender* and store *sender* and *number* at NI to accept only *number* multicast flits from *sender*

Table A.5.: Overview on our RISC-V instruction set extension for barriers (copy of Table 6.1).

mnemonic	source register 1	source register 2	immediate value	function
`brrav`	*node*	*uid*		arrive at barrier which is coordinated by *node* and has unique id *uid*
`bbnr`			*address*	branch to *address* when the barrier where the node participates via `brrav` has not yet been released
`cbrr`	*part*	*nodes*		configure the *part* of the `barrExpected` register to set/reset bits for given *nodes*
`mbrr`	*mode*			when the content of *mode* is set (is non-zero), enter configuration mode (no barrier is released). When it is reset to 0, leave configuration mode.

A.2. Encoding of our Instructions

Table A.6.: Encoding of PIMP I/O related instructions in our RISC-V instruction set extension.

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	000	00000	1011011	`snd`
0000000	00000	00000	010	rd	1011011	`rcvn`
0000000	00000	00000	011	rd	1011011	`rcvp`

Table A.7.: Encoding of PIMP status related instructions in our RISC-V instruction set extension.

31 25	24 20	19 15	14 12	11 7	6 0	
imm[12|10:5]	00000	00000	000	imm[4:1|11]	1111011	`bsf`
imm[12|10:5]	00000	00000	001	imm[4:1|11]	1111011	`bsnf`
imm[12|10:5]	00000	00000	010	imm[4:1|11]	1111011	`bre`
imm[12|10:5]	00000	00000	011	imm[4:1|11]	1111011	`brne`

Table A.8.: Encoding of `ready` related instructions in our RISC-V instruction set extension (copy of Table 4.2).

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	00000	rs1	001	00000	1011011	`srdy`
imm[12|10:5]	00000	rs1	101	imm[4:1|11]	1111011	`bnr`

Table A.9.: Encoding of broadcast/multicast related instructions in our RISC-V instruction set extension (copy of Table 5.2).

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	101	00000	1011011	`mrdy`
0000000	00000	rs1	110	00000	1011011	`mcst`
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1111011	`bnra`
imm[12|10:5]	00000	00000	111	imm[4:1|11]	1111011	`bnar`

Table A.10.: Encoding of barrier related instructions in our RISC-V instruction set extension (copy of Table 6.2).

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	100	00000	1011011	`brrav`
imm[12|10:5]	00000	00000	110	imm[4:1|11]	1111011	`bbnr`
0000000	rs2	rs1	111	00000	1011011	`cbrr`
0100000	00000	rs1	111	00000	1011011	`mbrr`

A.3. Control and Status Registers

Table A.11.: Overview on RC/MC specific CSRs (copy of Table 3.5).

CSR #	description
`0xc70`	Number of nodes present on the platform
`0xc71`	ID of the node where the code is executed

B

Overview on Implementation Details

B.1. messageTypes for Flits in the Network-on-Chip

Table B.1.: Overview on `messageTypes`.

`messageType`	Description
`none`	Indicates that the content of this flit is not valid.
`data`	"Normal" data flit
`ready`	This flit has no payload. It is to be received by the NI of the receiver to store the information that the sender node is `ready`.
`mcst`	Multicast flit, which is to be copied and forwarded until it has reached the farthermost node.
`barrival`	Barrier arrival flit signalizing that the sender node has arrived at a barrier.
`barrelease`	Barrier release flit, which is to be copied and forwarded like a multicast flit. It tells the BCU of the nodes that the barrier is released and they can continue their execution

B.2. Execution times of RISC-V Instructions on the RC/MC

Table B.2.: Overview on RV64IM instructions and their execution times.

Type	Instruction(s)	ex. time
Simple ALU instructions	`add, addi, sub, and, andi, or, ori, xor, xori, sll, slli, sra, srai, srl, srli, slt, sltu, slti, sltiu, lui, auipc, addiw, addw, subw, slliw, srliw, sraiw, sllw, srlw, sraw`	1
Unconditional Jumps	`jal, jalr`	2
Branches	`beq, bne, blt, bltu, bge, bgeu`	2
Integer multiplication	`mul, mulh, mulhu, mulhsu, mulw`	5
Integer division	`div, divu, rem, remu, divw, divuw, remw, remuw`	35
Loads / Stores[†]	`lb, lbu, lh, lhu, lw, lwu, ld, sb, sh, sw, sd`	1
CSR accesses	`csrrw, csrrs, csrrc, csrrwi, csrrsi, csrrci`	1
Syscalls	`ecall, ebreak`	3

[†] Due to scratchpad implementation of the RC/MC, loads/stores only require one cycle.

Table B.3.: Overview on RV64FD instructions and their execution times.

Type	Instruction(s)	ex. time
Small floating point instructions	`fminmaxs, fminmaxd, fsgnjs, fsgnjd, fcmps, fcmpd, fclasss, fclassd`	1
Floating Point Conversion Operations	`fcvts, fcvtd, fcvt2s, fcvt2d, fcvt3d, fmv, fmv2, fcvt3s, fcvt4s, fcvt4d, fcvt5d, fmv3, fmv4`	2
Common floating point operations	`fadds, faddd, fsubs, fsubd, fmuls, fmuld, fmuladds, fmuladdd, fmulsubs, fmulsubd, fnegmuladds, fnegmuladdd, fnegmulsubs, fnegmulsubd`	4
Floating Point Division (Single Precision)	`fdivs`	33
Floating Point Division (Double Precision)	`fdivd`	41
Floating Point SQRT (Single Precision)	`fsqrts`	45
Floating Point SQRT (Double Precision)	`fsqrtd`	57
Floating Point Load / Store operations†	`flw, fld, fsw, fsd`	1

† Due to scratchpad implementation of the RC/MC, loads/stores only require one cycle.

Table B.4.: Overview on RC/MC specific instructions and their execution times.

Type	Instruction(s)	ex. time
Send/receive	`snd, rcvn, rcvp, srdy, mcst, mrdy, brrav`	3
Status branches	`bsf, bsnf, bre, brne, bnr, bnar, bnra, bbnr`	3
Barrier configuration	`cbrr, mbrr`	1

B.3. The One-To-All Schedule without Corner Buffers

Originally, the 1:A schedule was proposed by Mische [MU14]. At his implementation, corner buffers are required: They store flits which arrived at the target column until they can take their reserved slot to their target node. Sewing investigated several variations of the 1:A schedule and developed an own *variant 3* without the need for corner buffers [Sew19]. Because it has the same timing guarantees, but reduces hardware effort, we decided to use this variant in our thesis. In this section, we describe how it works.

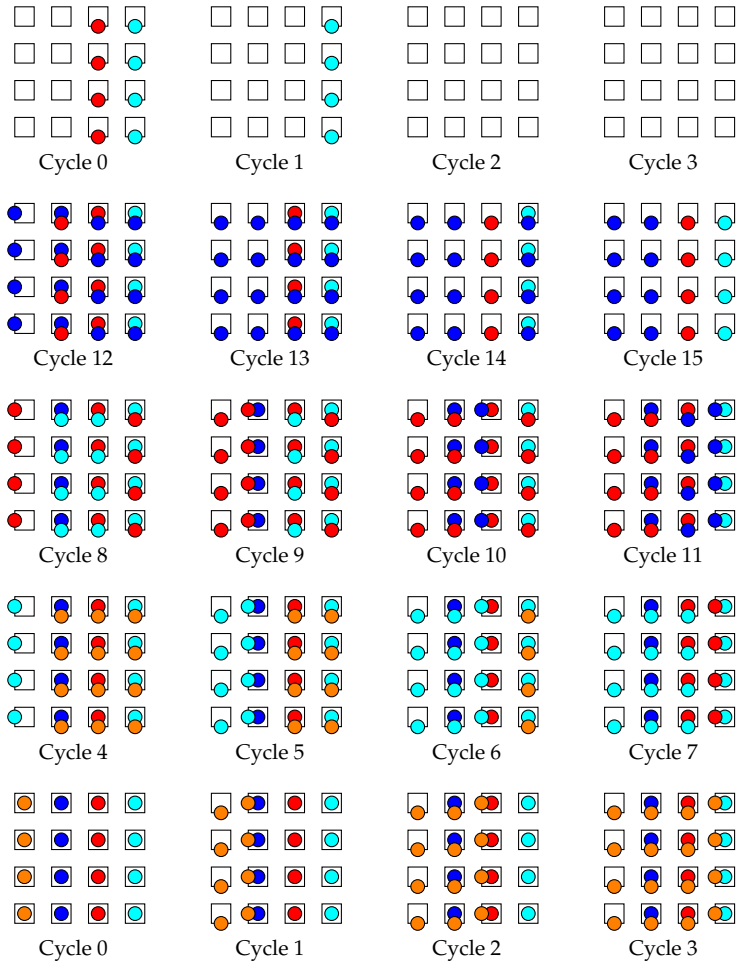


Figure B.1.: Workflow of *variant 3* of the 1:A schedule as developed by Sewing [Sew19]. We illustrate all possible paths to clarify that there are no colissions and no conflicts. Rectangles represent nodes, circles represent flits. Flits on the borders of a node are arriving at the south or west in ports of this node. **The figure should be read from the left bottom to the right top**. Figure based on [Sew19].

Figure B.1 displays the workflow of one period of the 1:A schedule, i.e. it starts when flits are sent and ends when these flits are delivered at their target nodes. The figure is divided into 20 small figures, each representing one cycle of the execution. These small figures show 16 rectangles, illustrating nodes. Furthermore, we see colored circles, showing flits travelling through the NoC. Circles in the middle of a node are flits waiting to be sent out. A flit appearing on the border of a node arrives at this node in the current cycle (on the left side it comes from west, on the bottom side from the south). When this flit arrived at its target node, it disappears at the next cycle in the figure. Otherwise, it will be seen at the border of the next node. At the **left bottom** of Figure B.1 at cycle 0, we see the situation before the period starts: All nodes want to send a flit. Because the schedule works off flits column-wise, each column appears in a different color. In the figure, we follow *all* possible paths to see that there are no collisions and no conflicts.

In cycle 1 (second figure in the **bottom row**), all flits from the first column are sent (orange circles): When their target is in the first column, they immediately take their way to the north. Otherwise, they travel to the west until they reach their target column (cycles 1-3). As soon as they arrived at their target column, the flits are sent to the north until they reached their target node (cycles 2-6). Thereby, the first column is a special case because it is the only one where flits are sent to the north and to the west at the same time. In the other columns, first only flits to the west are sent and later those to the north in the same column. This can be seen e.g. at the light blue flits in the last column, which are to be sent next. When their target node is **not** in the same column, they are sent out in cycle 4, otherwise in cycle 15. Flits travelling to the west are on their way to their target column in cycle 4-6. When these flits have arrived there, they are sent to the north from cycle 5 (for the first column) to cycle 9 (for the third column). All light blue flits in the last column which are already in their target column are sent out in cycle 15, arriving at cycle 1 (at the top row of the figure).

The red and dark blue flits are worked off the same way. Only when sent to the west, they start at cycle 7 (red flits) or at cycle 10 (dark blue flits). When the red or dark blue flits are already in their target column, they are sent out at cycle 13 (dark blue flits) or cycle 14 (red flits). The last flits arrive at their target nodes in cycle 0 and 1 of the next period (top row). This does not lead to conflicts or collisions, because in cycles 1 and 2 there are no other flits arriving from the south at these columns.