Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-2012-07

M.Sc. Thesis

MEP-MAS: A Message Passing Multiprocessor Array for Streaming Applications

Mitzi E. Tjin A Djie

Abstract

This thesis presents the design and implementation of a Chip-Multiprocessor (CMP) targeted at streaming applications(e.g. MPEG, MP3). Streaming applications are applications which can be split into several distinct stages working on data elements in a pipelined fashion. We propose a distributed-memory array (MEP-MAS), where the cores communicate via message-passing, optimizing the throughput. Application tasks are dynamically scheduled by a hardware scheduler taking the consumer-producer locality into account, thereby minimizing the communication overhead. The array is evaluated in terms of performance, scalability and predictability as a function of varied input stream sizes, multiple pipelines, number of pipeline stages and traffic volume. The array is configured as a 4 by 5 mesh and has reached speedups as high as 3.6x for a 4-stage pipeline and 13.4x for a 16-stage pipeline. Our experiments have highlighted the need for a balanced workload in order to optimize the performance. Furthermore, it is shown that MEP-MAS is scalable as the speedup and throughput almost linearly increases with the the number of added pipelines. The speedup has increased from 3.6x to 13.5x and the throughput from 17k data elements per second to 65k data elements per second. Increasing the traffic volume in the network marginally affects the speedup (-1.9%). Finally, increasing the traffic volume can cause a high deviation in arrival times between two subsequent data blocks in the pipeline of up to 8%.

MEP-MAS: A Message Passing Multiprocessor Array for Streaming Applications

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Engineering

by

Mitzi E. Tjin A Djie
born in Paramaribo, Suriname

This work was performed in:

Circuits and Systems Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

Copyright © 2012 Circuits and Systems Group
All rights reserved.

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"MEP-MAS: A Message Passing Multiprocessor Array for Streaming Applications"** by **Mitzi E. Tjin A Djie** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 05-09-2012

Chairman:

prof.dr.ir. A.J. van der Veen

Advisors:

dr.ir. T.G.R.M van Leuken

ir. Sumeet Kumar

Committee Members:

dr.ir. J.S.S.M. Wong

Abstract

This thesis presents the design and implementation of a Chip-Multiprocessor (CMP) targeted at streaming applications(e.g. MPEG, MP3). Streaming applications are applications which can be split into several distinct stages working on data elements in a pipelined fashion. We propose a distributed-memory array (MEP-MAS), where the cores communicate via message-passing, optimizing the throughput. Application tasks are dynamically scheduled by a hardware scheduler taking the consumer-producer locality into account, thereby minimizing the communication overhead. The array is evaluated in terms of performance, scalability and predictability as a function of varied input stream sizes, multiple pipelines, number of pipeline stages and traffic volume. The array is configured as a 4 by 5 mesh and has reached speedups as high as 3.6x for a 4-stage pipeline and 13.4x for a 16-stage pipeline. Our experiments have highlighted the need for a balanced workload in order to optimize the performance. Furthermore, it is shown that MEP-MAS is scalable as the speedup and throughput almost linearly increases with the the number of added pipelines. The speedup has increased from 3.6x to 13.5x and the throughput from 17k data elements per second to 65k data elements per second. Increasing the traffic volume in the network marginally affects the speedup (-1.9%). Finally, increasing the traffic volume can cause a high deviation in arrival times between two subsequent data blocks in the pipeline of up to 8%.

Acknowledgments

I would like to thank my advisor dr.ir. T.G.R.M van Leuken and Sumeet Kumar for allowing me to be part of this project and advising me during the coarse of the project. Special thanks to Sumeet, who was there for me on day to day basis and always checking up on me when needed. Furthermore I would like to thank Anthony Brandon and Roel Seedorf for helping me getting started with the ρ-Vex processor and toolchain. And last but not least I would like to thank my friends and family who have always supported me and helped me through tough times.

Mitzi E. Tjin A Djie
Delft, The Netherlands
05-09-2012

Contents

Abstract **v**

Acknowledgments **vii**

1 Introduction **1**
1.1 Problem Statement . . . 1
1.2 Thesis Goals . . . 2
1.3 Contributions . . . 2
1.4 Thesis Organization . . . 3

2 Background **5**
2.1 Parallelism . . . 5
2.2 Exploiting TLP and DLP . . . 5
2.3 Streaming applications . . . 6
2.3.1 The Stream Programming Model . . . 6
2.4 A Shared-memory Multiprocessor architecture . . . 7
2.5 A Distributed Memory Multiprocessor architecture . . . 8
2.6 Performance analysis and comparison of a Shared-memory architecture and a Message-Passing architecture . . . 8
2.6.1 Experimental setup . . . 9
2.6.2 Results . . . 11
2.6.3 Scalability . . . 12
2.6.4 Conclusion . . . 14
2.7 Related work . . . 14
2.7.1 The Intel SCC . . . 14
2.7.2 C-HEAP . . . 15
2.7.3 The IBM Cell-processor . . . 16
2.7.4 The PicoChip . . . 17
2.7.5 The Ambric parallel processor . . . 18
2.7.6 AsAP . . . 19
2.8 Summary . . . 19

3 System Overview **21**
3.1 System Architecture . . . 21
3.2 Scheduling Policy for scheduling on the VLIW array . . . 23
3.3 Overview of the Message Passing Architecture . . . 23
3.4 Summary . . . 25

4 Architecture **27**
4.1 The Distributed Scheduler . . . 27
4.1.1 The Primary Scheduler . . . 27
4.1.2 The Secondary Schedulers . . . 29

4.2 The Message Passing Tile . . . 30
4.2.1 The Processing Element . . . 30
4.2.2 Data Memory . . . 31
4.2.3 Instruction Cache . . . 32
4.2.4 The Bootloader . . . 32
4.2.5 Data Interface . . . 32
4.2.6 The Message Passing Buffer . . . 32
4.2.7 The Buffer Manager . . . 32
4.2.8 Outgoing Buffer . . . 33
4.2.9 The DMA Controller . . . 34
4.2.10 The Network Interface . . . 36
4.2.11 The Network Interface Bridge . . . 39
4.3 Summary . . . 40

5 Experimental Setup and Results **41**
5.1 The Baseline Platform . . . 41
5.2 The Applications . . . 41
5.2.1 The JPEG Decoder . . . 42
5.2.2 The FIR Filter . . . 43
5.2.3 Custom Application . . . 45
5.2.4 The Scheduling Overhead . . . 46
5.3 Performance Evaluation Metrics . . . 46
5.4 Performance Evaluation Process . . . 48
5.5 The Experiments . . . 48
5.5.1 Increased Input Sizes . . . 49
5.5.2 Multiple Pipelines . . . 51
5.5.3 Varied buffer sizes . . . 52
5.5.4 Varied number of pipeline stages . . . 53
5.5.5 Increased traffic volume . . . 54
5.6 Summary . . . 62

6 Conclusion **65**
6.1 Summary . . . 65
6.2 Future Work . . . 66

List of Figures

2.1 The mapping of an application when exploiting DLP. 6
2.2 The mapping of a streaming application. 6
2.3 A Dataflow Process Network 7
2.4 The basic structure of a shared-memory architecture 8
2.5 The basic structure of a Distributed Memory architecture 8
2.6 The shared-memory configuration used in this simulation 9
2.7 The message-passing configuration used for this simulation 10
2.8 The dataflow graph of the simulated application 10
2.9 The latency for different traffic volumes 11
2.10 The Speedup for different traffic volumes 12
2.11 The mapping of the multiple pipelines on the platform for every application 13
2.12 The latency for different traffic volumes for the scaled up configurations 13
2.13 The speedup for different traffic volumes for the scaled up configurations 13
2.14 Top-level view of the SCC multiprocessor 14
2.15 The CHEAP architecture 16
2.16 The Cell processor architecture 17
2.17 The PicoChip architecture 18
2.18 The Ambric channel structure 19

3.1 The complete multiprocessor architecture configuration 22
3.2 Tasks separated by markers. 22
3.3 The basic components of a message passing tile 23
3.4 The FIFO problem 24
3.5 The process of sending a message form core 0 to core 1 25

4.1 The basic structure of the primary scheduler. 28
4.2 Data Flow Graph representation 28
4.3 The different traversing orders 30
4.4 The LUT structure 30
4.5 The Message Passing Tile 31
4.6 The behavior of the Buffer Manager 33
4.7 Illustration of the disadvantage for multiple message send approvals . . 36
4.8 The FSM of the DMA-controller 38
4.9 The structure of a packet 39

5.1 The baseline platform configuration 42
5.2 The data flow graph of the JPEG decoder 42
5.3 The data flow graph of the JPEG decoder with four stages 43
5.4 The affect of adding a fourth core to a pipeline with an unbalanced workload 44
5.5 The serial moving average filter implementation 45
5.6 The partitioned moving average filter implementation 45
5.7 The data-flow graph of the FIR filter 46

5.8 The inner and outer loop transitions 47
5.9 The data-flow graph of the custom application 48
5.10 The different pipeline mappings 50
5.11 The Speedup as a function of an increased input-stream size 50
5.12 Execution breakdown for each application 51
5.13 The mapping of the multiple pipelines on the platform for every application 52
5.14 The Speedup as a function of the number of pipelines 52
5.15 The Throughput as a function of the number of pipelines 53
5.16 The Speedup as a function of the MPB size 53
5.17 The Speedup as a function of the number of partitions 54
5.18 The Speedup as a function of the number of partitions 54
5.19 Top-level view of the two platform configurations 55
5.20 The average arrival rate as a function of the injection rate for each application 56
5.21 The critical path of the FIR filter 57
5.22 The critical path of the custom application 58
5.23 The critical path of the JPEG decoder 58
5.24 The average deviation as a function of the injection rates 59
5.25 The average arrival rate as a function of the injection rate for each application 60
5.26 The different pipeline mappings 60
5.27 The average deviation as a function of the injection rates 61
5.28 The speedup for the first configuration 61
5.29 The speedup for the second configuration 62

List of Tables

2.1 Memory Latencies 9
2.2 NOC specifications 11

4.1 Control Registers 35
4.2 DMA-controller operations 37
4.3 The FSM states 37
4.4 The communication classes 38
4.5 The communication ID's and their descriptions 39

5.1 The run-times of the three partitions of the JPEG decoder 43
5.2 Instruction counts of the JPEG encoder 44
5.3 The run-times of the FIR filter 46
5.4 Instruction counts of the FIR filter 47
5.5 The run-times of the three partitions of the Custom Application 48
5.6 Instruction counts of the Custom Application 49
5.7 The scheduling overhead for the different applications 49
5.8 The execution times of core 1 for the 3-core and 4-core JPEG pipelines 51
5.9 The injection rates 55

1 Introduction

Current software applications demand high performance levels, which has exposed the limitations of microprocessors. Limitations such as long-wire delays and a limited amount of available instruction level parallelism in applications has shifted the focus towards Chip-multiprocessors (CMPs). CMPs consists of multiple processing elements which can be used to execute tasks in parallel. In order for an application to be executed on a CMP, it must be partitioned into multiple tasks, which can be executed in parallel on one of the processing elements. Several partitioning strategies exist for mapping applications onto a CMP, one of which is the stream programming model. Applications in the stream programming model are partitioned into distinct stages, where each stage works on data elements in a data stream. The stages are executed in a pipelined fashion and each stage in the pipeline requires the result of the previous stage in the pipeline. Stages in the pipeline execute in parallel, which reduces the execution time of the application. The main advantage of the stream programming model is that every application, where data is being processed iteratively, can be pipelined.

1.1 Problem Statement

Communication and scheduling are two important aspects that must be considered when designing CMPs targeted at streaming applications. There are two common CMP architectures that support different types of communication. The first architecture is a shared-memory architecture where tasks share one global memory and communicate by writing data to the shared-memory. Communication between tasks on a shared-memory architecture is implicit and not exposed to the application developer, which makes programming for this architecture relatively easy. However applications executed on shared-memory architectures may suffer from high communication overheads, as all communication has to go through the shared memory. This increase in communication overhead results in lower speedups. In a distributed-memory architecture tasks have private memories and communicate through message passing. The application developer must explicitly send and receive messages, by calling specific message-passing functions and is, therefore responsible for communication and synchronization which decreases the ease of programming on distributed-memory systems. However, distributed architectures present the possibility to exploit producer-consumer locality and, in the case of streaming applications, intermediate results can directly be sent to the next stage in the pipeline, possibly located on a neighboring core.

As previously mentioned, the scheduling method is also an important aspect that must be considered when designing a CMP targeting streaming applications. Dynamic scheduling requires scheduling logic in hardware, but has the advantage that run-time information can be used to make scheduling decisions. On the other hand, tasks can

be scheduled statically at compile-time, which avoids the need for scheduling logic but decreases the scheduling flexibility.

In summary, designing an architecture which is optimized for streaming applications is not trivial. In order to design such an architecture, we need to:

1. Design and implement a message-passing architecture that enables fast and efficient data transfers while maximizing the ease of programming.
2. Design a scheduler that implements an optimal scheduling strategy for streaming applications.

1.2 Thesis Goals

In the previous section we have mentioned the importance of the communication and scheduling methods for streaming applications, to that end we set the following goals:

- Design of a message-passing array to facilitate the communication between processors.
- Design of a distributed hardware scheduler, which takes the producer-consumer locality into account.
- Implementing a message-passing library, which allows for communication between tasks.
- Evaluating the proposed design in terms of speedup, throughput and predictability as a function of varied input stream sizes, multiple pipelines, varied number of pipeline stages and increased traffic volumes.

1.3 Contributions

In this thesis we propose MP-MAS, a Message Passing Multiprocessor Array for Streaming applications. MP-MAS offers a flexible, processor-architecture independent message-passing array of processors, where application tasks are dynamically scheduled. MP-MAS also includes a message-passing library containing easy-to-use message-passing functions. The main contributions of our thesis are, accordingly:

- The design and implementation of a distributed-memory array targeted at streaming applications, where the cores communicate through message-passing.
- The design and implementation of a dynamic distributed scheduler, which exploits producer-consumer locality. The scheduler implements two scheduling policies, increasing the flexibility of our scheduler.
- An in-depth performance evaluation of the multiprocessor architecture, evaluating the performance, scalability and the sensitivity to the unpredictable nature of the network. We have implemented and simulated a 4x5 mesh containing 18 processors and the distributed scheduler and observed speedups of 3.6x for a

4-stage pipeline and 13.4 for a 16-stage pipeline. Furthermore, with regard to the predictability, the maximum observed deviation of the arrival times of two subsequent data blocks is 8%.

1.4 Thesis Organization

This thesis is organized as follows:

Chapter 2 defines the basic concepts of parallel processing. Two common chip multiprocessor architectures (shared memory and distributed memory) are described. Furthermore, we evaluate the performance of both these architectures by conducting a small experiment. Finally, this Chapter provides an overview of several existing message-passing architectures. Chapter 3 gives an overview of the complete architecture, which will be discussed in detail in Chapter 4. In Chapter 5 we evaluate the performance of the proposed architecture by conducting several experiments to evaluate different aspects of the architecture. Finally, Chapter 6 provides a summary of the work done during the project and recommendations for future work.

2 Background

This Chapter starts with several concepts of multiprocessing and introduces the stream programming model. Next, two common Multiprocessor architectures are discussed, which includes a shared-memory architecture and a distributed-memory architecture. The performance of both architectures are measured by means of a simulation of the network behavior for both architectures. The experimental setup and the results of the simulation are presented and finally, this Chapter ends with a description of several existing Message Passing architectures.

2.1 Parallelism

Applications can contain different types of parallelism which can be exploited when executed on a multiprocessor. Exploiting these types of parallelism increases the execution performance. There are tree types of parallelism: Instruction-level parallelism (ILP), Data-level parallelism (DLP) and Task-level parallelism (TLP).

- ILP: Instructions that can be executed independently from each other, can be executed in parallel. Instruction B is independent from instruction A if instruction B does not need the result from instruction A to execute correctly.
- DLP: Applications that contain DLP have a dataset that is distributed amongst multiple tasks. The tasks contain the same set of instructions and can process their part of the dataset in parallel.
- TLP: Applications that consists of different tasks, contains TLP. Each task can be executed on a different processor in parallel. The tasks can work on the same or different parts of the dataset and can consist of the same or different set of instructions. Note that DLP is a form of TLP.

2.2 Exploiting TLP and DLP

Exploiting the different types of parallelism directly determines the mapping of an application on to a multiprocessor. Applications that contain DLP are usually partitioned in such a way that every task contains the same program code and works on a different part of the dataset. Figure 2.1 shows an example of the mapping of an application containing DLP onto a multiprocessor, where every task executes the same program code and works on the different data blocks A,B,C. Applications that contain TLP are typically divided into multiple tasks all working on parts of the same or different data. A special case of applications containing TLP are streaming applications were the tasks work on data elements in a stream of data. Figure 2.2 shows an example of

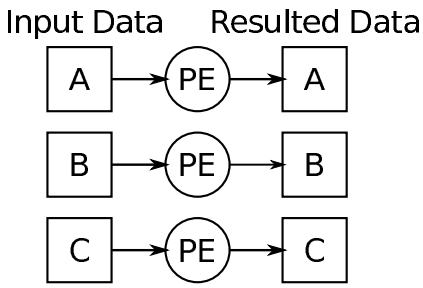


Figure 2.1: The mapping of an application when exploiting DLP.

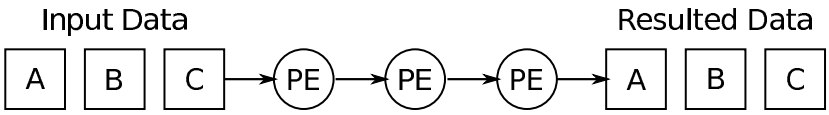


Figure 2.2: The mapping of a streaming application.

such a streaming application. Because the tasks only depend on the next data block in the stream, the tasks can be pipelined.

2.3 Streaming applications

There are a number of application domains that can be considered as streaming applications, usually within the digital signal processing domain. Examples include, multimedia processing (e.g. MPEG, JPEG) and FIR filters.

2.3.1 The Stream Programming Model

A program or application can consists of several distinct phases. Each phase can be considered as a task that processes input data and produces output data. The output data is further processed by the next phase or task. The tasks form a pipeline, where each stage processes data elements in a stream of data. The data elements are communicated through FIFO's between tasks. The tasks read and write data elements to and from these channels.

Kahn Process Networks(KPN's) [1] [2] is a common streaming model. The data elements in a KPN are called tokens. Each token is produced and consumed only once. Tasks block when they try to write to a full channel or read from an empty channel. Once a channel has been established between two tasks it can not be removed. The KPN's can be represented by a graph where the nodes represent tasks and the arcs represent FIFO channels. Figure 2.3 shows an example of a KPN. A special case of a KPN is a Dataflow Process Network [3]. The nodes in Dataflow Process Networks represent tasks and the arcs represent the data streams. In this thesis these dataflow networks will be used to describe the dependencies between processes in streaming applications, where the processes are tasks executed on a PE.

The tasks belonging to streaming applications execute in a pipelined fashion. Each task in the pipeline executes subsequent blocks in the data stream in parallel, reducing

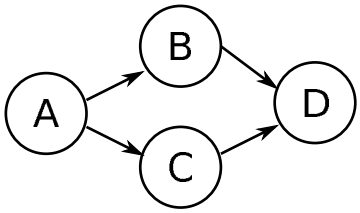


Figure 2.3: A Dataflow Process Network

the total execution-time of the application. Another advantage of the stream programming model is that every task that processes data iteratively can be pipelined, which is a rather large set of applications.

The stream programming model also has limitations, one of which is the fact that the tasks in the pipeline must have an equal workload for an optimal performance. The rate at which the pipeline can process data depends on the slowest task in the pipeline. Balancing the workload can be quite difficult and is sometimes not possible.

2.4 A Shared-memory Multiprocessor architecture

Tasks executed on Shared-memory multicore architectures share one address space that addresses a single centralized memory [4] [5]. Figure 2.4 shows what the structure of such an architecture looks like. There are several levels within the memory hierarchy. The first level is usually a small private data cache placed close the Processing Elements (PE's). The small cache size and the close connection to the PE's allows for single-cycle access times. The next level in the memory hierarchy is the shared l2-cache, this cache has a bigger size and is placed further away from the PE's. The increased size and distance between the PE's and the L2-cache increases the access time of this cache. The last level is the main memory, which is large and is kept off-chip. The PE's are connected via an interconnect. As the small caches are private, some coherency algorithm and logic is needed to keep all data in the private caches up to date.

The main advantage of a shared-memory architecture is that it is relatively easy to program in comparison to the distributed-memory architecture. Communication between tasks executed on a shared-memory platform is implicit and not exposed to the programmer.

One disadvantage of the shared-memory architecture with regard to streaming applications is the increased communication overhead caused by simultaneous reads and writes by the pipeline stages to the shared-memory. All stages in the pipeline need to write their intermediate result to the shared-memory and subsequently read new data from the shared-memory every iteration. This increases the traffic traveling towards the shared-memory and the contention between the stages for the shared-memory. These consequences are amplified for stages with an equal run-time.

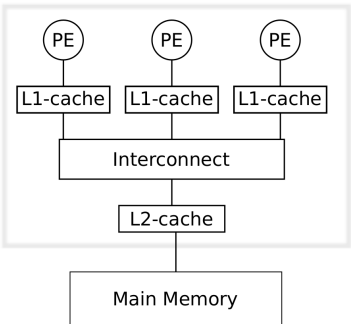


Figure 2.4: The basic structure of a shared-memory architecture

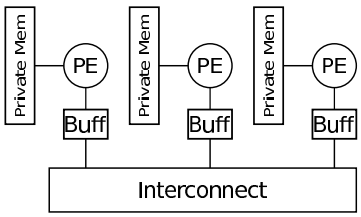


Figure 2.5: The basic structure of a Distributed Memory architecture

2.5 A Distributed Memory Multiprocessor architecture

The memory in distributed-memory architectures [6] is physically distributed over the chip and PE's have their local private memory. Tasks executed on such an architecture have a private address space. When data from a remote PE is required, data must be communicated over the network from the remote PE. The method used to exchange this data is called *Message Passing*. Figure 2.5 gives the basic structure of such an architecture. Incoming messages must be buffered until the receiving task is ready to receive the message, therefore distributed-memory architectures often require a message-passing buffer. Distributing the memory increases the memory bandwidth compared to a shared-memory architecture, as all PE's can access their local memory concurrently. Another advantage is that producer-consumer locality can be exploited, tasks that share data can be scheduled on neighboring PE's. Communication latencies depend on the distance between the sender and receiver and the traffic volume at the time. The application mapping and network architecture are, therefore, two important factors that determine the performance of distributed-memory architectures.

The main disadvantage of distributed memory-architectures is that the application developer is in charge of communication and synchronization, which makes programming for distributed memory architectures difficult.

2.6 Performance analysis and comparison of a Shared-memory architecture and a Message-Passing architecture

Given the advantages and disadvantages of both the shared-memory architecture (SM architecture) and the message-passing architecture (MP architecture) and the charac-

Table 2.1: Memory Latencies

Memory component	Latency in Cycles
L1 Data retrieval	1
L2 Data retrieval	3
L2 Miss penalty r/w	30

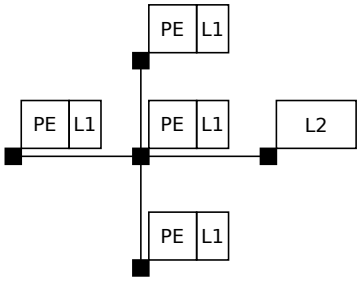


Figure 2.6: The shared-memory configuration used in this simulation

teristics of streaming applications, our preliminary hypothesis is that the MP architecture is better suited for streaming applications. To confirm our hypothesis, a simulation of the execution of a partitioned streaming application on both multiprocessor architectures is run. This gives us an indication of the performance of both archtictures.

2.6.1 Experimental setup

The simulation is run in Matlab, which implements a suitable programming environment for such a simulation. The performance of a multiprocessor is generally measured by the execution time of applications on the multiprocessor. Thus, the goal of this simulation is to attain the execution time of the same application on both architectures. The run-time behavior of the individual components belonging to both architectures are simulated in terms of latency.

The SM architecture configuration used for the simulation, consist of four PE's with a three-level memory hierarchy. Each PE has a private L1-data cache and communicate via a shared L2-data cache. The L2 is connected to an off-chip main memory. The memory-access latencies used for the simulation are listed in 2.1 and Figure 2.6 shows the SM configuration.

The message passing architecture consists of four PE's with a distributed memory, where all cores have their own address space and communicate via message passing. To send and receive messages a message passing buffer is used, Figure 2.7 shows the simulated MP architecture.

A dummy application is used for the simulation. This application is partitioned into three functional parts, all having an equal execution time of 500 cycles and therefore have a perfectly balanced workload. The size of the data blocks in the data stream is 64 bytes. Figure 2.8 gives the dataflow graph of the dummy application.

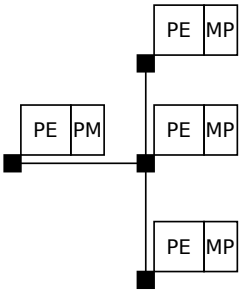


Figure 2.7: The message-passing configuration used for this simulation

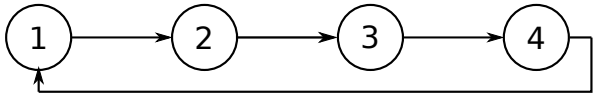


Figure 2.8: The dataflow graph of the simulated application

2.6.1.1 Noc specifications

The same interconnect is used to connect the components in both architectures. In previous work a scalable NOC architecture has been designed and is used for this experiment [7]. The basic building blocks of the NOC are the routers. The task of the routers is to route the packets arriving at the router to the next router on the path to its destination. The router has five in and output ports, each input port has a buffer which is used to buffer incoming packets that need to wait their turn. A round-robin scheme is used to arbitrate between the input ports. The NOC architecture implements wormhole switching, were a packet is broken into smaller pieces called flits. Each packet has a header flit and one or more data flits. The header flit contains information needed by the routers to determine the direction of the packet and also indicates the nature of the contents of the packet. The last flit is called the tail. The routing algorithm routes packets along the shortest path from source to destination. The time it takes for the arbiter to arrive at a particular input port can vary from 1 to 5 cycles, which is determined by the number of input ports. To keep the simulation rather simple, a fixed router forwarding latency is used. The total execution time of the application depends on the individual component latencies, they are listed in Table 2.2.

2.6.1.2 Traffic volume

In reality the traffic conditions on the NOC vary during run-time, this must be taken into account as well. When a packet arrives at a router it either proceeds immediately or must wait for other packets to pass through the router first. The worst possible latency a packet can incur at a router is when four other packets consisting of the maximum amount of flits may proceed first. The number of packets that an incoming packet encounters at the routers, indirectly indicates the traffic volume. In heavy traffic conditions, packets are stalled quite often and for a longer period of time. To model these traffic conditions, incoming packets are entered into slots, each slot indicates the number of cycles the packets must wait, i.e the time it takes for a number of packets,

Table 2.2: NOC specifications

Component	Value
Packetizing	3 cycles
Depacketizing	1 cycle
Router arbitration	4 cycles
Flit size	36 bits
Maximum packet size	16 flits
Buffer size	infinite

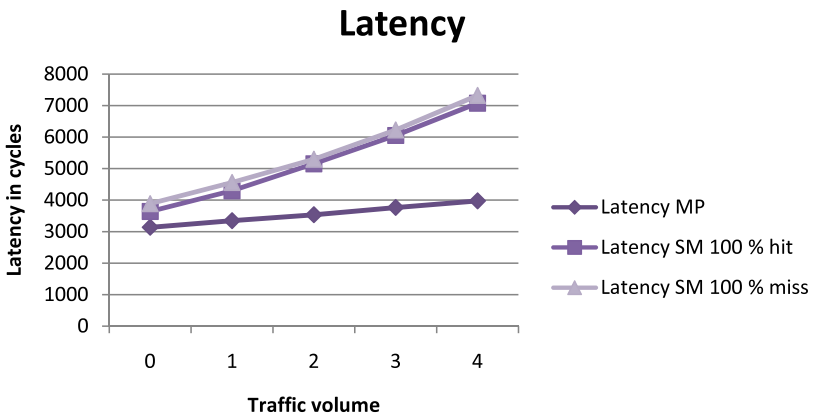


Figure 2.9: The latency for different traffic volumes

at most four packets, to pass through the router first.

2.6.2 Results

The latency and the speedup given as a function of the traffic volume are shown in Figure 2.9 and Figure 2.10, respectively. The speedup is calculated by dividing the latency of the MP architecture by the latency of the SM architecture. The latency of the SM architecture varies between two extremes, the worst latency is when the L2 has a 100% miss rate and all requests are forwarded to the off-chip memory. In the best case, the L2 cache has a 100% hit rate. Increasing the traffic volume, increases the communication latencies and thus the total communication overhead. The latency curve of the SM architecture latency curve is steeper then the MP latency curve, which indicates that the SM architecture is more affected by the extra traffic then the MP architecture. The MP architecture has a better performance, which is expected. The first reason is that on the SM architecture, intermediate data blocks from a pipeline-stage must be written to the L2, the next stage in the pipeline needs this data and requests it from the L2. Data travels first to the L2 and then to the destination core. On the MP architecture data is send directly to the next core in the pipeline, which means that the time spent in the network by the data elements is on average twice as long on the SM architecture. The traffic volume is therefore on average higher for the SM architecture. The second reason is that the cores in the pipeline need to read new data from memory and then subsequently write the intermediate results to memory. Only one read or write request can be processed by the memory at once, this increases the contention for the L2-cache between cores.

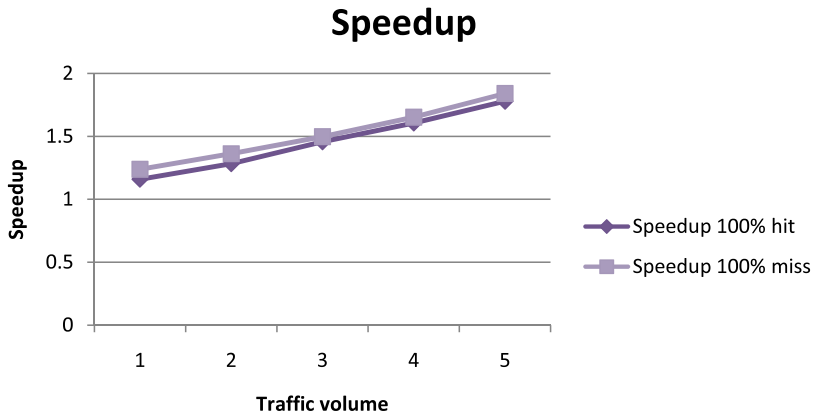


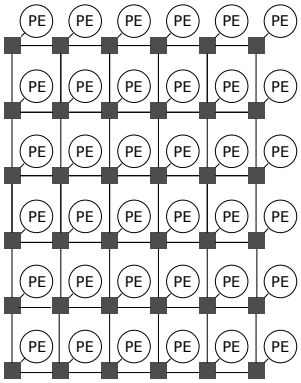
Figure 2.10: The Speedup for different traffic volumes

2.6.3 Scalability

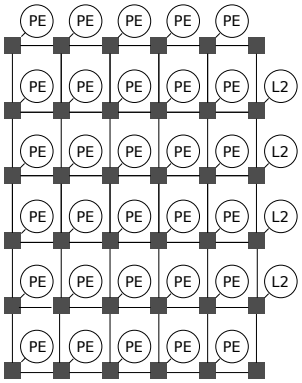
One of the important requirements of many-core architectures is scalability. Both architectures are scaled up and simulated to determine the scaled up performance. The number of PE's used in both scaled up architectures is increased to thirty PE's. Ten pipelines are executed in parallel, where every pipeline consist of three tasks. The L2 in the SM architecture is a banked memory consisting of four banks. The pipeline stages belonging to one pipeline read from and write to a different L2-bank. Figure 2.11a shows the MP configuration and Figure 2.11b shows the SM configuration.

To determine the minimum performance gain (lowerbound) of executing streaming applications on a distributed-memory architecture in comparison to the shared-memory architecture, the pipelines running on the distributed-memory architecture are placed on cores located on average 6 hops away. On an array of 6 by 6 cores, the maximum average hop count is 6 hops. To get an indication of the influence of the hop count on the performance, an average hop count of 5 hops is also simulated. The results of this simulation are given in Figure 2.12 and Figure 2.13. Analyzing the results of this simulation, two important aspects are observed. The first aspect is that the speedup has increased in comparison to the previous simulation. As the number of cores increases so does the number of read and write requests all traveling towards the L2, this further increases the contention for the network resources as well as the L2-cache. The second aspect is that the latency curve of the SM architecture is much steeper then the MP architecture, indicating that the increase in traffic volume has a greater effect on the SM architecture. This is a direct consequence of the fact that the data blocks in the SM architecture spent more time in the network.

Considering the lower latencies and the relatively small influence of the increased traffic volume on the performance of the MP architecture, the MP architecture is proven to be scalable and more suited for streaming applications then the SM architecture. In the MP architecture, traffic is distributed over the entire network, which increases scalability.



(a) The scaled up message-passing configuration



(b) The scaled up shared-memory configuration

Figure 2.11: The mapping of the multiple pipelines on the platform for every application

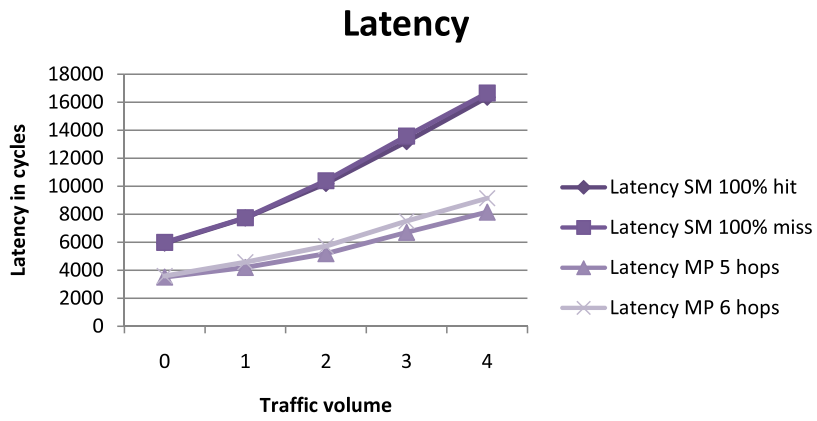


Figure 2.12: The latency for different traffic volumes for the scaled up configurations

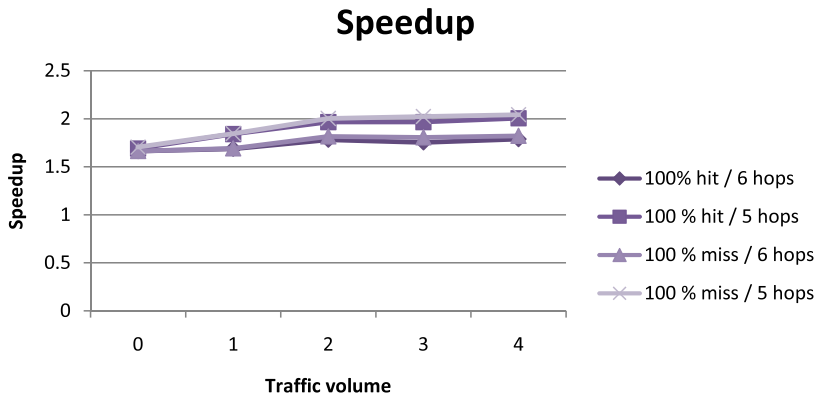


Figure 2.13: The speedup for different traffic volumes for the scaled up configurations

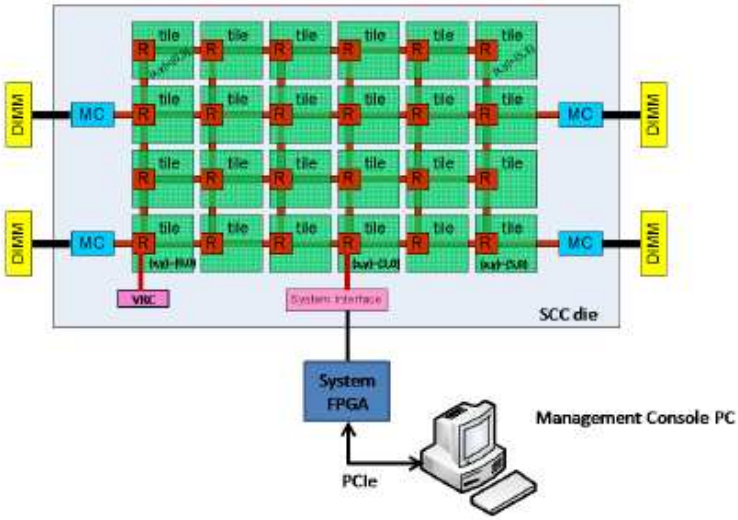


Figure 2.14: Top-level view of the SCC multiprocessor

2.6.4 Conclusion

The results indicate that the MP architecture has a better performance compared to the SM architecture for streaming applications. The speedup in the scaled up architecture has increased, which indicates that the MP architecture scales well. To conclude, the architecture most suited for executing streaming applications is a distributed memory architecture, where the PE's communicate via message passing.

2.7 Related work

There are several message passing architectures. Six architectures with a different Message Passing implementation are discussed in this paragraph.

2.7.1 The Intel SCC

The Intel SCC consists of 48 P54C Pentium cores with an x86 instruction set architecture [8]. The SCC is intended for experimental research and not for commercial use. This experimental research includes exploring the performance of applications on a message-passing architecture. There are 24 tiles, where each tile contains 2 cores. The tiles are connected via a network on chip organized as a mesh. Every core has access to off-chip private memory as well as shared-memory. The SCC does not contain any coherency logic, so the off-chip shared memory must be managed by the application developer. This off-chip memory can be reached through four Memory controllers. Each core has two levels of private caches. The L1-cache is part of the core itself and the L2-cache resides on the tile along with a cache controller. Figure 2.14 shows a top-level view of the SCC.

The SCC uses a Message Passing Buffer (MPB) and a Mesh Interface Unit (MIU) to facilitate message passing between cores. The message passing buffer is shared between

all the cores and is used to store incoming messages. The cores address space is used to address the private memory and all the message passing buffers. Every MPB has its own address space memory mapped into the address space of the core. When a message is sent, data is copied from the L1 to a local or remote MPB. The address of the target MPB appears on the bus and is translated by the MIU into the physical location of the MPB. The MIU uses a LUT for translation. When an address is translated into a remote physical address, the MIU packetizes the data and sends it to the network. The L1-cache contains data from private memory and from the MPB. Every cache line has an extra field that is set when it contains data from the MPB's. When a message is received, the data is copied from either the local or a remote MPB into the L1.

The SCC uses a relatively small message-passing library called RCCE [9]. RCCE implements a shared-name space model where all variables contain one name across all nodes. This enables the application developer to reference a variable by name and core ID. This is an advantage for applications where different cores work on data elements from one large data structure.

The advantage of this architecture is the available off-chip memory, this allows for a broad range of applications to be executed on the architecture without any data memory constraints. However when the processor array will be used for computationally intensive parallelized code, the available off-chip memory is usually not needed.

The main disadvantage of the SCC is that message passing buffer does not contain any logic that manages the messages in the buffer which, therefore, has to be done in software. When a message needs to be sent to a remote MPB, the application developer decides at what address the message will be stored in the MPB. To ensure that messages still residing in the MPB are not overwritten, the application developer must manage the buffer in software. Managing the buffer in software can become very difficult, the application developer must know exactly when space is available and how much.

2.7.2 C-HEAP

C-HEAP is a heterogeneous Multiprocessor architecture template for the design of embedded signal processing systems [10]. With this template the authors address the issue of having to make a trade-off between flexibility and efficiency on one hand and time to market versus cost on the other. The authors propose a flexible and scalable heterogeneous multiprocessor based on a distributed-shared memory architecture. The heterogeneity allows for an optimum balance between performance, power consumption, flexibility and effiency. The memory is distributed among all the processing elements and have one global address space. As these distributed memories are accesable by all PE's they are used as communication buffers. A message is passed by writing to these communication buffers and received by reading from these buffers. Parts of the memory are kept private for scratchpad purposes. Figure 2.15 shows the C-HEAP architecture template. The target applications for this architecture are based on Kahn Process Networks as described in a previous paragraph.

The tasks communicate via FIFO channels. Data producing tasks must claim space in the channel buffer, thus claim empty token buffers and release full buffers. The receiver needs to release empty token buffers and claim full buffers. This is a very efficient way with high communication bandwidth to facilitate message passing. There

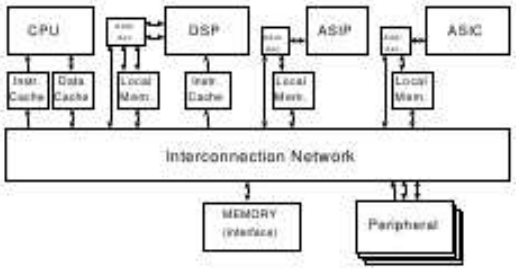


Figure 2.15: The CHEAP architecture

is no competition between communication resources. However these channels must be configured when the hardware is being implemented, which works well for these specific SOC's.

As this is a template for embedded systems, the complete configuration of the architecture including the FIFO channels is customized for one application. This approach is inflexible and not suitable for a more general use of the multiprocessor.

2.7.3 The IBM Cell-processor

The Cell-processor is a power-efficient and high-performance multiprocessor for a wide range of applications [11]. The Cell-processor is a heterogeneous processor that consists of 64-bit IBM Power Processing Element (PPE) and eight Synergistic Processing Units (SPU's). The PPE is the main processor and runs the OS and coordinates the SPU's. A SPU is based on a single-instruction multiple-data (SIMD) architecture and is intended for data-intensive processing. This configuration combines the flexibility of the IBM Power core and the computational power of the SPU's. The ISA's of the PPE and SPU are closely aligned, which increases portability between these cores. The cores are connected by high-speed, memory-coherent Element Interconnect Bus (EIB). Figure 2.16 shows a top-level view of the cell-processor architecture.

The PPE has two levels of memory hierarchy: a 32-KB L1-data cache connected to a 512KB L2-data cache. The SPU's use a local store to store instructions as well as data. Every SPU contains a Memory flow controller that consist of a DMA-controller with which main memory and local store transfers can be done. The DMA-controller also facilitates transfers between the PPE and the SPU's and between two SPU's. One SPU can transfer data from his local store to a remote local store, which is form of message passing. The application developer can initiate DMA-transfers with DMA-instructions.

The EIB consists of one address bus and four 16B-wide data-rings. Two data-rings run in a clock-wise direction while the other two in opposite direction. Each ring can facilitate up to tree data transfers given that the paths don't overlap. When a requestor needs to use the EIB, he makes a request to the EIB arbiter, which arbitrates amongst the requesters in a round-robin fashion. Only the memory-controller has a higher priority to prevent stalling by the requestor of a main memory read request. 16 bytes of data can be send and received every bus cycle via the EIB. The theoretical peak bandwidth can get is high as 204.8GB/s.

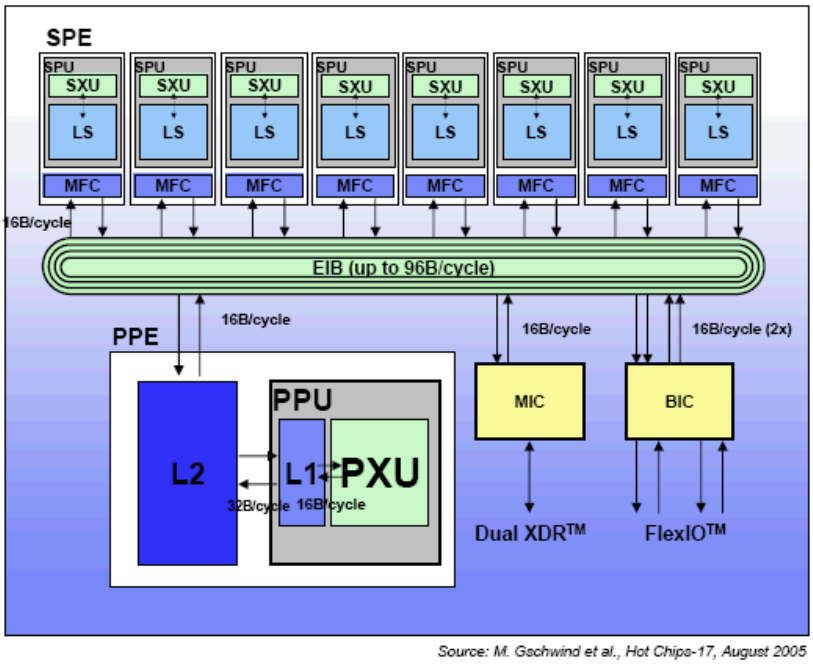


Figure 2.16: The Cell processor architecture

One advantage of this architecture is the potentially high communication bandwidth, as the DMA-controllers can transfer data with a high transfer rate. This type of bus based interconnect performs well with a relatively small amount of processes.

Another advantage is the use of DMA-controllers for data transfers, which can transfer data at a high transfer rate independently and concurrently with the processor.

One limitation of this architecture is the interconnect, which is not scalable as increasing the amount of PE's would increase the size of the rings and the amount of rings. The performance will not scale well with respect to the increase in area.

2.7.4 The PicoChip

The PicoChip is a multicore architecture targeted at digital signal processing applications (DSP) [12]. The picoChip is not intended to be used as general purpose processor but as an alternative to creating an ASIC. In this approach partitioning is done by the application developer and communication is fixed at compile-time, as would be the case for programming an ASIC. Figure shows the PicoChip architecture.

The PicoChip consists of an array of 430 heterogeneous processors. There are four RISC processor variants, the standard processor and three variation of the standard processor. The standard processor is modified by increasing the data memory size and instruction memory size and the addition of an MAC unit. Each processor tile contains a private instruction memory and data-memory and communicate via message passing.

Messages are passed by using DMA-controllers to transfer data from one component to another. The communication is synchronous and can only proceed if both components are ready. The DMA-controllers are instructed by means of special put and get

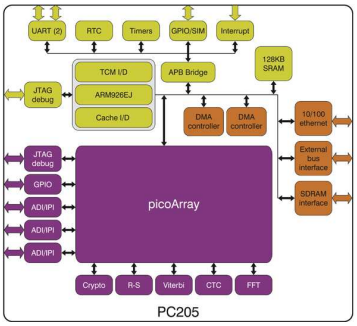


Figure 2.17: The PicoChip architecture

functions. Message buffers are used to buffer incoming and outgoing messages.

The processors are connected through a deterministic interconnection network, which consist of 32-bit unidirectional busses and programmable bus switches. The network uses time division multiplexing, where two communicating cores using the same communication channel are scheduled into time slots. The time slots are scheduled at compile-time.

The main advantage of this architecture is that there is a guaranteed throughput, the communication latency is known at compile-time, as there is no run-time bus arbitration.

The main disadvantage is the fact that the whole array needs to be configured for the a new application, which makes this architecture inflexible.

2.7.5 The Ambric parallel processor

The Ambric multiprocessor targets applications which require fast general-purpose integer computations and digital signal processing [13] , [14]. The processor consists of 360 32-bit RISC processors. There are two processor variants used, simple cores (SR) intended for control flow tasks and complex cores (SRD) intended for computationally intensive tasks. The cores all have access to their private RAM's. The multiprocessor is organized as clusters called brics, half of a bric consists of two SR's and two SRD's. The four cores have access to four local memories each 1kB in size. The memory banks can be shared in several ways, as the software requires.

Communication between brics is facilitated by channels. Each channel consists of a chain of registers, figure shows the structure of an Ambric channel. The channels can be thought of FIFO's between two communicating processors. Each processor contains a special register used as starting points and end points in the channels and can be read from and written to by the processor. The cores are stalled if the channel is full.

The Ambric tool chain includes a graphical interface for a textual language called aStruct. AStruct is used to construct the channels to connect communicating processes. The chip must be configured to facilitate the communication requirements of the target application.

The fact that the communication channels must be configured at compile time, implies that the tasks are statically scheduled on the cores. Run-time information is not taken into account while scheduling, which limits the scheduling possibilities.

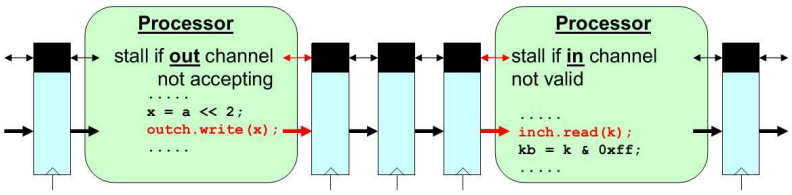


Figure 2.18: The Ambric channel structure

Another disadvantage of configuration at compile time is that the multiprocessor must be configured to fit the needs of each application. This makes this architecture not suitable for a more general multiprocessor.

2.7.6 AsAP

The AsAP [15] architecture consists of small and simple processing elements with globally asynchronous, locally synchronous clocking. These processors are connected through a nearest-neighbor interconnect. AsAP targets applications from the DSP domain and is therefore based on four key characteristics of DSP applications. These characteristics are:

- DSP applications often consist of simple cascaded tasks.
- The tasks require not more then a few hundreds of words of data and instruction memory.
- The need of a processor interconnect methodology that avoids long wires.
- Processing elements must only contain the required resources and no more, the excess of resources consume more power can slows down the PE.

The processors are simple processors containing 64 words of private instruction memory and 128 words of data memory. Each processor tile also contains a ocsilator to change the frequency when required and to clock gate unused processors. The processors communicate via FIFO channels, the processors contain two FIFO channels to receive data from two neighboring processors. The processors can send data to all four neighboring processors. The application developer can access these FIFO's with special variables.

One limitation of this architecture is that a core can only communicate with the four neighboring cores, decreasing the flexibility in partitioning the tasks.

Further more, the only available memory and the chip are small private instruction memories. The fine-grained partitioning limitation might not deliver an optimal performance.

2.8 Summary

This Chapter starts by describing several concepts of parallelism, after which the basic structure of a shared-memory architecture and a distributed-memory architecture are

explained. The advantages and disadvantage with regard to streaming applications for both architectures are listed. To compare the performance of both architectures, a simple simulation is run, which simulates the behavior of the network for both architectures. It is shown that the architecture most suited for streaming applications is a distributed-memory architecture, where the processing elements communicate through message-passing. Furthermore, an overview of several message-passing architectures is provided.

3 System Overview

This Chapter starts with a description of several architectural requirements of streaming applications, after which the complete heterogeneous multiprocessor architecture is explained, which consists of a combination of a shared-memory architecture and a distributed-memory architecture. The remainder of this chapter introduces the architecture of the message-passing tile and the distributed scheduler.

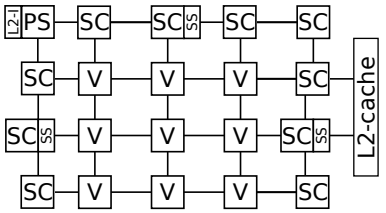
After further exploring streaming applications and the architectural characteristics of the multiprocessors targeted at streaming applications, we have observed the following characteristics typical to streaming applications:

- Streaming applications typically consists of simple, yet computationally intensive tasks which require small data and instruction memories [15].
- Producer-consumer locality, which can be exploited during the process of mapping tasks to cores, can reduce the communication latency between producers and consumers.
- Parallelism, all three types of parallelism (ILP, DLP and TLP) are abundantly present in streaming applications.

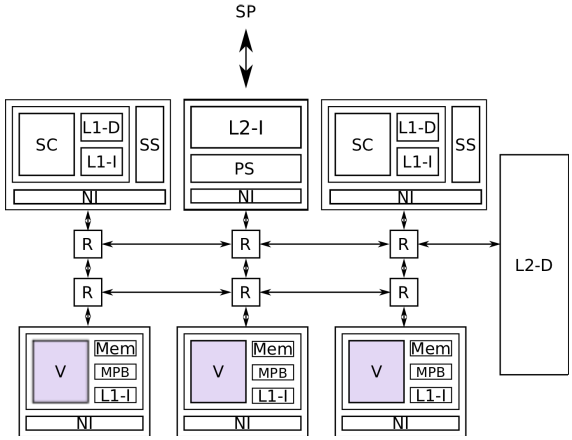
Considering the characteristics of streaming applications, the architecture should contain a dynamic scheduler which exploits producer-consumer locality and a message-passing tile, which provides low communication overhead and contains small scratch-path memories and small instruction memories.

3.1 System Architecture

As mentioned in Chapter 1, one of the main advantages of a shared-memory architecture is that it is easier to program compared to the distributed-memory counter part, as communication and synchronization is implicit and not exposed to the application developer. This makes the shared-memory architecture a more suitable option for a general purpose multiprocessor. However applications can consists of tasks that contain higher levels of ILP and can have streaming characteristics. The simulation results in Chapter 2 show that streaming tasks have an optimal performance on a message-passing architecture. To that end we propose a shared-memory architecture consisting of a large array of multiple simple cores augmented with a smaller array of VLIW processors communicating via message passing. A shared-memory task can spawn the streaming tasks on to the array when the streaming data is available. The complete architecture is shown in Figure 3.1a and consists of a combination of simple cores (SC) using a shared-memory and VLIW cores (V) which have a distributed memory and communicate via



(a) The basic structure of the multiprocessor array



(b) A detailed view of several components of the multiprocessor array

Figure 3.1: The complete multiprocessor architecture configuration

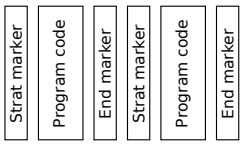


Figure 3.2: Tasks separated by markers.

message passing. The scheduler used for scheduling the multiprocessor tasks is divided into a primary scheduler (PS) and several secondary schedulers (SS). The primary scheduler is in charge of scheduling the shared-memory tasks on the simple cores and the secondary schedulers are used to schedule the streaming tasks on the VLIW array. The several secondary schedulers are placed on one tile next to the simple cores to facilitate fast scheduler access when needed. The tiles are shown in more detail in Figure 3.1b.

This multi-processor is intended to run under a supervising processor (SP), which runs the operating system. When the SP encounters an application intended to run on the multiprocessor, the program code is transfered to the multi-processor where it is stored into the L2-instruction cache, which is part of the primary scheduler. The SP sends the tasks as an instruction stream to the primary schedulers. To indicate the start and end of a task, the tasks contain start and end markers. Figure 3.2 shows the instruction stream transfered by the SP to the primary scheduler, where the tasks are separated by the markers.

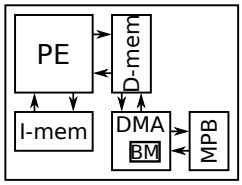


Figure 3.3: The basic components of a message passing tile

3.2 Scheduling Policy for scheduling on the VLIW array

The communication overhead in parallel applications may decrease the overall performance. Communication between tasks belonging to a streaming application has a repetitive nature, which further increases the importance of small communication latencies. The communication latency is partly determined by the physical location of the two communicating tasks i.e. the distance between communicating tasks. To take advantage of the producer-consumer locality of streaming applications, the scheduling algorithm has a nearest neighbor policy, where tasks that frequently communicate are scheduled on neighboring cores.

A data-flow graph (DFG) is used to determine which tasks depend on each other. The secondary schedulers traverse the DFG and the tasks are scheduled on the neighboring cores.

3.3 Overview of the Message Passing Architecture

Existing message passing architectures like the C-HEAP (see Section 2.7.2) and Ambric (see Section 2.7.5) communicate through FIFO channels, which is an efficient and suitable way of transferring data in streaming applications. Data blocks in streaming applications are processed in a certain order, and flow through the pipeline in the same order. This behavior coincides with the characteristics of a FIFO. Every task has its private FIFO channel, which ensures that messages from different tasks are always separate. These communication channels are established by the application developer and must be configured at compile-time. These FIFO channels makes the architecture inflexible and not suitable for our dynamic architecture.

Other message-passing architectures like the Intel SCC (see Section 2.7.1) have a MPB which is used as a RAM and is a part of the global address space of the core. The application developer must specify the exact address in the remote MPB where a message will be placed at, which implies that the buffer addressing must be managed by the application developer. This can severely complicate his job.

Considering the advantages and limitations of the existing architectures we propose the message passing architecture illustrated in Figure 3.3, which includes a combination of several positive aspects of the existing message-passing architectures. The architecture given in Figure 3.3 includes only the basic building blocks of the message-passing architecture. The PE has a private instruction and schratchpad memory. The DMA-controller and the message passing buffer are used for message passing.

The DMA-controller is responsible for transferring data between several components

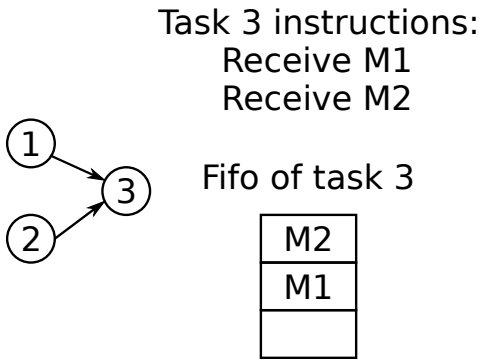


Figure 3.4: The FIFO problem

on the tile. Transfers are initiated by calling specific message-passing functions in the application software. These message-passing functions are defined in a library specifically written for this architecture. The message-passing functions consist of writing to the control registers of the DMA-controller. The control registers are memory mapped into the address space of the PE's. The DMA-controller monitors the data-memory address bus of the core to verify if the control registers are being written to. The control registers contain information that the DMA-controller needs from the software to facilitate a correct data transfer, which include the size of the message, the source and destination of the message and the address of the message in the private memory.

The incoming messages are stored in a message passing buffer which partly behaves like a FIFO and partly like a RAM. The FIFO has the advantage that the application developer does not have to manage the addressing of the buffer, however a problem occurs when a receiving task has multiple senders, as the messages might arrive out of order. Figure 3.4 illustrates this problem, where task 3 must first receive message M1 from task 1 and then message M2 from task 2. The order in which M1 and M2 arrive at the FIFO (buffer) can differ as a consequence of the network, with the result of task 3 receiving the wrong message. This problem is solved by using the buffer manager (BM). The buffer manager sees the buffer as a RAM and records the order in which the messages arrive along with the location of the message in the buffer. When a message needs to be received from a certain source, the buffer manager will provide the oldest message from that source.

To paint a complete picture of all the components involved for passing a message, the process of sending a message on one side and receiving the message on the other side is illustrated in Figure 3.5. The processes consist of a series of steps. The first step involves the application developer initiating a message transfer by calling the send function. The application developer specifies the message size, the destination and the address of the message in private memory. The second step involves writing the message transfer specifics to the control registers of the DMA-controller. The DMA-controller then starts to transfer the message according to the information in the control registers. The DMA-controller transfers the message from the private memory into the network, to the destination core, where the message is tranferend from the network into the message passing buffer of the receiving core. On the receiving side when the task running on core 1 requires the message, the receive function is called and the DMA-

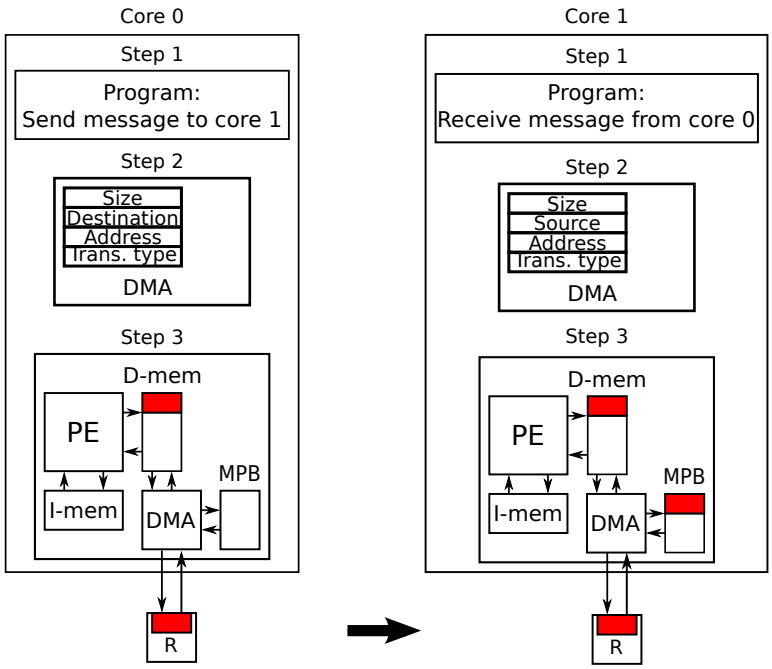


Figure 3.5: The process of sending a message form core 0 to core 1

controller will transfer the message from the message passing buffer to the private memory.

3.4 Summary

This Chapter describtion of the complete multiprocessor array, which consists of a large array of simple cores using a shared-memory and a small array of VLIW cores which communicate via message passing. Furthermore a brief introduction of our message-passing architecture is given, which includes a DMA-controller for data transfers and a MPB to buffer incoming messages.

4 Architecture

This chapter starts with a detailed description of the distributed scheduler. After which the architecture of each component of the message-passing tile is explained in detail.

4.1 The Distributed Scheduler

The Distributed Scheduler is a hardware scheduler that consist of one primary scheduler and several secondary schedulers. The advantage of a hardware scheduler is that run-time information can be taken into account when the tasks are scheduled. For example, degraded or completely broken-down cores can be avoided while scheduling.

4.1.1 The Primary Scheduler

The primary scheduler is responsible for accepting and storing incoming tasks sent by the supervising processor (SP) in a L2-instruction cache. As explained in Chapter 3 the SP sends the tasks as an instruction stream to the primary schedulers. Figure 4.1 shows the architecture of the primary scheduler. The SP asserts the valid bit to indicate that the next instruction is ready to be transfered. The primary scheduler writes the instruction at the given address in the L2-instruction cache.

To keep track of the tasks in the L2-instruction cache, the start address and the end address of the tasks are entered into a table along with the task ID. The start marker also includes a contents field and the task ID. The purpose of the contents field is explained in the next Section. The markers are not entered in the instruction cache along with the program code.

4.1.1.1 Scheduling Policy

To take advantage of the producer-consumer locality of streaming applications the scheduling algorithm has a nearest neighbor policy, where tasks that frequently communicate are scheduled on neighboring cores.

A data-flow graph (DFG) is used to determine which tasks depend on each other. Applications can consist of multiple independent data-flow graphs. The tasks that form each of these independent data-flow graphs are grouped together. Each group can be identified by a unique group number. A group number consist of 8 bits and can therefore support 256 groups. One advantage of such a mapping, for example, when a streaming application also contains data level parallelism, multiple pipelines can execute in parallel, all processing a different part of the data. Each pipeline belongs to a different group, which has its own data flow-graph and is scheduled by a different secondary scheduler.

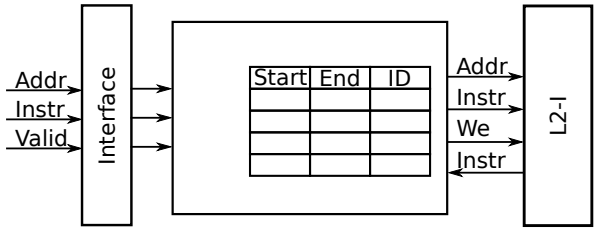


Figure 4.1: The basic structure of the primary scheduler.

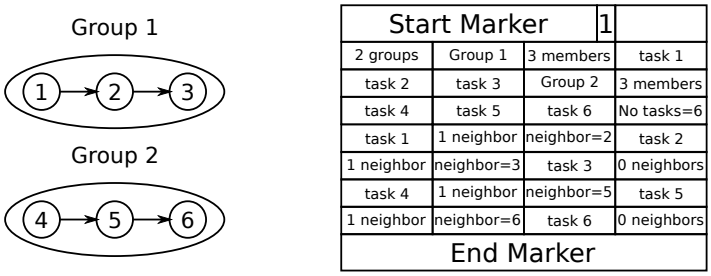


Figure 4.2: Data Flow Graph representation

The SP transfers the data-flow graph as a separate segment with a start and end marker to the primary scheduler along with the tasks. The contents field of the marker is used to indicate if a segment is a task or the data-flow graph. In order to start scheduling while the program code of the tasks is being transfered, which is done to reduce the performance overhead, the data-flow graph is send first.

The data flow graph(s) must be represented in such a way that it is simple to read-out while keeping the size small. An example of the representation of multiple independent DFG's belonging to different groups is illustrated in Figure 4.2. The contents field is set to one, which indicates that this is a DFG. First all the groups with their group members are listed to indicate which task belongs to which group. Next the actual data flow graph is given, where the dependencies of each node are listed.

4.1.1.2 The Secondary Scheduling Policy

As described in the last Section, the primary scheduling policy is the nearest neighbor policy, there may be occasions where there are several neighbors with an equal distance between the core and the neighbors. The secondary scheduling policies determine the direction of the pipelines i.e the direction in which the data flows through the pipelines in the multiprocessor array. This can be taken into account when the array is being configured to avoid traffic traveling in the same direction as the data blocks in the pipelines. For example in the case of the multiprocessor array depicted in Figure 3.1a which partly consist of a shared-memory. The fact that there will be an increase in traffic flowing towards the shared-L2 cache is known a priory, which can be taken into account by chosing the best secondary scheduling policy. There are several policies that can be applied depending on the starting points of the schedulers, which are:

- The SWEN policy, which first chooses the core to south and then if the core to

the south is unavailable the core to the west is chosen, then the core to the east and then at last the core to the north.

- The WSNE policy, which first chooses the core to the west and then the core to the south, moving on to the core to the north and at last the core to the east.

The best policy depends on the starting points of the schedulers, if the starting points are on the north edge, the SWEN policy is the most suitable. The pipeline then starts in the north and moves towards the south. The WSNE policy is most suited for starting points on the west edge. The pipeline then starts in the west and moves towards the east. For our architecture the secondary schedulers use the SWEN policy.

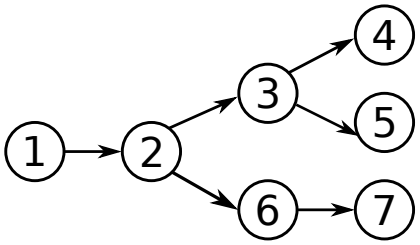
4.1.2 The Secondary Schedulers

The secondary schedulers are responsible for mapping the tasks belonging to one group on a core according to the data flow graph. As there are multiple schedulers on the platform, each group can be scheduled by a different secondary scheduler and can, therefore be scheduled in parallel.

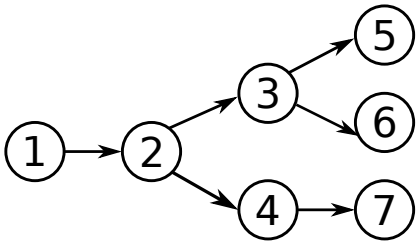
The secondary scheduler schedules tasks on cores by traversing the DFG and scheduling the tasks on the nearest neighbors. The secondary scheduler contains a map of the physical locations of the cores. The map also includes the availability status of the cores. The first task in the group is the head task. This task is the root of the DFG and is the starting point when traversing the DFG. After identifying the head task, the next step is to determine on what core the head task should be scheduled on. There are a few possible starting points. The most advantageous starting point would be somewhere on the edge of the mesh, because the head task has no upstream neighbors. The starting points of the schedulers are generic and can be changed when necessary.

After a task has been assigned to a core, the secondary scheduler notifies the primary scheduler that the program code must be sent to the target core. When the primary scheduler receives such a notification, the core that has been scheduled on is marked as unavailable. The primary scheduler thus has a list of the occupied cores and replies with a positive acknolegment if the target core is indeed available. If the core has already been scheduled on without the secudary schedulers knowledge, the primary scheduler replies with a negative acknowledgment. This situation occurs when two schedulers try to schedule on the same core roughly at the same time. The primary scheduler has a first come first serve policy. After receiving the positive acknoledgement, the scheduler notifies the remaining secondary schedulers that the core is now unavailable, which implies that the availability status is updated every time a core has been scheduled on. In the case of a negative acknowledgement, the secondary scheduler searches for the next available neighbor core.

There are two ways to traverse the graph, which will be explained with the help of Figure 4.3. Figure 4.3 shows the different orders in which the tasks can be scheduled when traversing the graph. The difference is in the order in which the nodes with more then one dependency are traversed. The first way is called depth-first where one whole branch is traversed and then moved on the the next branch. The second way



(a) Depth-first



(b) Breadth-first

Figure 4.3: The different traversing orders

Task ID	Local Tile Number
1	Tile 8

Figure 4.4: The LUT structure

is called breadth-first where one whole level is scheduled first, after which the next level is scheduled. A level is defined as all the nodes with the same depth. The first way has the disadvantage that when there are a limited amount of available cores the first branch will have an optimum mapping while the second branch is left with the remaining nodes, resulting in a suboptimal mapping. The second way does not have this disadvantage and is therefore implemented.

As mentioned earlier, tasks have an ID, which are used to identify the source and destination of a message. As the mapping of tasks to cores is determined during runtime, the physical location of the sending and receiving tasks are not known during development. To successfully send a message to the right destination task, a mapping of the physical locations and the tasks ID's is required. This mapping of the physical locations to the tasks is implemented as a Lookup table (LUT). The structure of the LUT is given in Figure 4.4. Every PE contains a LUT which is used to look up the destinations of the messages. Every tile in the mesh has a local tile number, which is used to identify the tile. The local tile numbers are entered into the LUT.

The secondary scheduler fills in the entries of the LUT's when the tasks are scheduled and have a physical location. The LUT's only contain entries from tasks that are actually communicating, which means that not every task is entered into every LUT.

4.2 The Message Passing Tile

The Message Passing tile architecture consists of several components which are the PE, data-memory, instruction-cache, message-passing buffer, outgoing buffer, DMA-controller and the network interface. Figure 4.5 shows all the components of the tile. The components are described in detail in the following Sections.

4.2.1 The Processing Element

To improve the adaptibilty of the message-passing architecture, it must be processor independent. Processor architecture independence is also needed to support hetero-

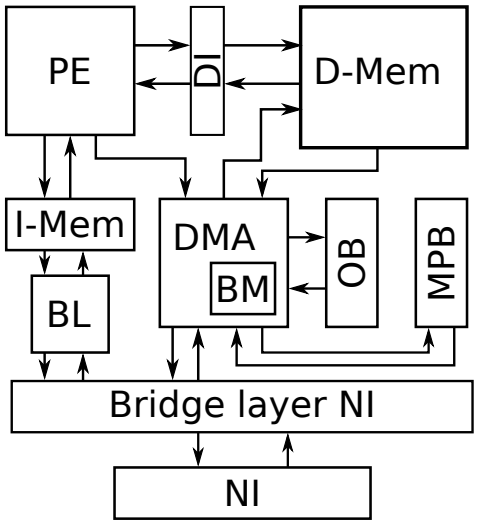


Figure 4.5: The Message Passing Tile

geneity. As such, we support full processor-independability with the limitation that the data-memory width has to be 32-bits wide. This limitation can be removed by supporting multiple data-memory widths, which we leave for future work.

The message-passing architecture is based on the ρ-Vex VLIW processor architecture [16]. The ρ-Vex is a suitable processor for our multiprocessor because it has three advantages:

- Parameterized, which makes this processor very flexible. The processor can be adapted to the application requirements.
- Reconfigurable operations. Custom instruction can be added to the instruction set to increase the performance for specific applications.
- A complete tool chain.

4.2.2 Data Memory

The PE's have a private data memory. This memory will be used for scratchpad purposes like the stack, global variables etc. The size of the data memory depends on the applications data-memory requirements. For streaming applications in particular, the requirements depend on how the application is partitioned and at what granularity the data is streamed. Generally, the computationally intensive parts of the streaming applications do not need a large amount of memory, as the largest data structure of the partition is most likely a single data block from the data stream. A general assumption must be made as to what size will be enough for most of the streaming applications. The data memory size is generic and can be easily modified when needed.

4.2.3 Instruction Cache

The instruction cache is used to store the program code and has the same architectural characteristics as the PE. The width of the cache and the address width are equal to the instruction width and the address width of the core. The instruction memory size must be big enough to completely fit the largest partition of the application. The instruction cache is generic and can be modified accordingly.

4.2.4 The Bootloader

The bootloader's main task is to load the incoming instructions send by the primary scheduler into the instruction cache according to the issue-width of the core, which can vary with a parameterized VLIW core or different VLIW cores. The instructions arrive as 32-bit words and must be combined to form an instruction with the correct issue-width. The bootloader keeps track of the addresses at which incoming instructions must be placed in the instruction cache.

4.2.5 Data Interface

The primary function of the Data Interface is to transfer the read and write requests with an address within the address range of the data-memory to the data-memory. Writes to the control registers are kept from reaching the data-memory.

As mentioned in Section 4.2.1 the message-passing architecture is processor independent, the data interface is also used to make the data-memory architecture compatible with the data-memory architecture of the PE.

4.2.6 The Message Passing Buffer

The message passing buffer (MPB) is used to buffer incoming messages. When the core is ready to receive a message it is transferred from the MPB to the data memory. The core is stalled when data is being transferred between the MPB and the data memory, to prevent the core from reading data that has not been transferred yet.

The size of the MPB must be big enough to hold at least one complete message, because a message can only be received when the complete message has been arrived. The message size depends on the size of the individual blocks in the data stream. An experiment described in Chapter 6 shows the independence of the size of the MPB on the performance. For optimal performance the MPB size must be big enough to fit all the data needed for one iteration, further increasing the size does not improve the performance.

4.2.7 The Buffer Manager

As explained in Section 3.3, the buffer manager is used to manage incoming messages in the massage passing buffer and provides the oldest message from the specified source. The buffer manager (BM) keeps track of the start and end address of the messages within the buffer along with the source of the message. When a message must be received, the core specifies the source of the message and the oldest message from that

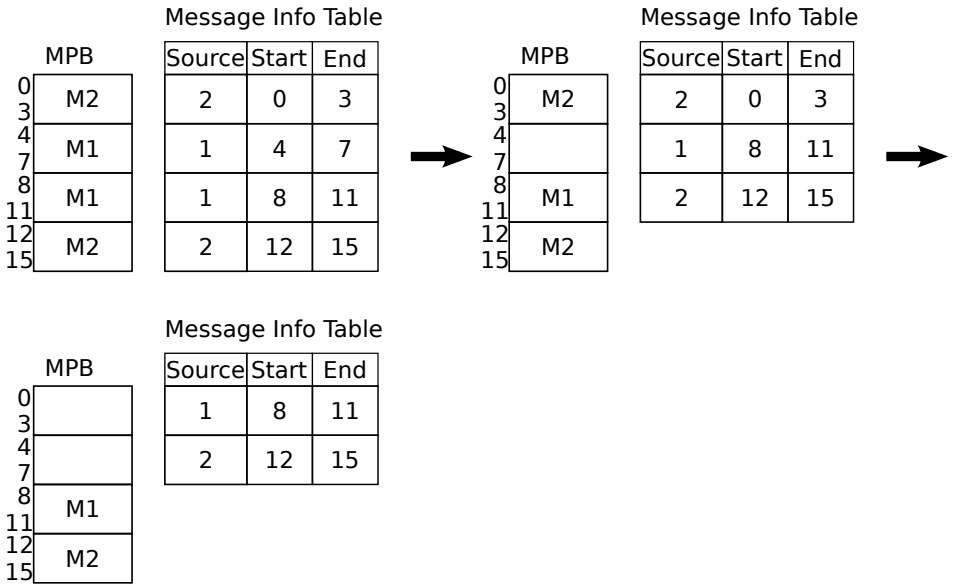


Figure 4.6: The behavior of the Buffer Manager

source is received. The behavior of the buffer manager is explained with the help of an example which is illustrated in Figure 4.6. In this example a message from source one must be received. To keep the example simple, the MPB can hold a maximum of four messages and the message size of all the messages in the buffer is four words. The start and end addresses of these messages within the MPB have been entered into the message information table in the order of the arrival times. To receive a message from source one, the BM searches the message information table and selects the oldest messages from source one, which is in this example the second entry in the message information table and starts at address four in the MPB. The next message that needs to be received is from source two, which is the first message in the message information table. After transferring the message, the message information table is updated and the oldest unreceived message is again the first entry in the message information table.

4.2.8 Outgoing Buffer

The outgoing buffer (OB) is used to buffer outgoing messages. The core is stalled when a message is being transfered to the OB, to keep the core from overwriting data that has not been transfered yet. The main advantage of the OB is that the core is not needlessly stalled by the flow control procedures which will be explained in Section 4.2.9.1, only by a full downstream buffer.

As soon as the message has been transfered to the outgoing buffer the core can resume execution. The OB is a FIFO, which means that the messages are sent out in the order of which they have been generated. A full buffer indicates that the MPB of the destination task of the oldest message is full.

The size of the OB must be big enough to fit at least one complete message, to take advantage of the fact that the core can resume execution immediately after the complete message is transfered to the OB. If the OB is too small, it can only store part

of the message, the core will stall until the last part of the message is transfered to the OB. This would mean that the core remains stalled for the time it takes to actually send the message into the network, i.e. the advantage of using the OB would be lost.

4.2.9 The DMA Controller

The DMA controller is in charge of transferring data between different components. There are four types of transfers supported by the DMA controller. The four types of transfers are:

- *Send*, where data is transfered from the data memory into the OB, after which it is transfered to the network.
- *Receive*, where data is transfered from the MPB to the data memory.
- *Read* from a global data buffer. This global buffer resides on a tile on the chip and is used to store data from the input data stream and buffer result data. The input data in the global data buffer can be placed there either by the SP or an external device. For the experiments in this thesis the data in the global data buffer is already in the buffer upon receiving the program code by the primary scheduler. A read from the global data buffer consist of sending a read request to the global data buffer and then, when the read data has arrived, transferring the read data to the data memory.
- *Write* to the global data buffer, where data from private memory is transfered into the network. Writes to the global data buffer are made when the last stage in the pipeline is finished processing the data blocks. The data blocks containing the result are buffered in the global data buffer and can be transfered back to the SP or the external device.

Transfers are initiated by calling specific message-passing functions in the application software. These message-passing functions are defined in a library specifically written for this architecture. The message-passing functions consist of writing to the control registers of the DMA-controller. Implementations of the two basic message-passing functions are presented in Listing 4.1. The control registers are memory mapped into the address space of the PE's. The DMA-controller monitors the data-memory address bus of the core to verify if the control registers are being written to. The control registers contain information that the DMA-controller needs from the software to facilitate a correct data transfer. There are four control registers, which are listed in Table 4.1. The control registers can contain different information depending on the function that has initiated the transfer.

Listing 4.1: Two basic message-passing functions

```
1 void MP_send(int Destination,int length_message, void *message){
      CONTROL_REG_1 = (int)message;
      CONTROL_REG_2 = length_message;
      CONTROL_REG_3 = Destination;
      CONTROL_REG_4 = SEND_CODE;
```

Table 4.1: Control Registers

Register	Description
Register 1	The starting address of the data in data memory for a send / The source task ID for a receive
Register 2	The size of the message in number of bytes
Register 3	The destination task ID / The starting address
Register 4	The transfer ID, which is used to identify the different types of transfers listed above

```
}
void MP_receive(int source,int length_message, void *message){
    CONTROL_REG_1 = source;
    CONTROL_REG_2 = length_message;
    CONTROL_REG_3 = (int)message;
    CONTROL_REG_4 = RECEIVE_CODE;
}
```

4.2.9.1 Flow Control

To ensure that data is not sent to full MPB's, there must be some form of flow control to verify if the receiving MPB has enough space to store a message. Without the presence of flow control, messages might be lost or the network will get saturated with messages that can not be received.

When a message needs to be sent, a MPB space request which consists of the size of the message is sent to the destination core first. The destination core replies with a positive acknowledgement if there is enough space available. In the case of a full buffer the receiver remains silent until the MPB is completely drained. A negative acknowledgement would be redundant because the sender would still have to wait for the positive acknowledgement.

If there are multiple tiles that need to send a message to the same tile, the receiving tile will receive multiple MPB space requests. Only the first sender gets permission to send his message, the other message requests are buffered in the order in which they arrive. When the message from the previously permitted sender has completely arrived, the sender from the next buffered MPB space request gets permission to send his message. This implementation avoids the need to keep track of the status of multiple incompletely received messages in the buffer, which increases the implementation costs. This implementation has the disadvantage that the performance might decrease as a consequence of serializing message transfers from multiple senders to one receiver. However this decrease in performance is not applicable in every situation, which will be explained with the example illustrated in figure. In this example core A and core B have received permission to send their message to core C. The cores start to transfer their messages which consist of three packets. The packets move in the same direction towards the sender. As explained in Section 2.6.1.1 the arbiter of the routers arbitrates between inputs in a round-robin fashion. As a consequence of the round-robin router arbritation the packets will arrive interleaved. The message can only be received if all his packets have arrived and in this example the message from A can only be received

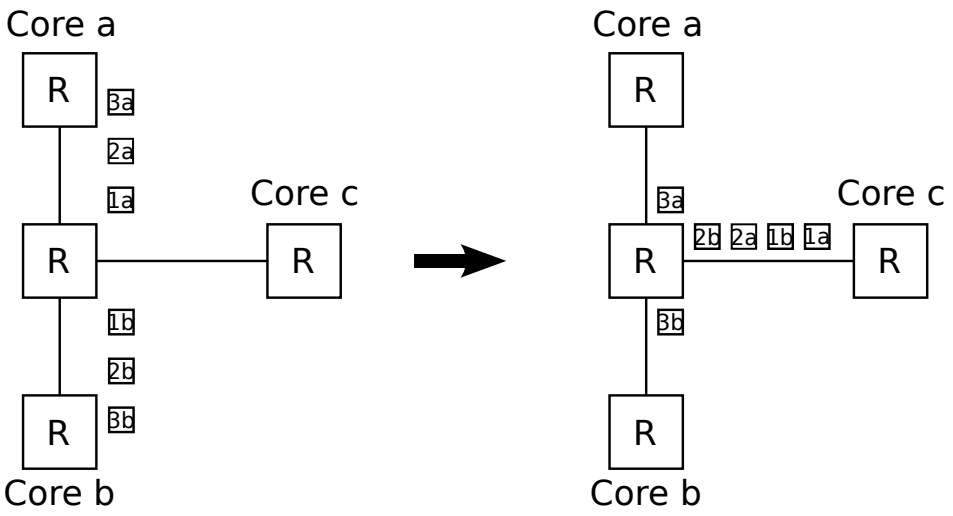


Figure 4.7: Illustration of the disadvantage for multiple message send approvals

after the arrival of the first two packets of the message from B. From the example we can conclude that approving multiple send requests also has its disadvantage.

4.2.9.2 The operational states of the DMA-controller

The DMA-controller is in charge of multiple operations, which are listed in table 4.2. These operations all consists of data transfers to and from the network. There exist situations where multiple components on the tile simultaneously have data transfer requests to the network, as there is only one physical channel for data transfers to the network, only one request can be handled at once. The remaining requests must be stalled. To cope with these simultaneous data transfer requests, the DMA-controller maintains a task buffer, where new request are entered in the order of which they are generated. The DMA-controller then handles the requests in that order. These requests in the task buffer are also referred to as tasks. Operations that include accepting data from the network have a high priority and are handled instantly, which prevents stalling the network.

The DMA-controller consists of several states. The controller remains in a waiting state when there are no new tasks in the task buffer. Every operation in Table 4.2 is handled in a different state. When a new task is available, the DMA-controller transitions into the appropriate state, where the task is executed. When the task is completed the DMA-controller transitions back to the waiting state, where the next task in the task buffer is selected. The FSM is given in Figure 4.8, the states are described in Table 4.3.

4.2.10 The Network Interface

The Network Interface (NI) connects the tile to the network. The NI is part of the Network and is responsible for packetizing and buffering outgoing data and depacketizing incoming data. A packet consist of a header flit and one or more payload flits. The

Table 4.2: DMA-controller operations

Request	Operation
A send request by the software	Data transfer between data memory and the OB and send MPB space request to remote PE
A receive request by the software	Data transfer between MPB and data memory
A read request for the global buffer	Send read request to global buffer
A write request for the global buffer	Data transfer from data memory to the network
A MPB space request	Send acknowledgement if the MPB has enough free space, if the buffer is full the request is put into a request buffer
A positive acknowledgement as a response to a MPB space request	Data transfer from OB to the network
Empty buffer requests	Send acknowledgments to all open requests until the buffer is full again
Message arrival	Data transfer from the network to the MPB
Global data buffer data arrival	Data transfer from the network to the data memory

Table 4.3: The FSM states

State	Description
S0	Wait state
S1	Send MPB request
S2	Data transfer to OB
S3	Data transfer from MPB to data memory
S4	Send write data to the global data buffer
S5	Send read request to global data buffer
S6	Send acknowledgement
S7	Send acknowledgments as response to the accumulated MPB state requests while the MPB was full

header contains several fields that are used to route the packet and indicate the nature of the contents of the packet. Figure 4.9 shows the structure of a packet. The first bit in the header is used to indicate that a flit is a header flit or a payload flit. The next bit is the parity bit and is used to discard duplicated flits. Two subsequent flits always have an opposite parity bit. The header flit further consists of the destination field, the source field, the task ID field and the communication ID field. The destination field consists of the coordinates of the destination tile in the mesh, both coordinates are represented by three bits. The maximum mesh dimensions for three bit coordinates is 8 x 8.

The source field contains the local tile number of the source tile, which is the main tile identifier. The local tile number is used to determine the function of the tile and the coordinates of the tile in the mesh. A translation table is used to translate the

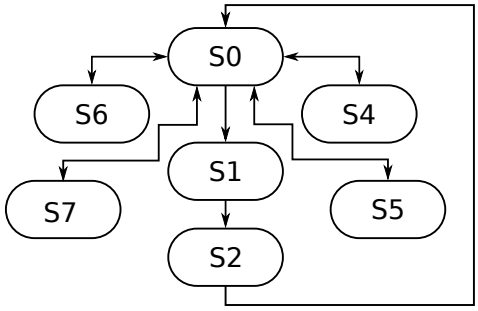


Figure 4.8: The FSM of the DMA-controller

Table 4.4: The communication classes

Class	Tile function
000	PE tile (PE)
001	PE in combination with a secondary scheduler (PSS)
010	Global data buffer (GB)
011	Primary scheduler (PS)

local tile number into the tile function called the tile class and the coordinates in the mesh. There are four classes, which are listed in table 4.4. To completely understand the packet contents both the class of the source tile and the communication ID are required. The communication ID is defined as the type of communication between two classes.

The task ID field contains the task ID of the sending task, which is required to succefully receive a message. The last bit of the header is always zero and is used to indicate the last flit in the packet.

4.2.10.1 Communication types

The communication ID is used to indicate the communication type between two communicating tiles and depends on the class (See Section 4.2.10) of the two communicating tiles. The same communication ID can have a different meaning depending on the class of the source tile. Communication between two PE's consists of data messages and flow control messages. The flow control messages consists of MPB space requests and MPB space acknowledgments. Table 4.5 lists the communication ID's for the different messages between the different classes.

We now briefly describe the communication between the various classes. Communication between the PE's and the primary scheduler (PS) only consists of transferring the program code to the PE's. Communication between the PE's and the secondary scheduler (PSS) consist of sending the LUT entries to the PE's. The LUT entries are entered into the LUT, which is used to map the physical location of the tile to the task ID. After finishing execution, the PE notifies the secondary scheduler that he has finished execution. Communication between the secondary scheduler and the primary scheduler consists of assigning the groups to the secondary schedulers. When the tasks of the groups are scheduled the secondary schedulers notify the primary scheduler that

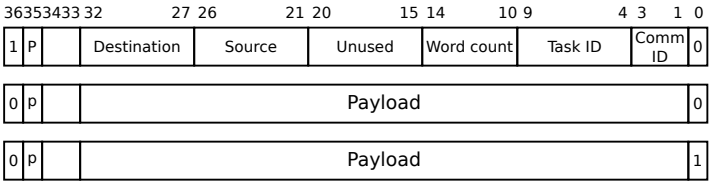


Figure 4.9: The structure of a packet

Table 4.5: The communication ID's and their descriptions

Communication ID	Description
PE-PE Communication	
000	Message
010	MPB space request
100	MPB space request reply
PE-PS Communication	
000	Program code
PSS-PSS Communication	
000	Message
010	MPB space request
100	MPB space request reply
011	LUT entries
101	Core occupancy
110	Finished execution
PS-SS Communication	
000	Group assignments
001	Scheduled task
PE/PSS-GB Communication	
000	GB read request
001	GB write request

the program code of the scheduled tasks must be send to the cores. Communication between the secondary schedulers consist of notifying the other secondary schedulers that a core has been scheduled on and is not available anymore.

4.2.11 The Network Interface Bridge

The network interface bridge (NIB) supplies the NI with the correct header information and the payload data. The NIB converts the communication requests of the tile into the correct communication ID. After the specific header information is given, the NIB transfers the payload data to the NI. The NI transforms the header information and payload data into a complete packet and sends it out into the network.

4.3 Summary

In this Chapter we have described all the components of the architecture in detail, starting with the distributed scheduler. The distributed scheduler consists of the primary scheduler and several secondary schedulers. The primary scheduler accepts the tasks sent by the supervising processor and stores the program code in the L2-cache. The secondary schedulers schedule the streaming tasks on to the array of VLIW processors according to the DFG.

The message-passing tile consist of several components, the most important components include the DMA-controller, the message-passing buffer, the buffer manager and the network interface. The DMA-controller is responsible for the data transfer between several component on the tile. The DMA-controller maintains a task buffer, where unanswered data transfers requests are buffered. Every data transfer request is executed by a different operational state of the DMA-controller.

The message-passing buffer is used to buffer incoming messages. When the receiving core is ready to receive the message, the message is transfered from the MPB into the private memory. Message in the MPB are managed by the buffer manager, which keeps track of the order in which messages arrive and provides the oldest message from the specified source.

The network interface connects the tile to the network, where the packets are packetized and depacketized. Each tile is classified into a class according to the function of the tile. To understand the contents of a packet the class and the communication ID are required.

Experimental Setup and Results

5

This Chapter describes the complete evaluation process. First, our baseline (hardware) platform is described, after which the partitioning strategies of the three selected applications are explained. Next, several evaluation metrics used for this evaluation are defined. Finally, the experiments evaluating different aspects of our architecture are explained and the results are analyzed, from which several conclusions are drawn.

5.1 The Baseline Platform

The baseline platform consist of an array of VLIW processors, which in this case are 20 ρ-Vex processors [16], a primary scheduler (PS), 4 secondary schedulers and a global data buffer (GB). Each secondary scheduler is placed next to a ρ-Vex processor on one tile (PSS). The several components are connected through a NoC in a 4 by 5 mesh. Figure 5.1 shows the baseline platform configuration, where the tiles and their functions in the mesh are illustrated. The baseline platform is slightly modified for two of the performance evaluation experiments as will be described in Section 5.5.5. The design is clocked at 100MHz.

The starting points of the schedulers are on the north edge, resulting in pipelines being scheduled from north to south with the SEWN secondary scheduling policy.

5.2 The Applications

The applications chosen for our performance evaluation are streaming applications, which contain different partitioning characteristics, data stream granularity and flexibility. The term flexibility is used as a measurement of how flexible the application is in varying the number of partitions and varying the data stream granularity. The flexibility is needed to determine the performance for different partitioning strategies for one application. For example, determining the effects on the performance when the number of stages in the pipeline are varied. Three applications are selected to run on our platform:

- The JPEG decoder
- The FIR filter
- A Custom application, which represents a well partitioned application with a good computation to communication ratio.

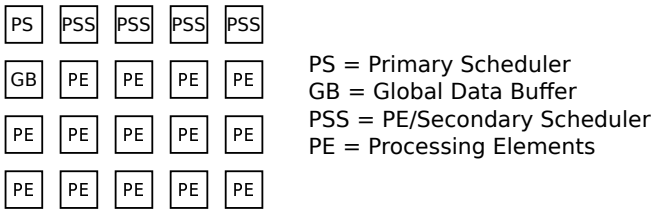


Figure 5.1: The baseline platform configuration

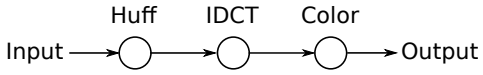


Figure 5.2: The data flow graph of the JPEG decoder

5.2.1 The JPEG Decoder

The JPEG decoder [17] is part of the Mibench [18] benchmark suit and implements the well-known JPEG algorithm. The algorithm consists of an encoder and a decoder. The JPEG encoder compresses bitmap images (BMP) into a JPEG image without the loss of useful information and the JPEG decoder converts a JPEG image back into a BMP image. JPEG images contain a header where information specific to the image is listed, which is used during the decoding process.

The JPEG decoder consists of four well defined stages: the Huffman stage, the IDCT stage, the color conversion stage and the reordering stage. The Huffman, IDCT and color conversion stage are computationally intensive tasks and produce data blocks which are part of the decoded BMP image. The reordering stage is responsible for copying these data blocks into a BMP-image frame. Figure 5.2 shows the data flow graph containing the three computationally intensive stages. The reordering stage is ommited during the performance evaluation experiments because the array of VLIW processors will mainly be used for computationally intensive tasks. The data blocks which contain the results from the color conversion stage are buffered in the global data buffer, where the data blocks can either be reordered by the SP or an external device. In a later stage in the project when the combined shared-memory and distributed-memory architecture, as described in Chapter 3 is realized, the data blocks can be reordered by a shared-memory core. The color conversion stage has a different behavior as the remaining two stages, the color conversion stage requires six blocks from the IDCT stage to start processing.

The granularity of the JPEG decoder is also well defined. A BMP image is divided into a grid of 8x8 pixels for every color component red, green and blue. The grid of 8x8 pixels is the smallest unit of data in the JPEG algorithm. Each 8x8 grid of pixels, which has a size of 64 bytes, is encoded into a block with a variable block size. The input block size of the JPEG decoder is therefore also variable. The size of an output data block of the color conversion stage consists of four data blocks for every color component and has a size of 764 bytes.

During the execution of the partitioned JPEG application, we have observed that the workload of the three stages is completely unbalanced. This can be observed in Table 5.1, where the run-times of the partitions are listed. The run-times are

Table 5.1: The run-times of the three partitions of the JPEG decoder

Stage	**Run-time (ns)**
Huffman	20820
IDCT	138660
Color	60290
Total Parallel	219770
serial	219920

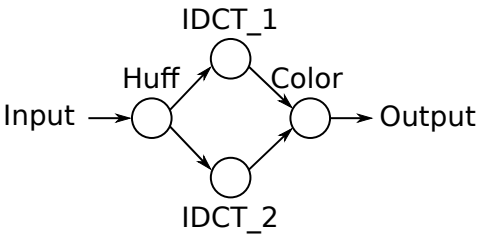


Figure 5.3: The data flow graph of the JPEG decoder with four stages

measured from the moment that the input block has been received by a certain stage until the point at which the data block is about to be send out to the next stage. The time spent on data transfers is not included in these run-times. Let it be noted here that normally the length of the Huffman stage varies depending on the input data. However, to effectively analyze our results, the run-time of the Huffman stage is kept constant, which is realized by using a plain image.

Table 5.1 shows that the stage with the longest run-time is more than five times as long as the stage with the shortest run-time. In pipelined applications the total execution time is determined by the stage with the longest run-time, which we refer to as the critical stage. To even out the workload, the workload of the IDCT stage is divided over two stages. The data flow graph of the new pipeline is given in Figure 5.3. The Huffman stage sends the even numbered data blocks to the first ICDT stage and the uneven numbered blocks to the second ICDT stage. The effect of adding a second ICDT stage is illustrated in Figure 5.4.

The instruction counts of the partitioned and the serial JPEG implementations, generated by compiling the application for the ρ-Vex, are given in Table 5.2. The instruction count of the partitions together approaches the instruction count of the serial implementation. This indicates that this application consists of three distinct phases. The extra instructions in the partitions are used for the communication instructions.

The fact that the partitions and the granularity of the JPEG decoder data blocks are well defined makes this application relatively inflexible.

5.2.2 The FIR Filter

The FIR filter [19] is also part of the MiBench benchmark suite and is an algorithm commonly used in digital signal processing applications. The basic function of a FIR filter is to filter out unwanted noise in a signal. There are several types of FIR filters,

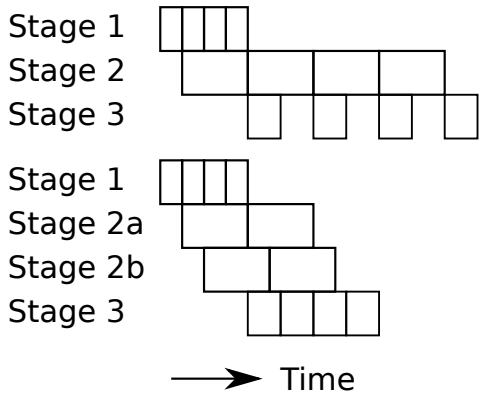


Figure 5.4: The affect of adding a fourth core to a pipeline with an unbalanced workload

Table 5.2: Instruction counts of the JPEG encoder

Stage	**Number of instructions**
Huffman	2017
IDCT	1773
Color	250
Total Parallel	4040
Serial	3706

such as low-pass filters, high-pass filters, moving-average filters etc. For our evaluation we chose the Moving Average filter, because it is relatively easy to understand and partition and also contains DLP. The moving average filter calculates the average of a input signal based on the last N values. We refer to N as the window size. Increasing the window size, increases the precision of the filter. The Moving Average filter is implemented based on the equation given below, where x and y are the input and output signal respectively.

$$y[i] = \frac{1}{N}x[i] + \frac{1}{N}x[i-1] + \frac{1}{N}x[i-2] + ... + \frac{1}{N}x[i-N-1]$$

The baseline partitioning of this application consists of three partitions with more or less equal workloads. It is, however, possible to vary the number of partitions, while maintaining this balanced workload. The optimum number of partitions is observed in the experiment described in Section 5.5.4. Figure 5.5 and Figure 5.6 illustrates how the moving average filter is partitioned. The serial task in Figure 5.5 shows how an input signal x consisting of six values is transformed into the output signal y. Figure 5.6 shows how the serial task is partitioned in three tasks. The data-flow graph of the FIR filter is shown in Figure 5.7.

The filter contains two parameters that must be determined in order to partition the application, which are the window size and the input stream granularity. The window size depends on the required precision of the filter and the input stream granularity, i.e the size of the input data blocks, partly depends on the required performance and partly on the window size. There exist a certain block size which results in the best performance. The block size is partly determined by the window size, because al values in the window need to be known to calculate the moving average.

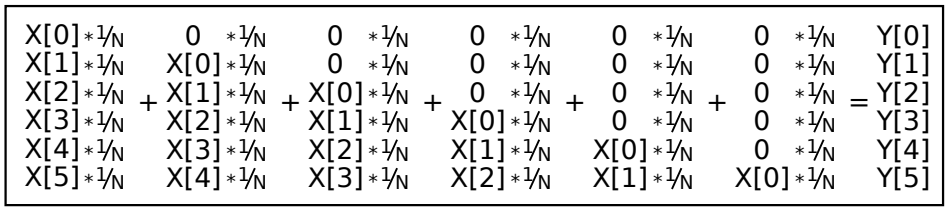


Figure 5.5: The serial moving average filter implementation

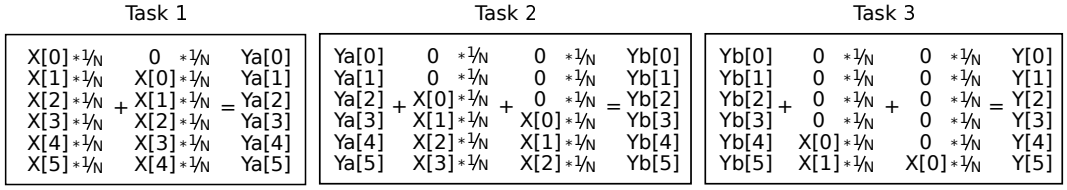


Figure 5.6: The partitioned moving average filter implementation

The window size used for the experiments is 90, which is divided over three partitions, resulting in a window size of 30 for each partition. The window size is kept high to ensure that every partition spends enough time on calculating the result in comparison to the time spent on communication, i.e to keep the communication-time percentage small. The input signal is divided into blocks of 94 signal values.

The run-times of the three partitions and the serial implementation are given in Table 5.3. The run-times are measured from the moment that the input block has been received by the stage until the point at which the data block is going to be send out to the Outgoing buffer. One observation in the table is that the added run-times of the partitions is considerably higher then the run-time of the serial implementation. Normally the extra time is spent on communication, however in this case the extra time is also spent on inner and outer loop transitions. This will be explained with an example illustrated in Figure 5.8. The task in Figure 5.8 consists of two loops, which is partitioned in two partitions. Each partition consists of the same amount of outer loop iterations as the serial task, the difference is in the number of inner loop iterations. The number of inner to outer loop transitions is twice as much as the serial implementation. A transition to the outer loop comes with the retrieval of the outer loop variables. This limits the speedup that can be achieved with this partitioning strategy.

This can also be observed in the program code sizes given in Table 5.4. The size of the serial version and the partitions have more or less an equal size.

5.2.3 Custom Application

The custom application represents a well partitioned application with a balanced workload and a good communication to computation ratio. The custom application exposes the capabilities of the message-passing architecture. The custom application consist of four loops with an equal number of iterations and number of calculations. This application is partitioned into four partitions, which all have an equal workload. The data block size is 256 bytes. The data-flow graph is shown in Figure 5.9. The run-times of the four partitions and the serial implementation are given in Table 5.3. The run-times of the partitions are more or less equal, indicating a balanced workload.

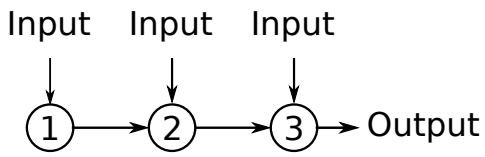


Figure 5.7: The data-flow graph of the FIR filter

Table 5.3: The run-times of the FIR filter

Stage	**Run-time (ns)**
Partition 1	71180
Partition 2	73610
Partition 3	79760
Total Parallel	224550
Serial	185020

The instruction counts of the partitions are listed in Table 5.6. The added instruction counts of the partitions is larger then the instruction count of the serial implementation. The extra instructions of the partitions are used for the communication instructions.

5.2.4 The Scheduling Overhead

The scheduling overhead which consist of sending the program code to the L2-instuction cache, whereafter the tasks are scheduled based on the neirest neighbors and finally the program code is send to the target core. The scheduling overhead is given in Table 5.7 where the scheduling time of the first task in the pipeline and the scheduling time of the last stage in the pipeline are given as the percentage of the total time.

From the results given in Table 5.7 we can conclude that the scheduling overhead only marginally affects the total execution time and thus marginally effects the performance of the applications.

5.3 Performance Evaluation Metrics

In order to evaluate the performance of our multiprocessor, we need to establish the performance metrics. The speedup and throughput are two very common metrics used to evaluate the performance of multiprocessors. The speedup determines the run-time advantage gained by increasing the available resources, throughput determines the rate at which data is processed.

Streaming applications can have timing constraints, which need to be met. An example of such an application is the MPEG decoder which must produce the video frames at a certain rate to ensure an uninterrupted playback. One unpredictable component of the architecture, which can influence the execution time, is the traffic volume in the NoC. Therefore it is important to evaluate the effects of the traffic volume on the execution of the applications. We have defined two different metrics for the traffic volume evaluation, which are block arrival rate and the average deviation of the average block arrival rate.

The performance evaluation metrics and their definitions are:

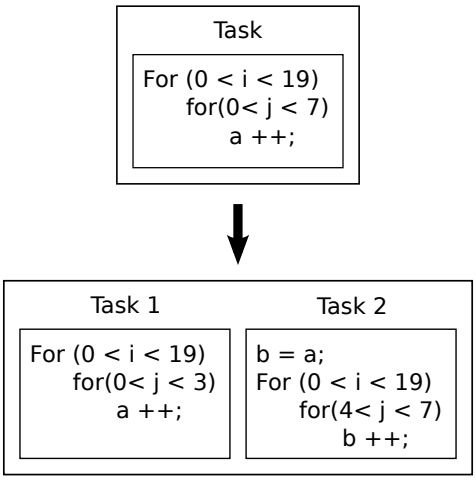


Figure 5.8: The inner and outer loop transitions

Table 5.4: Instruction counts of the FIR filter

Stage	**Instruction count**
Partition 1	92
Partition 2	97
Partition 3	97
Total Parallel	286
Serial	90

- Speedup (S), which is calculated as:

$$S = \frac{E_{Serial}}{E_{Parallel}} \tag{5.1}$$

  Where E_{Serial} is the execution time of the serial implementation and $E_{Parallel}$ is execution time of the parallel implementation. The execution time is measured from the moment the data-flow graph is being transfered to the primary scheduler until the last core has finished execution.

- Average arrival rate is defined as the average time between two subsequent data blocks arriving at the global data buffer. The average arrival rate (R) is calculated as:

$$R = \frac{\sum_{i=0}^{n-1} T_{i+1} - T_i}{n} \tag{5.2}$$

  Where T_i and T_{i+1} are the arrival times at the global data buffer of two subsequent data blocks and n is the total number of blocks.

- Average deviation (D) from the average arrival rate is calculated as:

$$D = \frac{\sum_{i=0}^{n-1} |T_i - R|}{n} \tag{5.3}$$

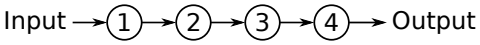


Figure 5.9: The data-flow graph of the custom application

Table 5.5: The run-times of the three partitions of the Custom Application

Stage	**Run-time (ns)**
Partition 1	51860
Partition 2	51850
Partition 3	51850
Partition 4	51980
Total Partition	207540
Serial	207090

In experiments where there are multiple pipelines the average deviation is calculated for every pipeline separately.

- Throughput(TP), which is defined as the average rate at which the resulted data blocks arrive at the global data buffer. The throughput is given in data blocks per second and is calculated as:

$$TP = \frac{1}{R * 10^{-9}} \tag{5.4}$$

- Maximum rate (R_{max}) is defined as the maximum arrival time difference between two subsequent data blocks and is calculated as:

$$R_{max} = max(T_{i+1} - T_i) \tag{5.5}$$

- Minimum rate (R_{min}) is defined as the minimum arrival time difference between two subsequent data blocks and is calculated as:

$$R_{min} = min(T_{i+1} - T_i) \tag{5.6}$$

5.4 Performance Evaluation Process

The performance evaluation process consists of two phases. During the first phase we wrote and simulated the behavioral VHDL program code of the individual components of the architecture in Modelsim. The simulation results are written to a vcd file. The vcd file consist of a list of timestamps and the signals which have transitioned from zero to one or the other way around at that timestamp. Vcd files are large and not easily readable, therefore we need a script which translates the raw vcd data into useful data. During the second phase we have written several awk-scripts, which depending on the required data, produces the data used in the graphs in the following Sections.

5.5 The Experiments

To evaluate different aspects of the multiprocessor, several experiments are conducted. The limitations and the capabilities of our message-passing architecture are evaluated

Table 5.6: Instruction counts of the Custom Application

Stage	Instruction count
Partition 1	94
Partition 2	88
Partition 3	88
Partition 4	93
Total Parallel	363
Serial	223

Table 5.7: The scheduling overhead for the different applications

Application	First stage	Last stage
FIR filter	0.09%	0.22%
JPEG Decoder	0.29%	0.65%
Custom Application	0.10%	0.28%

with the results of the different experiments. The different experiments are listed below and will be discussed in more detail in the following Subsections.

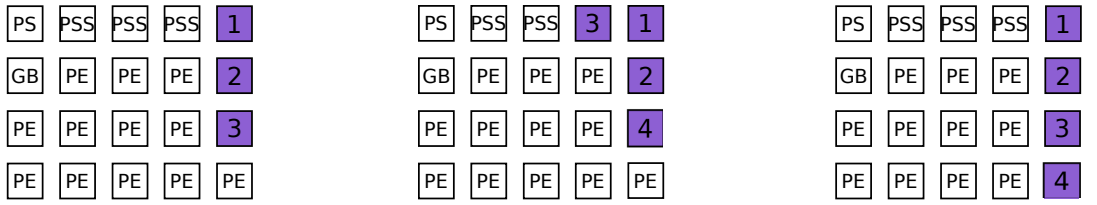
- Varying the input data stream size.
- Varying the number of pipelines.
- Varying the buffer size.
- Varying the number of pipeline stages.
- Varying the network traffic volume for two different platform configurations.

5.5.1 Increased Input Sizes

In order to determine the influence of the input size on the performance, the input sizes, i.e. the number of input blocks is increased. The experiment is done for all the applications, which have been discussed in Section 5.2.

The platform used for this experiment has the baseline configuration. In order to make a fair performance comparison, both the partitioned and the serial version of every application are executed on the platform. A single pipeline is executed on the platform for a different number of input data blocks for every application. The pipelines of the different applications are scheduled according to the data-flow graph. The mapping of the different pipelines on the platform for the different applications is given in Figure 5.10.

The results for this experiment are shown in Figure 5.11, where the speedup of the different applications are given as a function of the input size. The points on the x-axis are differently defined for the different applications. The first point on the x-axis in the case of the JPEG application is an image with the dimension of 64 by 64 pixels. The image dimensions are doubled in both directions for every subsequent point on the axis. The starting point on the axis for the custom application and the filter represent



(a) FIR/JPEG 3-core (b) JPEG 4-core (c) Custom application

Figure 5.10: The different pipeline mappings

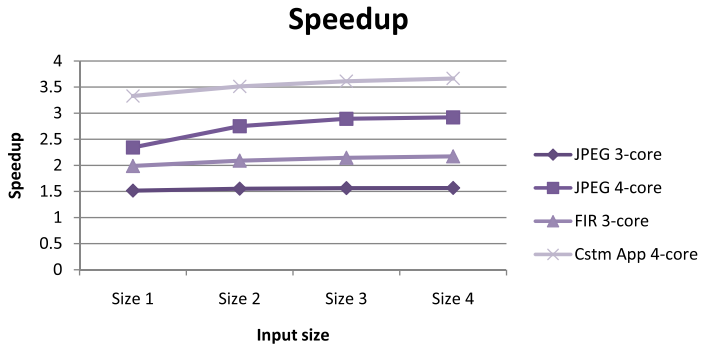


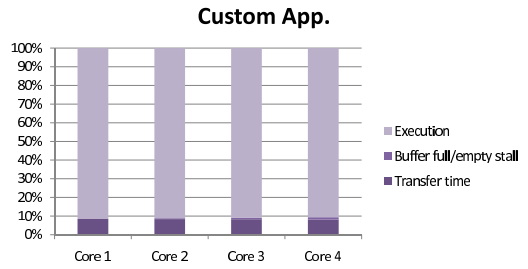
Figure 5.11: The Speedup as a function of an increased input-stream size

a signal consisting out of 800 samples. The number of samples is doubled for every subsequent point on the graph.

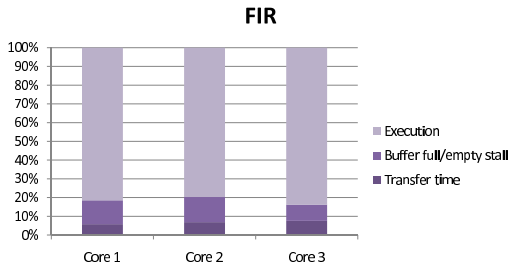
An ideal speedup increases linearly with the number of cores, i.e. the speedup should be equal to the number of cores used. This ideal speedup is generally not reached, because there is usually an additional percentage of time spent on communication between tasks. The ideal speedup for the FIR is 3 and the ideal speedup for the JPEG 4-core and custom application 4. The speedup observed in Figure 5.11 for FIR and JPEG 4-core are far from ideal (2.1x and 2.9x respectively), however the custom application comes close to reaching the ideal speedup with 3.6x. These performance differences are partly caused by the communication overhead and mostly caused by an unbalanced workload. The cores with a lighter workload will spend a lot of time idle, waiting for the downstream core to finish processing or waiting for the upstream core to send the next data block.

Figure 5.12 shows the execution break down of every core for every application. The cores from the custom application spent most of the execution time (around 90%) processing data and a small percentage of time is spend on transferring data to the downstream task or global data buffer. The JPEG and FIR applications have an unbalanced workload, where the cores with a lighter workload spend a large percentage of the time waiting, wasting the resources, resulting in lower speedups.

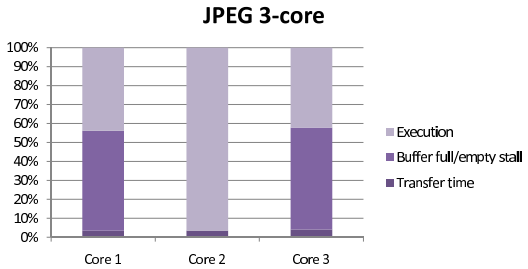
Comparing the execution-breakdown graphs of the two JPEG configurations in Figure 5.12c and Figure 5.12d, we can further observe the influence of load balancing. The total percentage of idle time for the last stage in the pipeline has dropped with the addition of the fourth core (from +/-55% to +/-15% idle time). The addition of the forth core to the second stage in the pipeline has doubled the data block arrival rate of the last stage in the pipeline.



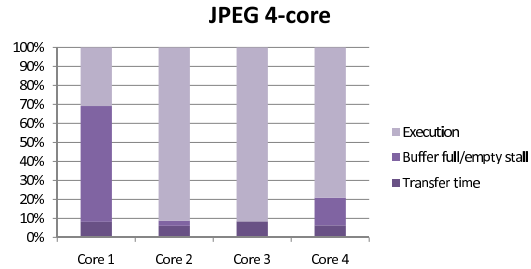
(a) The execution breakdown for the cores of



(b) The execution breakdown for the cores of



(c) The execution breakdown for the cores of the 3-core JPEG decoder



(d) The execution breakdown for the cores of the 4-core JPEG decoder

Figure 5.12: Execution breakdown for each application

Table 5.8: The execution times of core 1 for the 3-core and 4-core JPEG pipelines

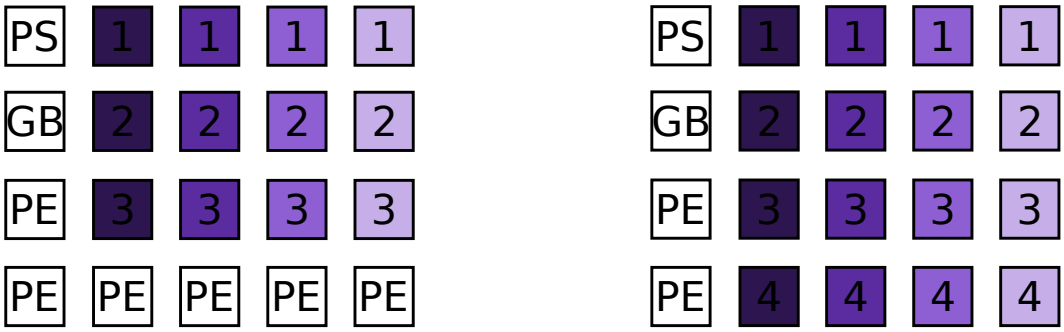
Configuration	Total execution time (ns)	Buffer full stalls (ns)
3-core JPEG	37714885	19702920
4-core JPEG	20381235	12212460

There is one unexpected characteristic observed in the bar-graph of 5.12d, the total buffer stall percentage of the first stage in the 4-core configuration has increased in comparison to the 3-core configuration. After analyzing the results further we discovered that the total number of stall cycles did not decrease at the same rate as the total execution time has, which increased the percentage of the total number of buffer full stalls. The results are listed in Table 5.8

5.5.2 Multiple Pipelines

To fully take advantage of all the available resources, multiple pipelines are executed in parallel. The number of pipelines are varied from one to four. The input data is then divided over the multiple pipelines. The pipelines are mapped onto the baseline platform, which is shown in Figure 5.13.

This experiment shows how the architecture handles a large number of resources, i.e. the scalability of the architecture. The results for this experiment are presented in Figure 5.14. The optimal speedup for the increased number of pipelines, would be when the speedup scales linearly with the number of pipelines. The curves are almost linear with a slight deviation (not noticeable in the graph), this small deviation from the optimal speedup is caused by the increased traffic volume, which increases the



(a) FIR (b) JPEG and Custom application

Figure 5.13: The mapping of the multiple pipelines on the platform for every application

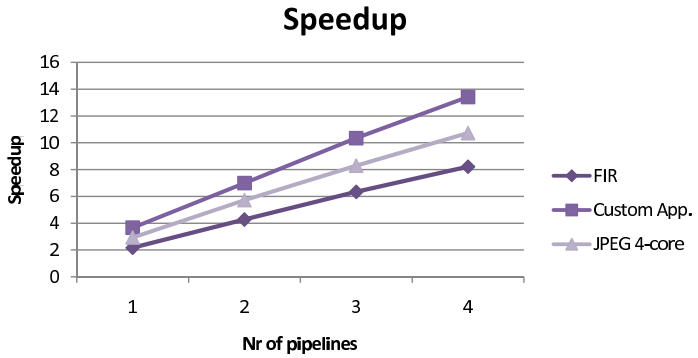


Figure 5.14: The Speedup as a function of the number of pipelines

communication overhead.

The throughput as a function of the of number pipelines is given in Figure 5.15. The throughput scales almost linearly with the number of pipelines for all applications. This linear increase in throughput is expected, if a pipeline produces output data blocks at a certain average rate, doubling the number of pipelines will double the average rate at which the blocks arrive at the global data buffer. To conclude, the architecture scales well with the increase in pipelines.

5.5.3 Varied buffer sizes

To determine the influence of the MPB size on the performance of the applications, the MPB size is increased. This experiment has only been done for the JPEG and the FIR application because the custom application has a completely balanced workload and only uses the buffer to briefly store one data block. The results are shown in Figure 5.16, where the x-axis is the buffer size in words and the y-axis is the speedup. From this figure we can conclude that the buffer size does not influence the execution time of the applications. The main reason for this is that the rate at which the critical stage processes the data blocks does not increase with the increase of the buffer size. To conclude, the buffer size must be big enough to fit all the data required for one iteration, increasing the buffer size beyond this point has no influence on the performance.

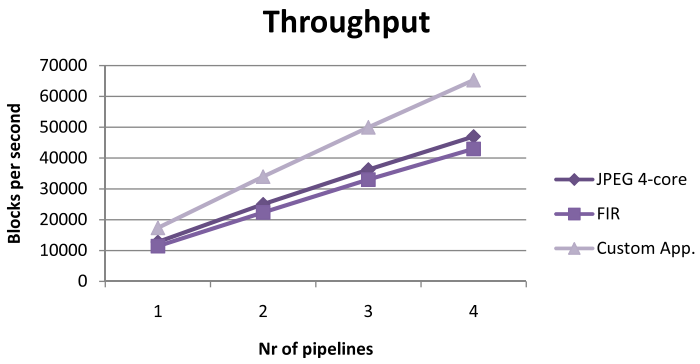


Figure 5.15: The Throughput as a function of the number of pipelines

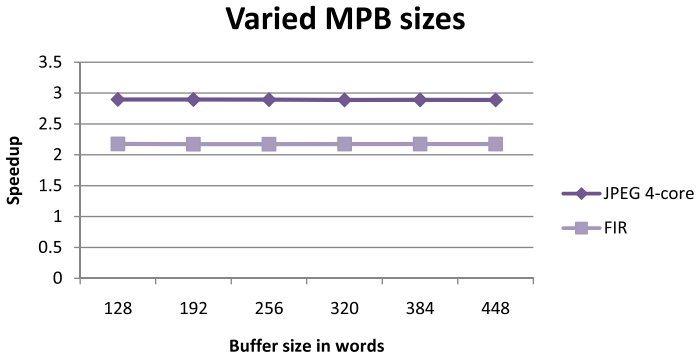


Figure 5.16: The Speedup as a function of the MPB size

5.5.4 Varied number of pipeline stages

We determine how the architecture performs for applications with deeper pipelines by varying the number of pipeline stages used for a particular application. This experiment consists of two parts: for the first part we have kept the window size and input size of the FIR filter constant and divided the same amount of computational work over an increased number of pipeline stages. The amount of work done by each stage is decreased with addition of each pipeline stage. The pipeline is executed on the baseline platform and the results are shown in Figure 5.17. The speedup is given as a function of the number of pipeline stages.

The speedup increases with the increased number of cores. There is a slight dip in the curve with the addition of the seventh core, which can be explained by the increase in the total percentage of the communication overhead. Increasing the number of pipeline stages increases the communication overhead while decreasing the amount of work done by one stage. If the time spent on communication reaches a certain percentage of the total execution time, the speedup will significantly decrease.

The second part of this experiment involves the custom application. We are interested in the performance of an application with a very deep pipeline. The custom application is modified to this end and can now be partitioned into a maximum of sixteen stages with an equal workload. The number of pipeline stages are doubled for every run. The pipeline is executed on the baseline platform and the results are shown in Figure 5.18.

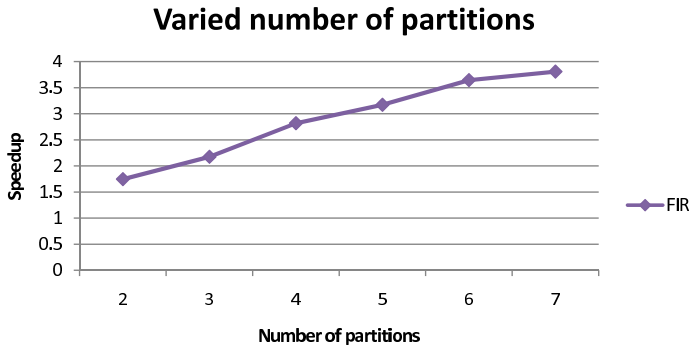


Figure 5.17: The Speedup as a function of the number of partitions

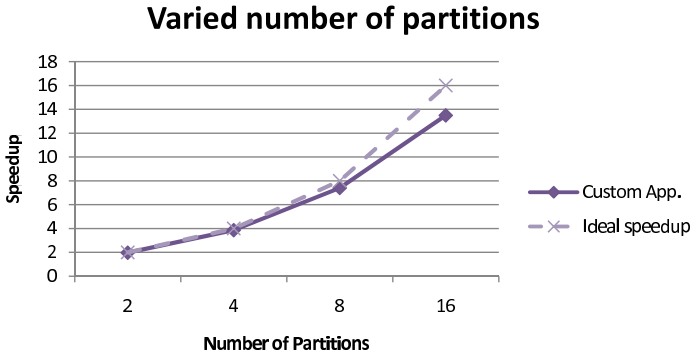


Figure 5.18: The Speedup as a function of the number of partitions

The speedup curve closely approaches the ideal speedup, the transition from eight to sixteen cores deviates from the ideal speedup curve. The communication overhead has increased to the point that it includes a relatively large percentage of the total execution time, which is expressed in the decrease in speedup.

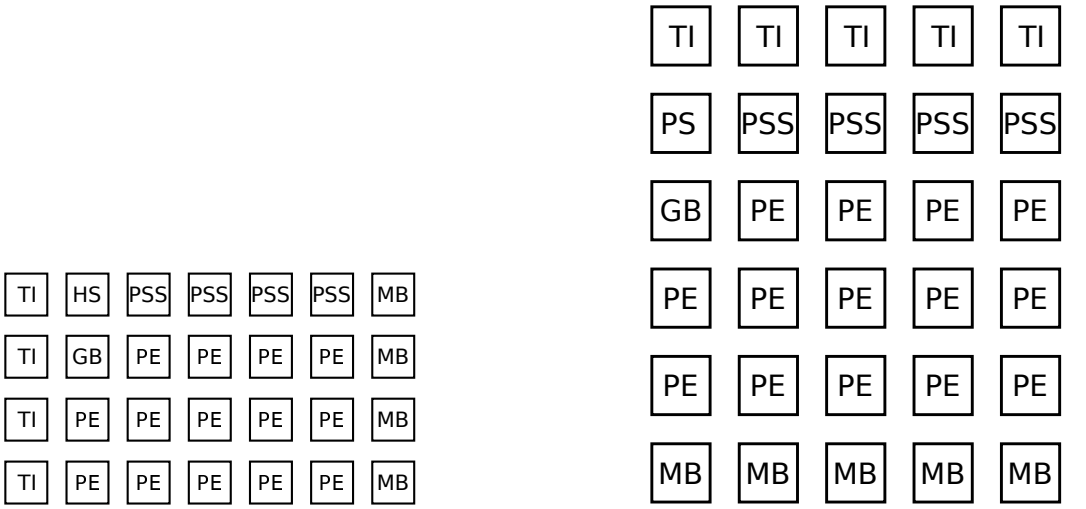
5.5.5 Increased traffic volume

To determine the influence of the traffic volume on the performance. Keeping the complete multiprocessor configuration described in Chapter 4 in mind, the traffic flowing through the network partly consists of read and write request for the shared-memory. We simulate additional traffic through the network by using traffic injectors and a shared-memory consisting of four memory banks. The traffic injectors generate read and write requests for the shared-memory banks. The destination memory banks for the requests and the type of request is random. Read requests are answered by sending read data back to the requester.

Two different mesh configurations are considered for this experiment. These different configurations both determine the performance for different traffic flow patterns relatively to the data flow of the pipelines. The tasks in both configurations have the same mapping to cores as in the previous experiments, where four pipelines are executed in parallel for every application. The first platform configuration is shown in Figure 5.19a and consists of the baseline platform with the addition of 4 traffic injectors

Table 5.9: The injection rates

Time interval	Injection rate
4 packets every 44 cycles	0.09
4 packets every 36 cycles	0.11
4 packets every 28 cycles	0.14
4 packets every 20 cycles	0.2
4 packets every 12 cycles	0.33



(a) The first platform configuration (b) The second platform configuration

Figure 5.19: Top-level view of the two platform configurations

(TI) and 4 memory banks (MB). The data flows from the traffic injectors towards the the memory banks in a perpendicular direction as the data in the pipelines.

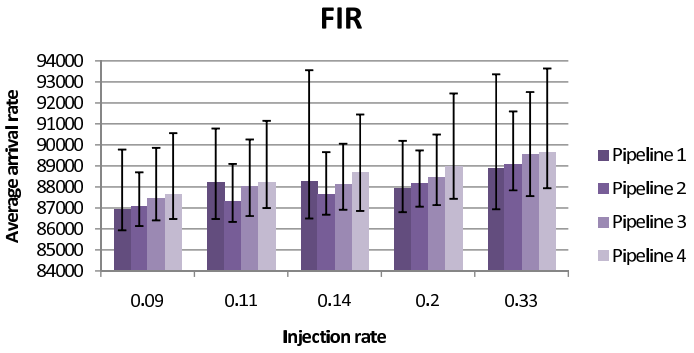
The second configuration consists of the same components as the previous configuration, the difference is in the locations of the traffic injectors and the memory banks. Figure 5.19b shows the second platform configuration. The data flows from the traffic injectors in the same direction as the data in the pipelines towards the memory banks.

Every traffic injector sends a packet into the network periodically i.e. once every interval. Decreasing this interval increases the injection rate. The injectors are synchronized to inject four packets every interval. The time intervals chosen are listed in the Table 5.9. The table also lists the corresponding injection rate in packets per second.

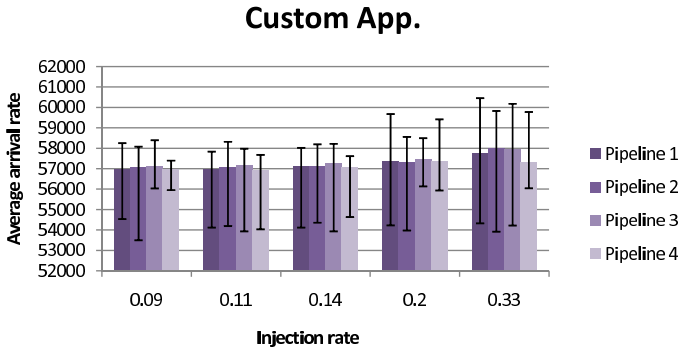
The read-request packets consists of four bytes of payload data, while the packets belonging to write requests can consists of 4,12 or 68 bytes of payload data. Read data always consists of 64 bytes of payload data, simulating the replacement of one cache line.

5.5.5.1 Results of the first configuration

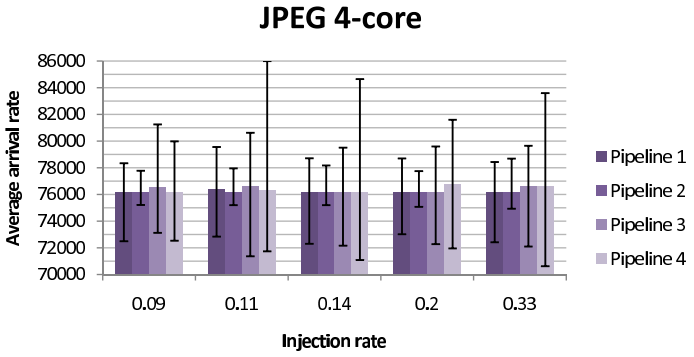
Figure 5.20 display the bar-graphs of the data block arrival rate at the global buffer as a function of the injection rate of the traffic injectors for the FIR filter, Custom application and JPEG decoder. The injection rate is given in packets per cycle and the



(a)



(b)



(c)

Figure 5.20: The average arrival rate as a function of the injection rate for each application

arrival rate is defined as the average arrival rate at the global data buffer of data blocks belonging to the same pipeline. The error-bars on the bar-graphs represent R_{max} and R_{min}, defined in Section 5.3. We will analyze the bar-graphs one by one, starting with the FIR filter.

The first observation of the bar-graph of the FIR application is that the average arrival rate of the different pipelines for one injection rate slightly increases. This can be explained by the fact that the distance between the pipeline and the global data buffer increases with every subsequent pipeline. Data blocks traveling on certain paths are more affected by the injected packets then on other paths, the path taken by the data blocks which is most affected by the network is now referred to as the critical path.

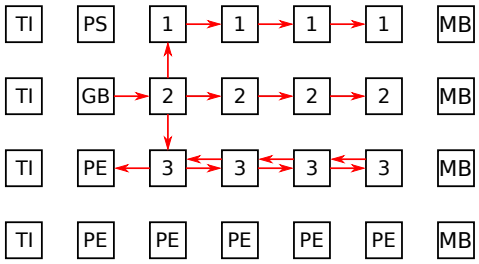


Figure 5.21: The critical path of the FIR filter

The critical paths of the FIR filter are given in Figure 5.21. The data blocks traveling to and from the global data buffer are more affected by the injected traffic, since the injected packets travel in the same direction. Packets that spent a large amount of time on the critical paths have an increased chance of getting delayed as a consequence of the traffic on the path.

As discussed in Section 5.2.2, each stage in the FIR pipeline needs to fetch a new data block from the global data buffer every iteration. This characteristic increases the effect of the injected traffic on the execution-time of a single data block going through the pipeline. Therefore the average arrival rates of the pipelines increases with the increased distance between the global data buffer and the pipeline.

From Figure 5.20a we can also observe a slight increase in the average arrival rate of the pipelines with the increase in traffic rate, which is as expected. The increase in traffic increases the time spent in the network by the data blocks, which increases the total time spent in the pipeline.

The error bars on the bar-graph represent the range of the arrival rates, i.e. the maximum and minimum difference between two subsequent blocks. Recall that R_{max} is calculated as $max(T_{i+1} - T_i)$. R_{max} is reached when T_{i+1} is as high as possible and T_i is as low as possible. The opposite conditions apply for R_{min}. The occurrence of these circumstances are highly dependent on the traffic volume at particular moments in time and are therefore not constant. However, as can be observed in higher traffic rates (e.g. Injection rate of 0.2 and 0.33), the chance of these situations is increased.

Moving on to the bar-graph of the custom application. The average arrival times of the pipelines also increases with the increase in injection rate. However the difference between the first and last injection rate is much smaller for this application in comparison with the filter. Only the first stage needs an new input block from the global data buffer, which is indicated by the critical path in Figure 5.22. This drastically decreases the affect of the injected traffic on the average arrival rates.

The difference between the average arrival rates of the pipelines for one injection rate is more are less constant with the exception of the last rate. This indicates that the distance between the global data buffer and the pipelines has little effect on the data blocks in the custom application. This behavior can be explained by the randomness of the size and destination of the injected packets, the different pipelines can experience different delays depending on the size and destination of the injected packets.

The arrival rate ranges for this application also slightly increases with the increase

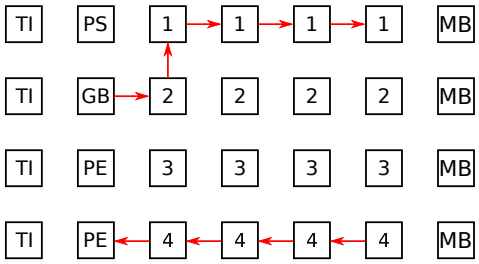


Figure 5.22: The critical path of the custom application

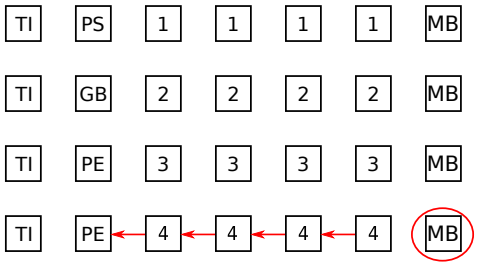


Figure 5.23: The critical path of the JPEG decoder

in traffic, which is similar to the arrival rate range for the previously discussed FIR.

The bar-graph of the JPEG decoder behaves differently. The arrival rates for all points on the x-axes are more or less constant, which is mostly as a consequence of the unbalanced workload. The rate at which the Huffman stage produces data blocks is much higher then the rate at which the two IDCT stages consume data blocks. The data blocks coming from the Huffman stage will be placed in the MPB for a longer period of time, which implies that the affects of the traffic volume on the data blocks traveling to the IDCT stage does not have an influence on the total execution time of that block. The data blocks traveling towards the global data buffer are interfered by injected traffic from one particular memory bank. Figure 5.23 shows the critical path, on which the resulted data blocks travel towards the global data buffer. The memory bank that is responsible for the injected traffic on this critical path is indicated with a red circle. A memory bank only injects a packet into the network upon the receipt of a read request for that particular memory bank, which is approximately 20 % of the total amount of requests.

The wide arrival rate ranges indicated by the error-bars in Figure 5.20c do not vary much for the different injection rates. The difference in arrival rate are mostly caused by the simultaneous writes to the global data buffer and occasionally by the injected packets of the memory bank. The injection rates have a marginal or no effect on the arrival times of the data blocks for this configuration.

Now that an average arrival rate has been established for the several pipelines, we want to know how much the arrival rates deviate from this average arrival rate. The percentage of the average deviation for all applications is given in Figure 5.24. The average deviation is defined in Section 5.3. Two aspects of the bar-graph stand out. The first one is that the JPEG decoder has on average a higher deviation percentage in comparison with the remaining two applications. This is caused by the large output

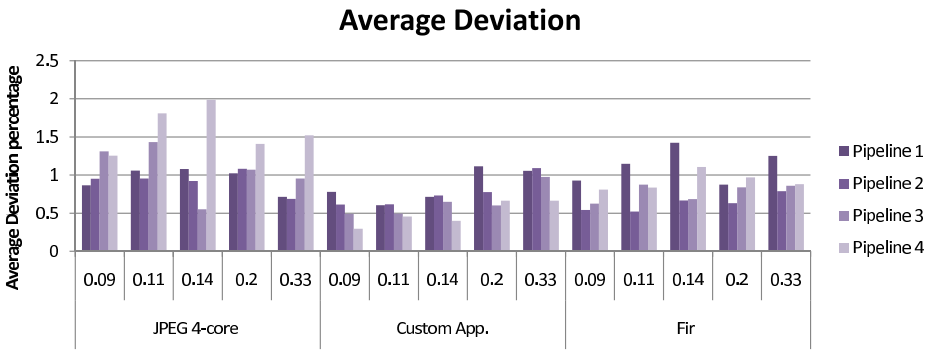


Figure 5.24: The average deviation as a function of the injection rates

data block size and the constant change from simultaneous writes to serial writes to the global data buffer.

The second aspect is that the deviation is inconsistent in every way, from which we can conclude that it is not only influenced by the injection rate but on the circumstantial traffic on the critical paths. The amount of traffic present on the critical paths depends on the amount of traffic injected in the network, the number of writes to / reads from the global data buffer and the size of the data blocks.

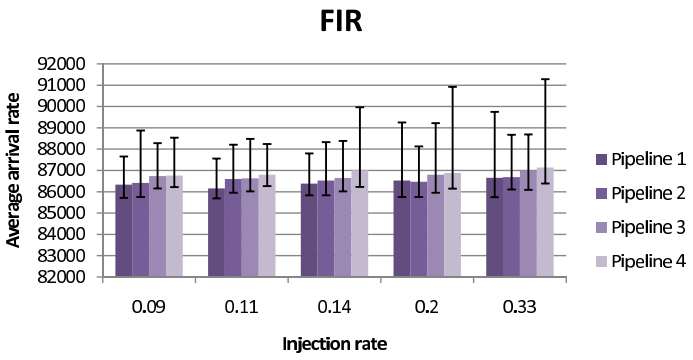
5.5.5.2 Results of the second configuration

The bar-graphs for the different applications are given in Figure 5.25. The bar-graphs of the three applications more or less have the same behavior. The average arrival rate for every pipeline increases with the increase in injection rate. The bar-graph of the JPEG decoder has two inconsistencies at rate 0.11 and 0.14. These are caused by the high amount of simultaneous writes to the global data buffer.

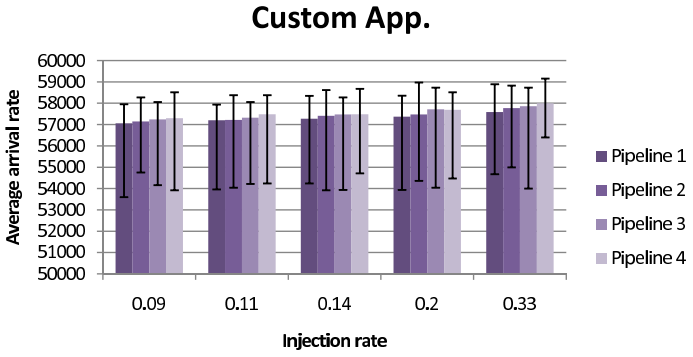
The average arrival rate ranges indicated with the error-bars are much wider for the JPEG decoder then the ranges for FIR and the custom application. The critical path of the JPEG decoder is given in Figure 5.26c. The critical path consists of two hops, which limits the total effect of the traffic on the arrival times of the data blocks. The large difference in arrival times of the data blocks is independent from the injection rate and is only caused by the simultaneous read and writes to the global data buffer and the size of the output packets.

The average deviation for the applications is given as a function of the injection rates in Figure 5.27. The first observation is that the average deviation of the custom application is higher then for the remaining two applications. This can be explained by the critical paths of the different applications (see Figure 5.26), the custom application clearly has the longest critical path. The next observation is that the average deviation is not only dependent on the injection rate in this configuration as well.

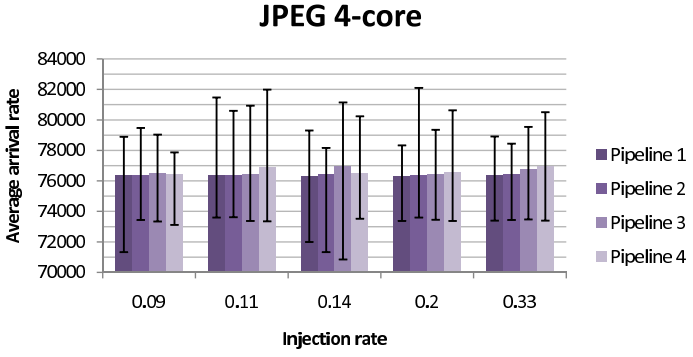
The NoC [7] used for this simulation has Best Effort (BE) flow control, where packets are fairly routed in the order at which they arrive at the router. This form of router arbitration has the drawback that the packets have unpredictable latencies, which results in unpredictable data block arrival rates observed in Figure 5.27 and Figure 5.24. Real-time systems require the guarantee that a data block will arrive



(a)

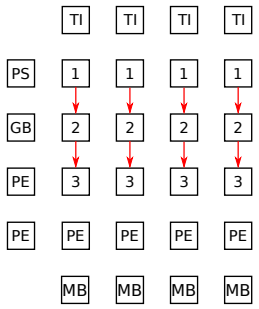


(b)

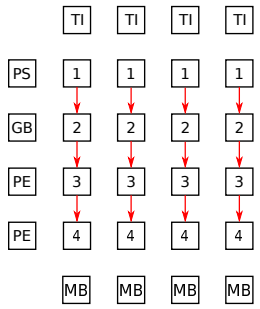


(c)

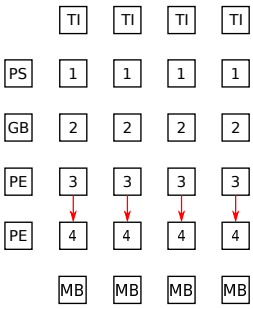
Figure 5.25: The average arrival rate as a function of the injection rate for each application



(a) Critical path of the FIR filter



(b) Critical path of the Custom application



(c) Critical path of the JPEG decoder

Figure 5.26: The different pipeline mappings

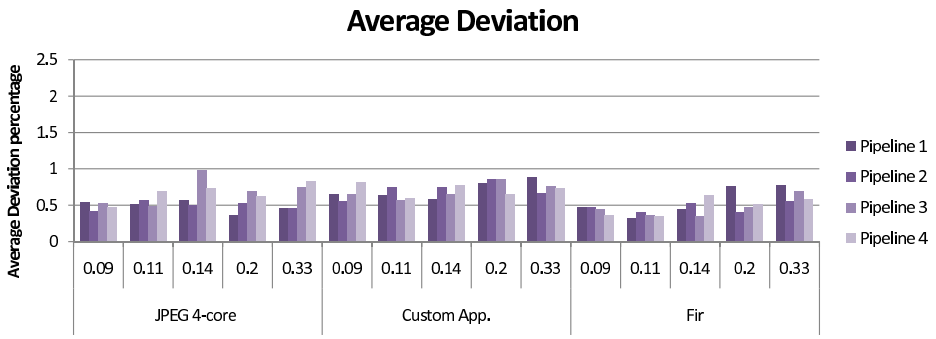


Figure 5.27: The average deviation as a function of the injection rates

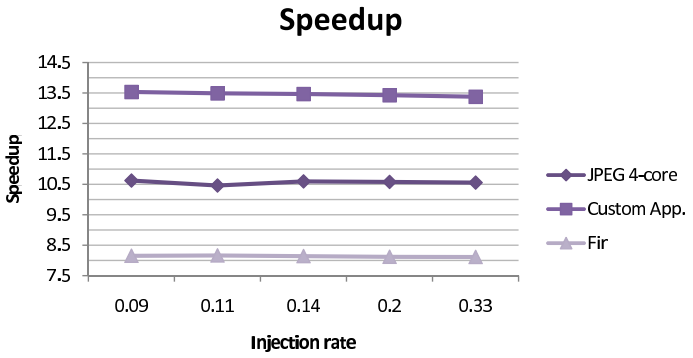


Figure 5.28: The speedup for the first configuration

within a certain time interval, from which we conclude that Guaranteed Service router arbitration might be better suited for our architecture to meet the requirements of real-time systems.

5.5.5.3 Comparison of the two configurations

The increase in the average arrival rate with the increase in injection rate for the second configuration is considerably lower than for the first configuration. The number of hops in the critical path of the first configuration is higher than the number of hops in the critical path of the second configuration. This indicates that the placement of the components is an important aspect of the multiprocessor that must be considered during the design time. The length of the critical paths of the data blocks determines the degree in which the traffic volume affects the arrival rate of the data blocks.

In order to determine the influence of the traffic volume on the performance of the architecture, the speedup is given as a function of the injection rate for both configurations. Observed in Figure 5.28 and Figure 5.29 is that the speedup is only marginally affected by the injection rate in both cases, which indicates that the performance is marginally affected by the traffic volume in the network.

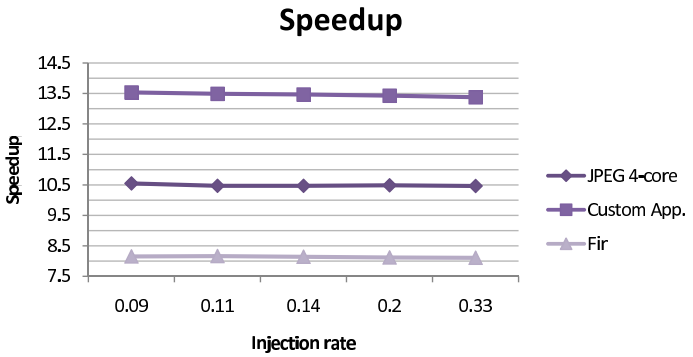


Figure 5.29: The speedup for the second configuration

5.6 Summary

In this Chapter we described our baseline platform configured in a four by five mesh consisting of several ρ-Vex processor, a primary scheduler, a global data buffer and several secondary schedulers. Three application have been selected to run on our processor, which include the JPEG encoder, FIR filter and a custom application. The partitioning strategies and the mapping of these applications on the platform have been explained. To evaluate our architecture four evaluation metrics are used, which include speedup, throughput, average arrival rate, average deviation, the minimum arrival rate and the maximum arrival rate.

Different experiments have been conducted to evaluate different aspects of our architecture. The first experiment involved increasing the input stream size from which we conclude that an unbalanced load severely limits the speedup and can be considered as one of the limitations of stream programming. Further more we have shown that the speedup approaches the ideal speedup in the case of a perfectly balanced workload where the ideal speedup is 4 and the measured speedup is 3.6. In order to determine how this architecture scales with respect to multiple pipelines, the number of pipelines have been increased for each applications. The speedup and throughput have increased almost linearly with the number of pipelines. To determine the influence of the buffer size on the performance, the buffer size has been increased. The buffer size must be big enough to store all data needed in one iteration, further increasing the buffer size does not improve the performance. We were also interested in the performance of our architecture for applications with deeper pipelines. To that end we have increased the number of pipeline stages for the FIR filter and the custom application, to conclude that the communication overhead increases significantly for a certain number of pipeline stages. The point at which the communication overhead significantly affects the speedup depends on the application.

The last experiment determined the influence of the traffic volume on the execution time of the applications. The speedup did not significantly change with the increase in traffic rate, to conclude that the performance is not significantly affected by the increased traffic volume in the network. We have also studied the affects of the network on the individual data blocks going through the pipeline, to determine if this architecture is suitable for real-time applications, which requires throughput guarantees. We

have concluded that the average deviation of the data blocks arriving at the global data buffer is not only dependent on the injection rate but also depends on simultaneous read and writes by the multiple pipelines to the global data buffer, the length of the critical path and the size of the data blocks. The unpredictable behavior of the arrival times of the data blocks have led us to believe that a Best effort QoS strategy [20] might not be suitable for real-time streaming applications, instead we need to consider a guaranteed service QoS strategy [20].

6 Conclusion

This chapter gives a summary of the work done during the project and recommendations for future work.

6.1 Summary

Streaming applications are applications partitioned into tasks working on data blocks in a data stream. The advantage of such a partitioning is that the tasks are pipelined resulting in an increased throughput and decreased execution-time compared to the serial implementation.

A simple simulation described in Chapter 2 of the network performance has indicated that an architecture consisting of a distributed memory where the processors communicate via message passing is better suited for streaming applications than a shared-memory architecture. To that end, a message passing architecture based on the requirements of streaming applications is designed and implemented.

The proposed multiprocessor architecture, as explained in Chapter 3 and 4 consist of a dynamic distributed scheduler and a distributed memory where the cores communicate via message passing. The dynamic scheduler schedules tasks at run-time based on a nearest neighbor scheduling policy. The key components of a message-passing tile are a DMA-controller, a message passing buffer and a outgoing buffer. The message passing buffer is used to store incomming messages and the outgoing buffer is used to briefly buffer unsent messages. The DMA-controller manages the transfer of data between the several components on the message-passing tile. Existing message-passing buffers are either implemented as a RAM, which imposes the problem of managing the buffer address by the application developer or as FIFO channels. Each communicating pair (sender and reciever) has their own FIFO channel. The use of FIFO channels eliminates the need for buffer address managing by the application developer but decreases the flexibility of the architecture. The FIFO channels must be configured at compile-time and can not be dynamically reconfigured. Our approach combines the advantages of both the RAM and FIFO implementations by introducing the buffer manager, which keeps track of the order in which messages have arrived along with the addresses of the messages within the buffer.

The message passing architecture is evaluated in Chapter 5 by conducting several experiments evaluating different aspects of the architecture. The platform on which we have done our experiments includes a distributed scheduler, twenty ρ-Vex processors and a global data buffer which is used to store the streaming data. Three applications have been selected to run on our architecture: the JPEG decoder, a FIR filter and a custom application. The custom application is specifically written to expose the capa-

bilities of the architecture. From the first experiment which involved varying the input stream sizes, we have concluded that balancing the workload is crucial for optimal performance. Balanced workloads can result in speedups that can closely approach the ideal speedup, which is observed in the results of the custom application. The speedup of the balanced custom application on four cores is 3.6 in comparison with the unbalanced JPEG encoder on three cores with a speedup of 1.4, which emphasizes the importance of load balancing. Further more, increasing the number of pipelines, almost linearly increases the speedup and throughput. We have also tested the architecture for applications with a deeper pipeline, where the number of pipelines of the filter and custom application are increased. This experiment has exposed the increased communication overhead as a consequence of the high amount of pipeline stages. The speedup for the custom application with sixteen pipeline stages is 13.4 and the speedup for the FIR application consisting out of seven stages was 3.8. The last experiment involved increasing the traffic volume in the network by means of traffic injectors. To determine the effect on the run-time of the individual data blocks going through the pipeline, we calculated the deviation of the arrival times of the data blocks arriving at the global data buffer for various traffic-injection rates. The deviation is not significantly influenced by the traffic injection rate compared to the overhead of multiple pipelines trying to send their results to the global data buffer simultaneously. The highest average deviation percentage of 2% has been calculated for the JPEG decoder. The maximum deviation percentage can get as high as 8%. The unpredictable behavior of the arrival times of the data blocks indicates that a guaranteed service router-arbitration strategy might be better suited for real-time streaming applications.

6.2 Future Work

While we have provided a complete message-passing archtecture and a dynamic distributed scheduler, there are still aspects which could be improved and / or further investigated.

- As concluded in the previous section, a balanced workload is crucial for optimal performance. However this is not an easy task to do manually. Therefore, we need a dedicated tool that can partition the code with an optimal workload.
- The current message passing library supports the basic message passing functions. In order to offer more flexibility to the application developers, the library must be expanded.
- Evaluate the message-passing architecture for applications with more complex pipelines, applications with a irregular dataflow pattern, etc.

Bibliography

[1] M. Geilen and T. Basten, "Requirements on the execution of kahn process networks," *ESOP'03 Proceedings of the 12th European conference on Programming*, pp. 319–334, 2003.

[2] S. Brookes, "On the kahn principle and fair networks," *Technical Report CMU-CS-98-156, School of Computer Science, Carnegie Mellon University*, 1998.

[3] E. Lee and T. Parks, "Dataflow process networks," *Readings in hardware/software co-design*, pp. 59–85, 1995.

[4] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," *Appears in Proceedings Seventh International Symp. Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pp. 2 – 11, 1996.

[5] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun, "The stanford hydra cmp," *IEEEMicro*, vol. 20 Issue 2, pp. 71–84, 2000.

[6] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 2006.

[7] S. S. Kumar and R. van Leuken, "A 3d network-on-chip for stacked-die transactional chip multiprocessors using through silicon vias," *Design & Technology of Integrated Systems in Nanoscale Era (DTIS), 6th International Conference*, 2011.

[8] Intel, "The scc platform overview." http://www.intel.com, 2010. Accessed: 28/8/2012.

[9] T. M. (IL) and R. van der Wijngaart (SSG), "Rcce: a small library for many-core communication." http://www.intel.com, 2010. Accessed: 28/8/2012.

[10] A. Nieuwland, O. P. Gangwal, R. Sethuraman, N. Busa, K. Goossens, R. P. Llopis, and P. Lippens, "C-heap: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of emebedded signal processing systems," 2002.

[11] IBM, "Cell broadband engine architecture," 2007.

[12] A. DULLER, G. PANESAR, and D. TOWNER, "Parallel processing the picochip way!," *Communicating Process Architectures*, 2003.

[13] T. R. Halfhill, "Ambric's new parallel processor," *Microprocessor Report*, 2006.

[14] A. Inc., "Ambric, creating massively parallel solutions," 2008.

[15] B. Baas, Z. Yu, M. Meeuwsen, O. Sattari, R. Apperson, E. Work, J. Webb, M. Lai, T. Mohsenin, D. Truong, and J. Cheung, "Asap: A fine-grained many-core platform for dsp applications," *IEEEMicro*, vol. 27 Issue: 2, pp. 35–45, 2007.

[16] S. Wong, T. van As, and G. Brown, "ρ-vex: A reconfigurable and extensible softcore vliw processor," *Accepted at the IEEE International Conference on Field-Programmable Technology*, pp. 369 – 372, 2008.

[17] CCITT, "Information technology digital compression and coding of continuous-tone still images requirements and guidelines," 1992.

[18] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," *IEEE Annual Workshop on Workload Characterization*, 2001.

[19] J. O. SMITH, "Introduction to digital filters." https://ccrma.stanford.edu/~jos/fp/. Accessed: 28/8/2012.

[20] B. Gebremichael, F. Vaandrager, and M. Zhangy, "Formal models of guaranteed and best-effort services for networks on chip," *Verification of Hard and Softly Timed Systems (HaaST)*, 2005.