MASTER'S THESIS 2020

Evaluating a Real-time Multi-core Processor for Embedded Streaming

Linus Gudmundsson, Jacob Canbäck

ISSN 1650-2884
LU-CS-EX: 2020-24

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-24

Evaluating a Real-time Multi-core Processor for Embedded Streaming

Linus Gudmundsson, Jacob Canbäck

Evaluating a Real-time Multi-core Processor for Embedded Streaming

(A case study on the Patmos architecture)

Linus Gudmundsson
dat15lgu@student.lu.se

Jacob Canbäck
dat15jca@student.lu.se

June 22, 2020

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Jörn W. Janneck, jorn.janneck@cs.lth.se

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

Modern processors provide high performance for just about any average application. This is possible as they are optimized for general purpose use. However, optimizations that improve the average case execution time often result in an increase of the *worst-case execution time* (WCET). Furthermore, to determine the WCET for arbitrary applications on such processors is equivalent to solving the halting problem, which has been proven to be undecidable. This in turn leads to WCET analysis programs abstracting from implementation details and approximating results in order to keep the complexity at a manageable level [29]. In this project we investigate a processor designed for real-time systems, Patmos [28], and how it functions when used for non time driven applications. Patmos features a processor array and a network on chip used for transfers. To conduct the investigation we implemented an actor based streaming application which performs MPEG-4 decoding. When implementing an actor based application on a processor array there needs to be some form of mapping between actors and cores. We explored how this mapping impacted the performance and if this mapping could be automated. We found that the most problematic limitation was the amount of available memory, which led to extensive trimming on the application's memory usage. When this was done it was possible to run the application on the Patmos platform. The mapping between actors and cores was found to significantly impact the performance, with some mapping layouts almost doubling the performance. In addition it was possible to automatically compute the mapping layouts. During this project the Patmos architecture has shown considerable potential and will most likely be beneficial for both real-time and data driven applications once a stable version of Patmos is released.

Keywords: Patmos, Actors, Streaming Application, Mapping, Memory Optimization, Embedded Processor Array

Acknowledgements

This project was conducted at the Department of Computer Science, Lund University. We wish to thank Jorn W. Janneck for his invaluable assistance as our supervisor and for providing us with the necessary hardware, Flavius Gruian for his guidance on all administrative matters regarding the project, Martin Schoeberl and Eleftherios Kyriakakis from the Patmos development team for their help whenever we had questions regarding Patmos, and Anders Bruce for setting up our project workstations.

Contents

Chapter 1
Introduction

This chapter introduces the Patmos platform and the streaming application briefly, as well as describes the main problems we wish the explore in this thesis. It also describes our research goals and attempts to give the readers an overview of the Patmos platforms unique aspects and elaborate on why this is interesting.

1.1 Patmos Microprocessor

1.1.1 Real-Time

Today almost all modern processors can handle most generic applications within a reasonable time-frame. This is a result of being optimized for average case execution time. This lets them run any application, on average, at a reasonable high speed. These optimizations do however often have a drawback as they can lead to a degradation of the worst case execution time [17]. Because of this an application's run time can vary a great deal between executions. While this is completely fine for an average application, some critical applications, such as plane or train controllers, cannot afford such stalls as they are not allowed to fail even once. For such applications some processors have been designed to be real-time and *worst case execution time* (WCET) analyzable. One of these processors is Patmos, a processor array developed to be the next big thing in the stale market of real-time systems. Unlike most architectures which try to minimize the average-case execution time of applications Patmos focuses on exposing and minimizing the WCET [30]. This makes it a perfect fit for any applications that requires stability and predictability as there will never be any unexpected exceptions when it comes to execution time. Calculating WCET in a single core environment is fairly complex, but can still be achieved with some effort [24]. Patmos however takes it a step further and applies this even to multi-core applications. This is possible because of the way the cores, and the communication amidst them is designed and regulated.

1.1.2 Processor Array

Patmos is a many-core architecture and is composed of a matrix of cores connected using a Network-on-Chip (NoC). Each of these cores have their own local resources, see figure 1.1. The network to the right contains one router per core. Each router is connected with four other routers as well as one core. The broken arrows form the bi-torus topology. One of the cores is expanded to the left to show the network interface. The most important of the local resources is the Scratch-Pad Memory (SPM), as it serves two purposes. The SPM is an ideal memory and should be used as a fast local storage, allmost like a data cache but with its own address space. The second job of the SPM is to serve as the transmission and receive buffer for message passing between cores, as the NoC can copy data blocks between the SPM of different cores. To transfer data between cores the sender core programs its Direct Memory Access (DMA) unit to copy a range of its own SPM to another range in the receivers SPM, see figure 1.2. This copy operation is asynchronous and the cores can work on unrelated computations while the transfer happens.

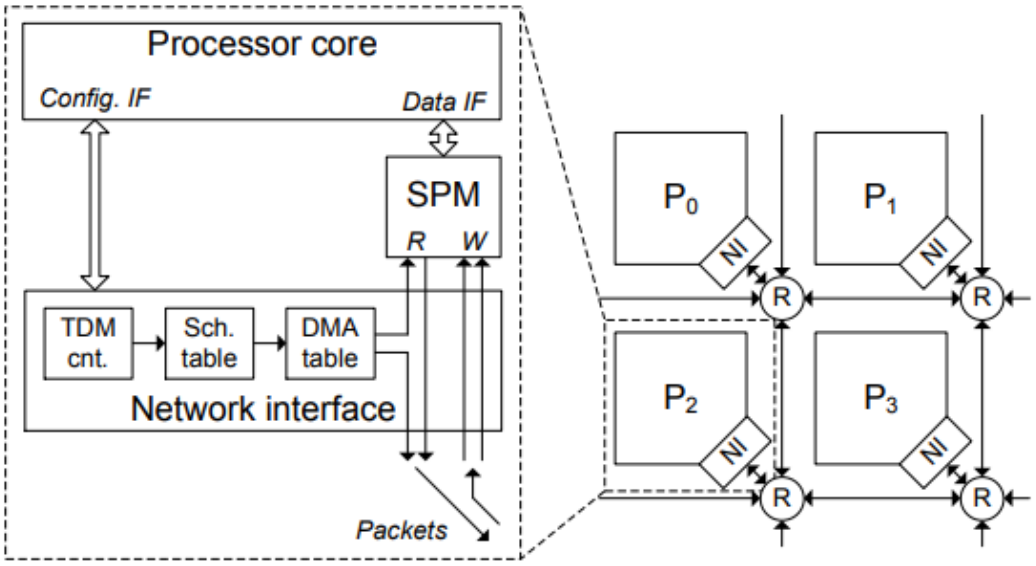

Figure 1.1: NoC routing of a four core configuration of Patmos [2].

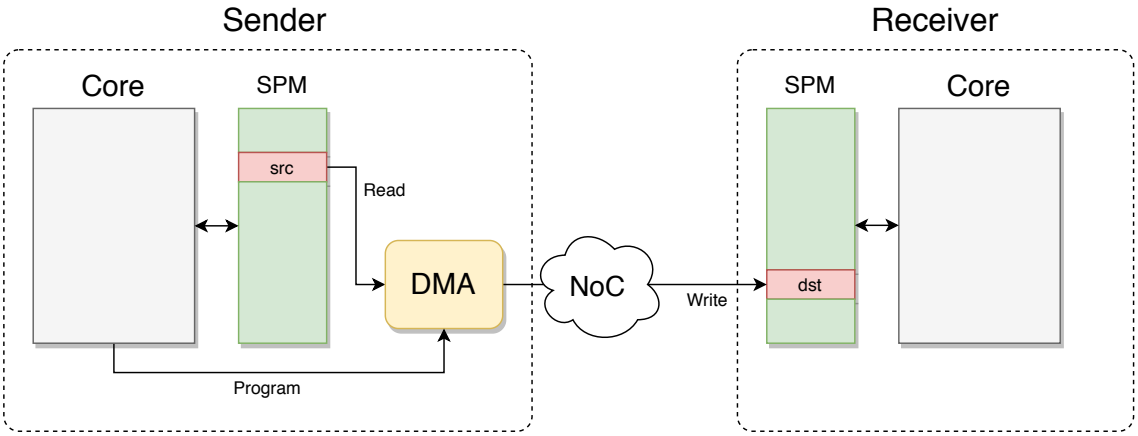

Figure 1.2: Patmos message passing.

1.2 Application

The application chosen for this project is a video decoder implemented as a actor system consisting of 41 actors, see figure 1.3. Each box represent one actor and each arrow at least one communication link. Queues with the same source and destination are collapsed for readability, but in actuality each queue has a single producer and a single consumer. Some actors at the top like parser_parseheaders sends tokens to the majority of actors below. There are three feedback loops towards the bottom of the figure formed by the actors with the motion prefix. Other than that the system is strictly feed-forward as all data travels from top to bottom. This application was chosen as it put many different loads on the system as well as varies greatly with the input data. Furthermore the application decodes MPEG-4 which was the first MPEG set of standards to feature reference software [5]. The application was chosen as it provides a substantial amount of diversity when it comes to actor performance characteristics. Some actors consist of many thousands lines of code while others use less than hundred lines. Some actors use barely any memory while others use many megabytes and some perform extensive computational operations such as discrete cosine transforms while others only rearrange data. The actors using most memory are those that access the framebuffer which stores the previous video frame. Because of this diversity we can utilize more parts of the Patmos architecture and evaluate if they could be potential bottlenecks. Furthermore the application varies substantially in the communication as the packets being sent from actor to actor range from single bits to chunks of hundreds of bytes with a rate varying from once per video frame to once per pixel. The actor system's topology is largely feed-forward but there exists a few feedback loops which in turn makes latency and bandwidth a concern. As the application is a decoder it inflates data causing the bandwidth requirement to increase towards the end of the pipeline. Another important aspect of this application is that the processing is input dependent. Different input stress different actors creating a correlation between the input and any potential bottlenecks.

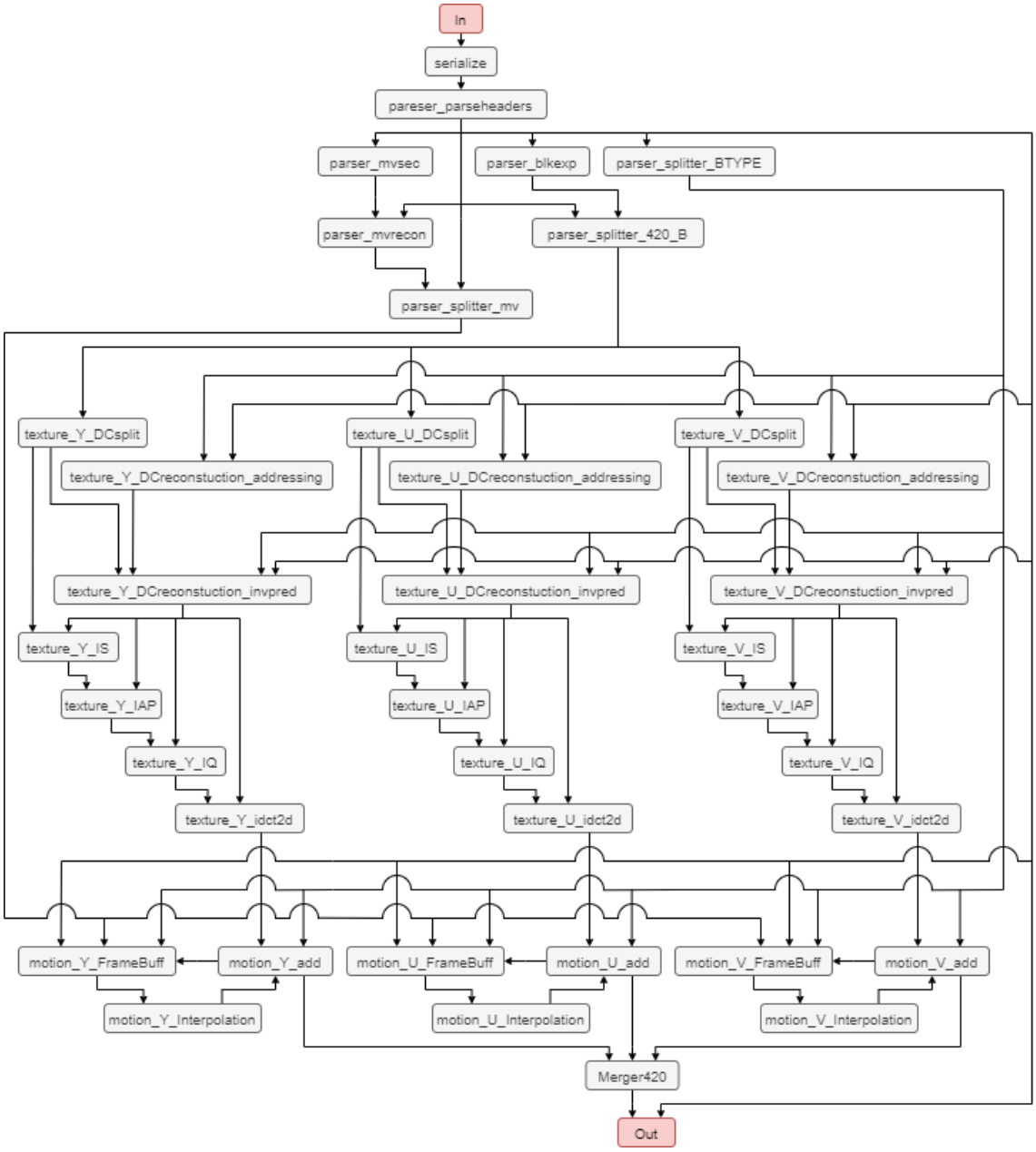

Figure 1.3: All actors of the streaming application.

1.3 Research Goal

Our goal is to perform a case study on the Patmos platform. More specifically how does Patmos perform when implementing a data driven application instead of a real-time application. The particular type of processor is interesting as it has great potential when it comes to scalability. In the quest for increased performance the only way forward seems to be improved parallelization [9]. This has lead to new ways of designing hardware, like processor arrays, that scale better as well as programming models such as actor systems [13]. In order to fully explore the platform and discovering the benefits and drawbacks relevant to our thesis, we will be implementing an actor based application. Since each actor is restricted to local

computations without the use of any shared memory, the actor system can be parallelized in a straightforward fashion. As the workload only makes use of message passing to communicate it is most suitable for an architecture featuring a processor array with a *network on chip* (NoC) [25]. Patmos is such an architecture, as can be seen in figure 1.1. As the Patmos architecture aims to allow the WCET of any application to be completely predictable, the average performance is lowered compared to other average case optimized systems. This design is not a perfect fit for the streaming application as it is input dependent and therefore difficult to predict.

1.3.1 Research Questions

In this study we try to answer the two following questions:
1) What are the benefits and drawbacks to the Patmos platform viewed from the unconventional perspective of implementation of a streaming application?
2) Is it feasible to automatically map an actor system to the Patmos architecture without loosing performance compared to a manual mapping?

1.4 Contributions

This master's thesis contributes to:

- Identifying the benefits and drawbacks to the Patmos architecture in the perspective of implementing a data driven application instead of a time driven application.
- Issues such as disabled NoC and Ethernet was found and then fixed by the the Patmos team. Also some configuration problems with the SPM size was solved.
- Pipeline for mapping any actor application to the Patmos platform. (easily adapted to any new application)
- Flexible mapping method using constraint programming that can easily be modified to account for more hardware limitations.
- Static analysis tools to explore and visualize an actor system as well as live analysis tools for measuring actor execution time, queue bandwidth and processor utilization on Patmos.

1.4.1 Division of Work

All of this project has been carried out with both authors present while employing the pair programming style. However at times when less significant code was to be implemented we did work in parallel. This led to Linus Gudmundsson being more responsible for the Python scripts utility and Jacob Canbäck being more responsible for the client program. However both authors were involved in these steps as well as later corrections and changes were made together.

When it comes to the report all sections have been rewritten and approved by both authors by making use of color coding for the sections. However Jacob spent more time on the

FIFO sections (sections 4.2 to 4.2.7) while Linus spent more time on the abstract, Constraint programming and Conclusion sections (sections 3.5, 3.5.1 and 6). However, as the report nears completion the time spent on these sections has also balanced out as additional edits are made.

1.5 Related Works

When it comes to related works for this thesis, quite a few works could be considered as related. Any project which implements a streaming application on an embedded system could be considered related. But as hundreds of such works exists we will limit ourselves to listing a few other works we consider to be the closest related to our thesis, as well as the papers we felt contributed to our thesis.

The most obvious one being the *Implementing a streaming application on a processor array* [16] work, which is another, older, thesis which implements the same streaming application on another embedded processor array. However, other than the actor program being the same there were not many similarities between theirs and our work. Our thesis separates itself from this one however as we use a time-driven platform which in turn leads to several new design choices, and we also research the mapping between the actors and the processor cores on a whole new level. Since they had none of these aspects in their thesis, and they had much more memory available to them, their work was not relevant beyond the actor system provided initially.

Another related work is the *Towards a time-predictable dual-issue microprocessor: The Patmos approach* [30] article which is written by the developer of Patmos. This paper discusses several concepts that Patmos consists of and enlightens the reader to the design choices made when designing Patmos. This work was mainly used to understand the Patmos platform, as one would expect from an article explaining the features of Patmos.

The work *Realizing efficient execution of dataflow actors on manycores* [10] describes how necessary it is to make use of manycore architectures and suggests efficient methods of doing so by creating actor systems in CAL. This is exactly the kind of actor system we have available to us in this thesis, as it was even supplied by our supervisor Jorn W. Janneck which is also one of the authors in that paper. Because of this we could make use of their paper to understand some of the decisions that were made when creating this actor system and how we could take advantage of them. This was one of the papers we used as our basis for the heuristics used when mapping actors to cores.

When it comes to the mapping problem one relevant work is *Orchestrating the Execution of Stream Programs on Multicore Platforms* [14]. This work targets the Cell architecture [33] which is a many-core processor and similar to our work it maps a software actor system to the hardware. However, there are a few differences when it comes to the software and hardware. The actor systems they experimented with was strictly feed-forward as dealing with feed-back loops was out of the scope of their work. Their hardware also differs considerably from ours as it is an Application Specific Integrated Circuit (ASIC) optimized for throughput as opposed to Patmos which is a real-time optimized micro controller implemented on an FPGA. Another important difference in the hardware is the topology as the Cell architecture consist of one general purpose processor and up to 16 co-processors with 256 KiB of local memory each for a total of 4 MiB of local memory. This memory is analogous to what we call SPM and

they had twice as much of it as we had external memory. This paper did achieve impressive results as they reached an avarage scalability of 14.7x over using just one of the co-processors when benchmarking a range of dataflow applications. Another important difference to our work is that they did not make use of constraint programming to solve the mapping problem, instead they used a solution based on integer linear programming to solve the packing problem of assigning actors to cores. This seem to have worked fine in their project as additional constraints such as memory limitations was not an issue as they claim that there is "ample memory available to hold the intermediate buffers" while we had to trim the memory usage of our application considerable just to make it run.

Chapter 2
Background

In this chapter we discuss the theoretical background behind Patmos and the Streaming application. We start of by describing the Streaming application as well as MPEG-4 more in depth and then do the same with the Patmos architecture where we discuss the memory and hardware.

2.1 Streaming Application

The streaming application was originally a MPEG-4 video decoder implemented in an programming language from 2001 called *Caltrop actor language* (CAL) [8] [7]. This language makes use of the concept of actors, which was originally introduced by Carl Hewitt as an approach of modelling intelligence as a society [11], and has since become widely used. The language was designed to create actor systems where the behaviour of each actor was controlled by a state machine and communication was performed using *First In First Out* (FIFO) queues. The original code was then compiled to C by a CAL to C compiler [34] before being provided to us. The actor system contains 39 actors communicating using 143 FIFO queues, see figure 1.3. A source file was provided for each of the actors as well as header files containing buffer size and token type for each of the queues. The C code was not executable as is. The FIFO queues had to be implemented and the API interface defined by macros using the C preprocessor.

2.1.1 MPEG-4

MPEG-4 is a standard which defines the compression of visual data and digital audio. It was created in 1998 and quickly became a standard used for audio and video coding [21] It was designated as such by the *Moving Picture Experts Group* (MPEG). Common uses include distributed CDs as well as media streaming. In this project we focus only on decoding compressed video as the compression algorithms are very complicated. However, it is essential to understand how the video is encoded in order to decode it.

2.1.2 Encoded Video

Video compression standards like MPEG-4 exploit the fact that a few following pictures in a video is often nearly the same. And even if they change the changes are often motion or just a few local changes. This makes it possible to represent most pictures as a delta applied to the previous picture. This delta can use substantially less memory than storing the actual picture independently. This is why encoding formats like MPEG-4 breaks up the video into two different types of frames with different memory cost, as explained in the following sections. This means that the amount of bandwidth and computations needed to decode the streamed video could have a significant variance over time.

2.1.3 Intra-coded Frames

Intra-coded Frames (I-frame) is the simplest and most memory intensive type of frame in an encoded video. They store the entire frame independently of other frames but they still use local compression similar to many popular image formats. There are three cases when I-frames are needed. The first case is at the beginning of the video as there is no previous frames to depend on. The second case is when the previous frames are available but the difference is too significant to encode as a delta such as a scene cut. The third case is access points that are stored at evenly spaced times, perhaps every 20th frames. Access points are needed for a few reasons. The video stream would be fragile if it had long sequences of delta encoded frames. One small error could propagate and destroy the video until another I-frame is processed by the decoder. This is especially important when streaming video over the internet as packets may be dropped or corrupted. Another important reason to have evenly spaced I-frames is to make the encoded video able to start playback at any frame index without having to decode too many delta frames. When starting playback at an arbitrary frame index the latest I-frame with the same or lower index is located and the decoder starts from there and has to decode the video until the desired index is reached before the playback can begin. When an I-frame is processed the bandwidth requirement at the front end of the decoder is substantially higher than the processing of frames represented as a delta on previous frames.

2.1.4 Predicted Frames

Predicted frames (P-frames) require prior decoding of previous frames. P-frames use less memory than I-frames and will not stress the front end of the decoder as much. They contain motion/displacement vectors to describe the changes between the current and previous frame. Some parts of the previous frame may not have to be changed at all such as a static background and this can be encoded efficiently. If some local area change a lot the P-frame can also contain image data to replace the pixels in that area. If two following frames are nearly identical the second frame can be represented as a tiny P-frame. This means that the front end could be mostly idle while there is no motion in a video. The back end of the decoder will always handle the same load no matter the memory usage of the encoded frames, this makes it tricky to load balance the various actors.

2.1.5 Color Space

The *luminance-bandwidth-chrominance* (YUV) encoding system encodes color as 3 components [15]. The Y component represents the luminosity (brightness). The other two components represents chrominance (color). U represents blue projection and V represents red projection. YUV encoding was invented to support color on black and white TVs while keeping the gray-scale/brightness signal for backwards compatibility. The alternative to YUV is the more modern and frequently used RGB color encoding. But YUV has some advantages when it comes to compression. Human eyes are a lot better at detecting differences in brightness than detecting differences in color [22]. This means that the Y component needs higher precision than the U and V components, something that is not possible with RGB color encoding. This is exploited in the video decoder/encoder by using chroma sub-sampling [23], see figure 2.1. The figure illustrates four different examples of reducing the amount of chrominance stored in a two by four pixel area.

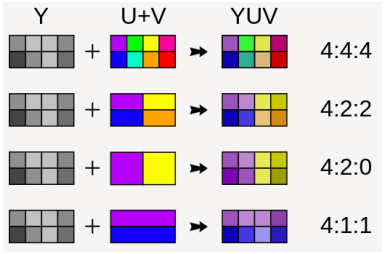

Figure 2.1: Chroma sub-sampling examples. The color of the video signal is sampled at a lower frequency than the brightness.

The output of the decoder is not regular RGB image data. It is encoded as YUV and has to be converted to RGB before the frame can be displayed. As both encoding formats are linear the conversion can be done by multiplying with a 3x3 matrix, see equation 2.1. The output of our particular streaming application is not directly observable as images. The image has to be computed from the 4:2:0 sub-sampled chunks produced by the Merger420 actor. The chunks consist of a 32x32 pixels of Y followed by 16x16 pixels of U and 16x16 pixels of V. It is just a larger version of the third row in figure 2.1. This conversion is not part of the decoder and we implemented it as a post process when visualizing the video.

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1.13983 \\ 1 & -0.39465 & -0.58060 \\ 1 & 2.03211 & 0 \end{pmatrix} \cdot \begin{pmatrix} Y \\ U \\ V \end{pmatrix} \tag{2.1}$$

2.2 Patmos

Patmos is a time-predictable platform optimized specifically for the WCET [29] instead of the average-case execution time. Such systems are an essential part of safety-critical systems such as transportation controllers that depend on guaranteeing that even the worst case will still be safe. This means that they need to be able to perform a reliable WCET analysis on their subsystems. This is where platforms such as Patmos becomes important. They allow the WCET to be analysed easily and with a high degree of confidence. This in turn results in a

much lower cost for safety-relevant applications due to both the reduced system complexity and the faster time-predictable code execution. The developers of Patmos stated in their article that the mission of the T-CREST project, which Patmos is a part of, was to develop tools for WCET analysis and building a platform that avoids unexpected delays [27]. But as stated earlier in the introduction, the real-time aspects are not useful for the streaming application.

2.2.1 Memory

The memory system used by the Patmos architecture features a large shared memory as well as a few types of local memories per core. These local memories include a SPM, a data cache memory, a stack cache memory and a instruction cache memory. This structure shares many similarities with more common processor architectures, such as the instruction and data caches for each core, but introduces new aspects such as the stack cache and SPM. Any access to the external memory goes through a time division multiplexed scheme called a memory wheel [29]. This leads to each core taking turns to transfer or fetching data in a round-robin fashion. As a result of this design the Patmos cores are simply unable to interfere with each others performance unlike most modern multi core processors. The purpose of the stack cache is to assist the data cache. It works like a window at the top of the stack and is implemented as a ring buffer. This helps with the WCET analysis as the stack access can be analyzed independently. The SPM is an ideal memory with two purposes. The first is to store general purpose data only accessed locally, which helps to avoid unnecessary access to the slower main memory. The second purpose is to asynchronously send and receive data packets to and from other cores through the NoC as explained earlier in the introduction. Unlike most multi-core processor architectures Patmos has no cache coherency protocol. This avoids several problems [3] but leads to the shared main memory being unsuitable for high performance communication as the data cache has to be invalidated in order to read something another core has written. Cache misses are especially costly on the Patmos architecture as the main memory is slow due to time multiplexed access. When accessing the external memory the data is transferred through the NoC which acts as multiple to one communication to the memory controller, see figure 2.2. The figure contains four cores at the top and the shared memory at the bottom. Between them there are two tries of registers. The first tree carrying signals from the cores to the memory controller features time multiplexers to divide access over time. The second tree is carrying signals (dotted arrows) from the memory controller to the cores.

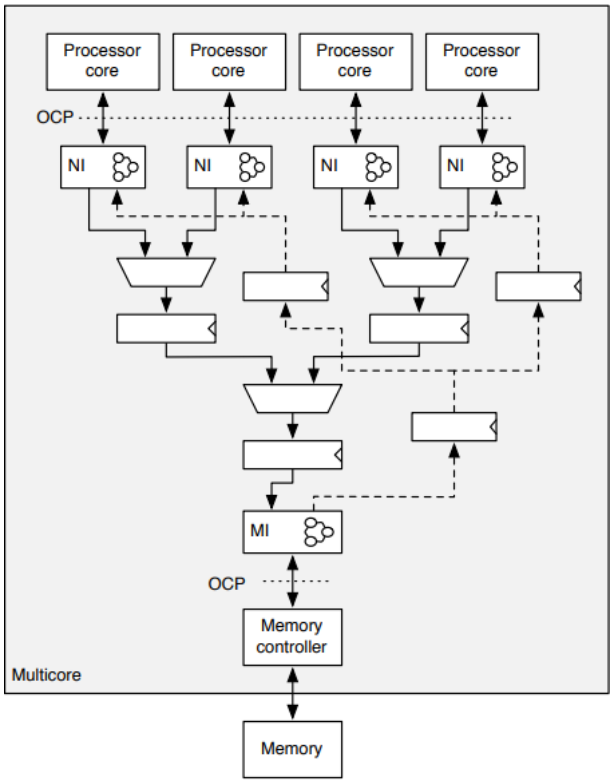

Figure 2.2: Time division-multiplexed NoC between the cores and the memory [2].

2.2.2 Core to Core Communication

The primary objective of the on-chip communication between cores is, as expected, to be time-predictable. The preferred way for the cores to communicate is to use their SPM to transfer data directly between cores using the NoC. In order to actually make such communications WCET analyzable the NoC is required to provide individually analyzable end-to-end connections. These can be implemented either by circuit-switching or by controlling the rate of the traffic flows. The NoC communication Patmos uses is implemented with the use of packet switching and source routing with the topology of a bi-torus [27]. This corresponds to the core layout being wrapped, for example in a 3x3 layout the bottom 3 cores would consider the top 3 cores neighbours. *Direct memory access* (DMA) driven block transfers are performed between the senders local SPM to the receivers SPM which enables asynchronous message passing between cores. Such communication between cores is similar to communicating with the external memory in the sense that they both use the NoC and both are time multiplexed, but the communication between cores is cheaper. This influenced our decisions when designing layouts to the point where we benchmarked how much remote queues affected the overall performance. More details about this can be found in section 5.

2.2.3 Hardware

In order to run the Patmos architecture a *Field programmable gate array* (FPGA) is needed as there is no *application specific integrated circuit* (ASIC) hardware yet. The FPGA used for this

project is the Altera DE2 [18], see figure 2.3. This model was recommended by the Patmos team. It comes with a lot of features such as audio, VGA graphics, Ethernet and a lot of miscellaneous IO such as LED's and switches. The memory available is 128 MB SDRAM, 2MB SRAM and 8MB FLASH as well as 3888 Kbits of embedded memory that can be used to implement for example cache memories or SPM. Patmos is able to use only a subset of these features. The memory system only use the 2MB SRAM for the shared memory and all the per core memories compete for the same 3888 Kbits of embedded memory modules. The primary IO devices used by Patmos is a USB for serial communication and the 100 *Mb/s* Ethernet port. For the miscellaneous IO Patmos enables the push buttons, some of the LED's and the 7-segment display. The configuration of Patmos is flexible. The amount of cores, memory sizes and enabled IO devices can be configured through a hardware config file. The amount of on chip memory used depends on the configuration. The amount of cache and SPM available per core decrease when the core count increase. We mainly used 4-core configuration with default cache sizes and 32KB of SPM per core. We also tried a 8-core configuration but the memory was only enough for 8KB of SPM per core.

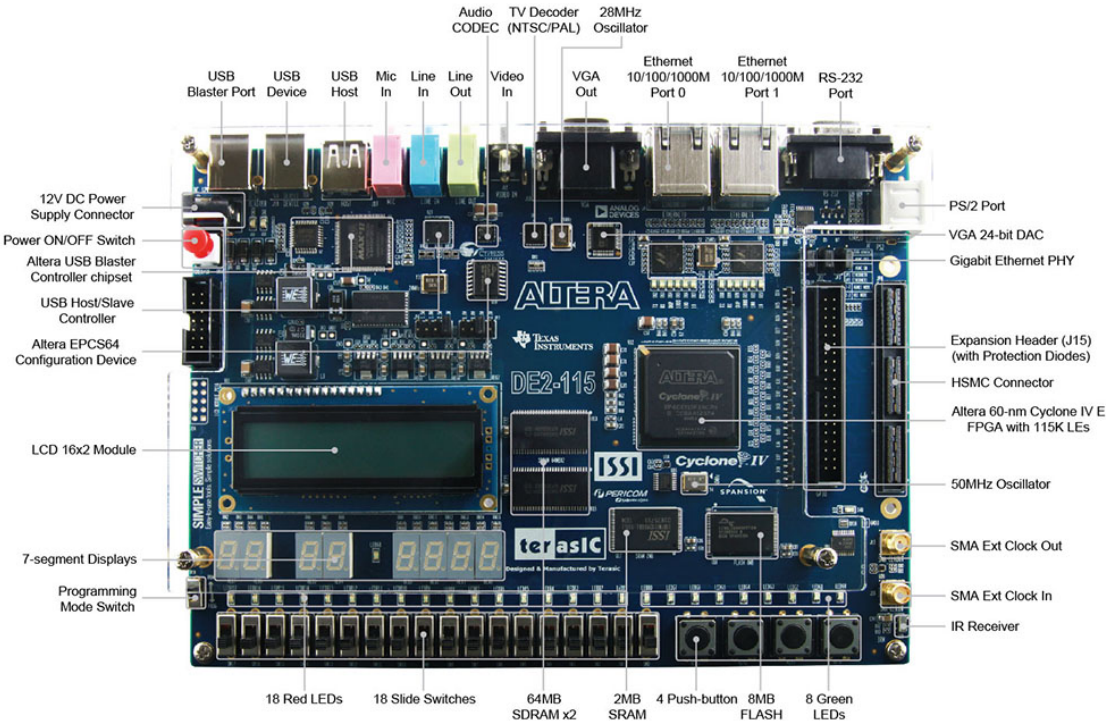

Figure 2.3: Layout of the Altera DE2 Board [1].

Chapter 3
Method

In this chapter we describe the methodology used to approach our problems. Section 3.1 describes the general approach used and the following sections fills in these general steps with more detail.

3.1 Overall Procedure

Our method consisted mainly of taking small steps forward by writing different test programs and analyze the outcome for any aspects we wished to understand. The first step of the project was static analysis on a multitude of the applications aspects. This includes plotting the communication network with actors and queues and finding out the properties of the queues such as token type and buffer size. After gathering an appropriate amount of data from our static analysis we implemented FIFO queues by making use of the preprocessor to inline different versions of functions in a similar fashion to the C++ templates used in the related work on the same application [16]. We then added some finishing touches and completed the x86 version of the streaming application. Then we moved on to Patmos, working on the provided FPGA. After overcoming some issues with the Patmos build, we had working multi-core Patmos hardware. Once this was done we learned how the NoC worked and decided on a direction for our final program. As we wished to have a modifiable layout we decided to make a small pipeline which built the Patmos program from the provided source code of the actors and some configuration files. This is when the main bottleneck started to reveal itself. Because of the applications high memory usage and the small amount of available memory on the Patmos device, all queue sizes, as well as the framebuffer resolution, had to be drastically reduced. This required substantial effort and reverse engineering, but at last we could fit the application on the Patmos hardware. Then we needed to upload the video to the Patmos device. We tried to convert the video into a C file by storing the raw data in an array and compiled it as part of the application. But once we made most of the actor attributes use SPM memory we were still unable to fit the entire video in external memory. Thus we decided

on the more reasonable approach of streaming the video over Ethernet. Luckily the Patmos team managed to help us make the NoC and Ethernet controller to work concurrently, which made this a viable solution. We could then finally implement a client application which both streamed the encoded data to the device as well as received and displayed the decoded data as a video. At this point the only thing left was to experiment with different ways of mapping actors to cores.

3.2 Static Analysis

As the streaming application features many thousands lines of code, we required some form of help tools or programs to analyze the application. In order to conveniently understand and display the applications current layout and structure several of said help tools were created by employing static analysis. After analysing such things as queue sizes, packet sizes, data-flow and the actor layout, we created several Python programs to display the data. The first Python program generated a data-flow graph which displayed the actor layout amongst the cores and how they communicate, see figure 3.1. This figure shows each actor as a colored box, where the color represents the core it resides on, connected to each other by arrows which represents a FIFO queue. The automatically generated graph is not meant to be readable at this size, see figure 1.3 for a manually refined version. To prevent this graph from becoming excessively cluttered we designed two other help programs. These programs displayed a list of all queues with a more in depth analysis of the communication and an analysis of the SPM usage respectively. We were able to calculate how much SPM usage an actor would require by parsing the attributes and buffers as these were the only factors which influenced the SPM used. Example output from the queue analysis, displaying parts of the output from the program, can be found in appendix B.1.

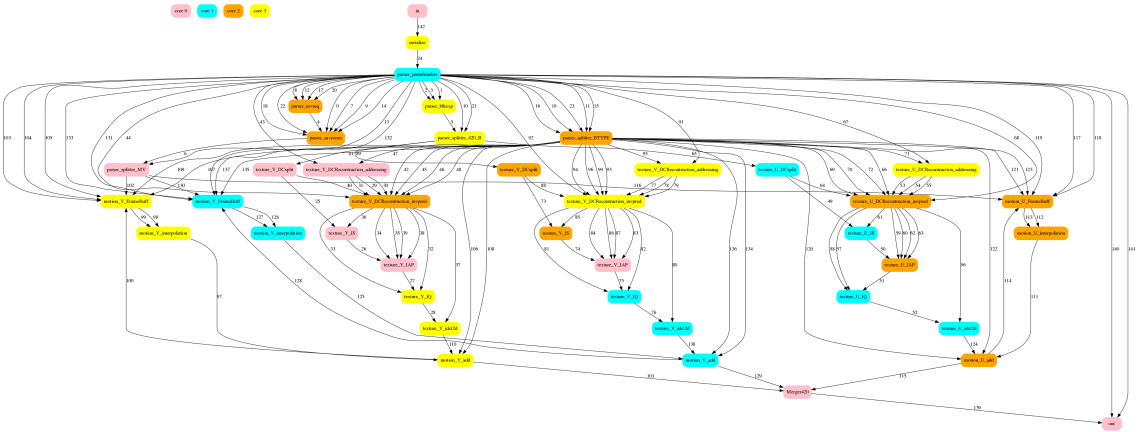

Figure 3.1: Auto generated graph of the data flow between the actors.

3.3 Code Generation

One of the most common development methods used in this project was code generation. To enable easy swapping of the actor layout which depicted actors resided on which core, a

configuration file containing the active layout was created. As this was the case large efforts went into parsing and modifying all of the actors through Python programs. Eventually this resulted in two pipelines, one for the x86 implementation and over for the Patmos implementation. The figure 3.2 demonstrates the x86 pipeline. The pipeline were divided into three phases, the adaptation phase, the generation phase and the compilation phase. The Adaptation phase only adapted the code by making changes such as removing comments to ease the parsing and lowering the framebuffer and attribute size. The Generation phase generated new actor files which made use of our new syntax, such as our FIFO queues. It also generated an actor updater which was used to call the actors update functions in a round-robin fashion. Finally the Compilation phase used the standard tools to link and compile the program.

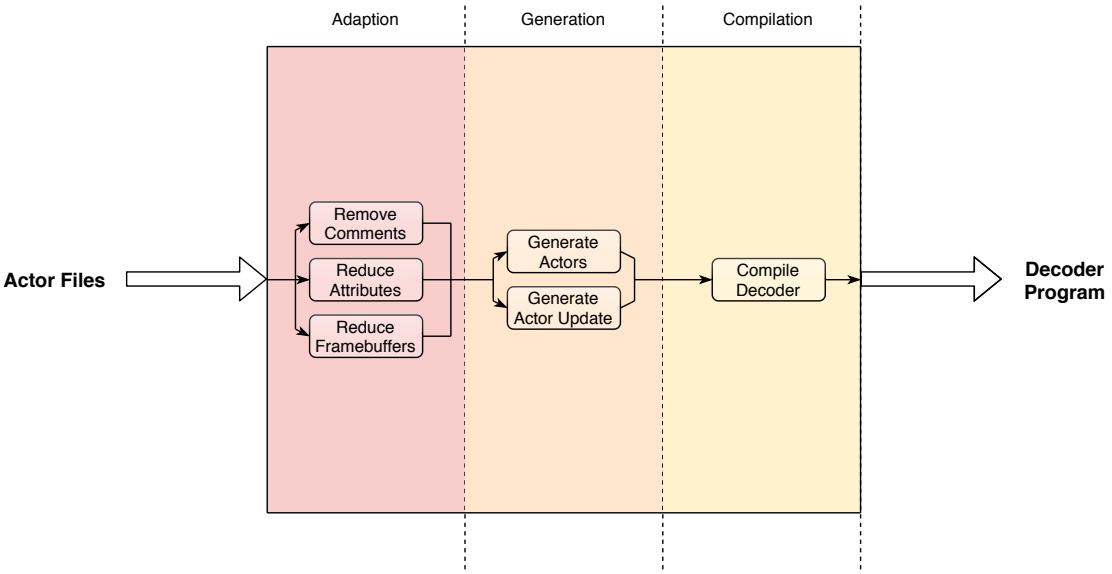

Figure 3.2: x86 Pipeline

The Patmos pipeline is similarly shown in figure 3.3. It is also divided into the three phases but has some differences to the x86 pipeline. The Adaptation phase is exactly the same. It takes the same actor files as input and makes the same changes as the x86 version. The Generation phase still generates the actors as well as the update scheduler, but now uses the Patmos syntax. It also makes use of two new programs. The first of these new programs allocates the attributes, the FIFO writers, the FIFO readers and the FIFO buffers in the SPM. As the SPM can be seen as a range of available addresses per core it was needed to determine where a variable would be stored. This was done in compile time to make sure it would not have any negative impacts on the performance. The second program modifies an already existing controller to conform with the active actor-to-core mapping. This controller handles the Ethernet IO for the application as well as some general scheduling. The Compilation phase is now carried out in the Patmos environment instead and make use of the Patmos commands found in the handbook [28] to compile and upload the program to the FPGA board.

Furthermore, while it is not a part of the actual pipeline the Patmos implementation also has a program to map the actors to the cores which can be run beforehand to generate the file containing these mappings. As this would only need to be run when the hardware configuration was changed it was not included in the pipeline.

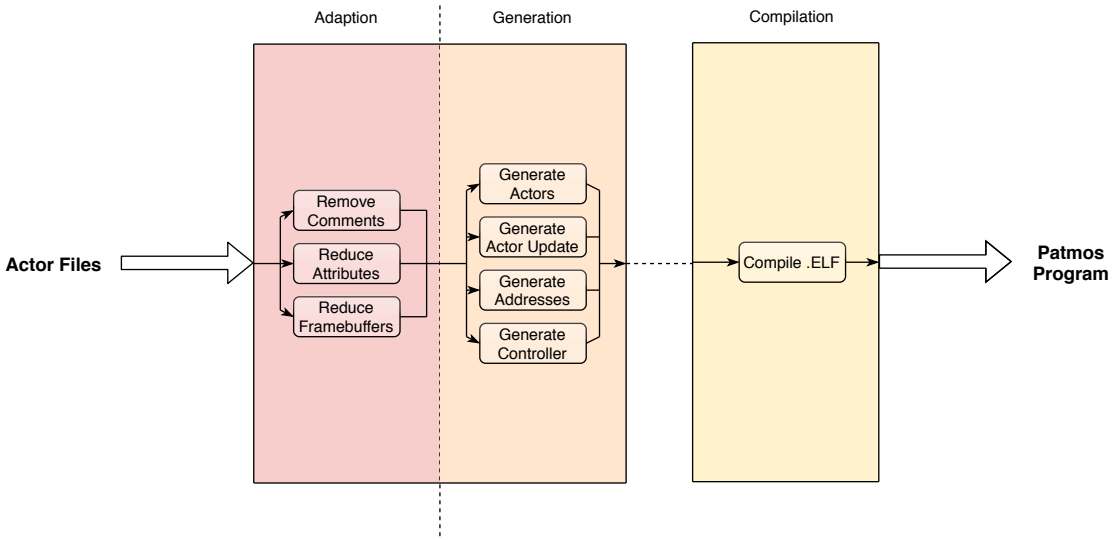

Figure 3.3: Patmos Pipeline

3.4 Multi-core Communication in the Application

As previously stated in section 2.2.2, the DMA message passing is completely asynchronous. Because of this any application that wants to make use of this message passing is required to take precautions such as implementing semaphores to prevent the actors from accessing a protected resource in improper order [4]. In our implementation of the streaming application we opted against using locks as these could impact the performance in a noticeable way, potentially even skewing the results. However we could still avoid having data hazards and race conditions present in the application. This was possible as the Patmos architecture guarantees that messages send with the NoC will arrive in the same order they are transferred. This is due to the fact that any FIFO queue writer/reader pair will produce the same in and output messages. As a result of this it was enough to allocate these writer/reader pairs, as well as the related buffers, in the SPM in a strict fashion such that there is no overlapping write or read addresses. This discovery lead to another issue, namely when to allocate these address ranges as well as how to spread them out amongst the cores, since the readers and writer layout amongst the cores directly corresponded to the actor layout. With the intention of maximizing performance we decided to compute the actor layout and the address allocation during compilation, as can be seen in the generation phase in figure 3.3.

3.5 Layout

When implementing an actor application on a multi-core platform there must always be some form of mapping between the actors and the cores. However this mapping can be done in many different ways. In this project we set out to investigate how this mapping would impact the performance, so the mapping was made to be as easily configurable as possible. This was implemented by letting the actor layout be read from a config file during

compilation. Originally we implemented a manual configuration which divided the actors amongst 4 cores, containing the parsing, Y, U and V actors respectively, see figure 3.4. In the figure the color of the boxes represent which core they are assigned to. This was chosen as we judged the Y,U and V actors respectively to require the same minimum SPM usage. Note that we say minimum as ideally the Y actors should have more SPM available than the U and V actors. The Y actors do four times more work but the minimum required SPM usage for queues is the same as the U and V actor, this was discovered when we were minimizing each actors memory usage as much as possible. The total SPM available per core was an issue at the time. Once this was operational, and we had managed to trim SPM usage even more which allowed for more freedom of movement, we decided to automate the actor placement and investigate the performance impact from this mapping.

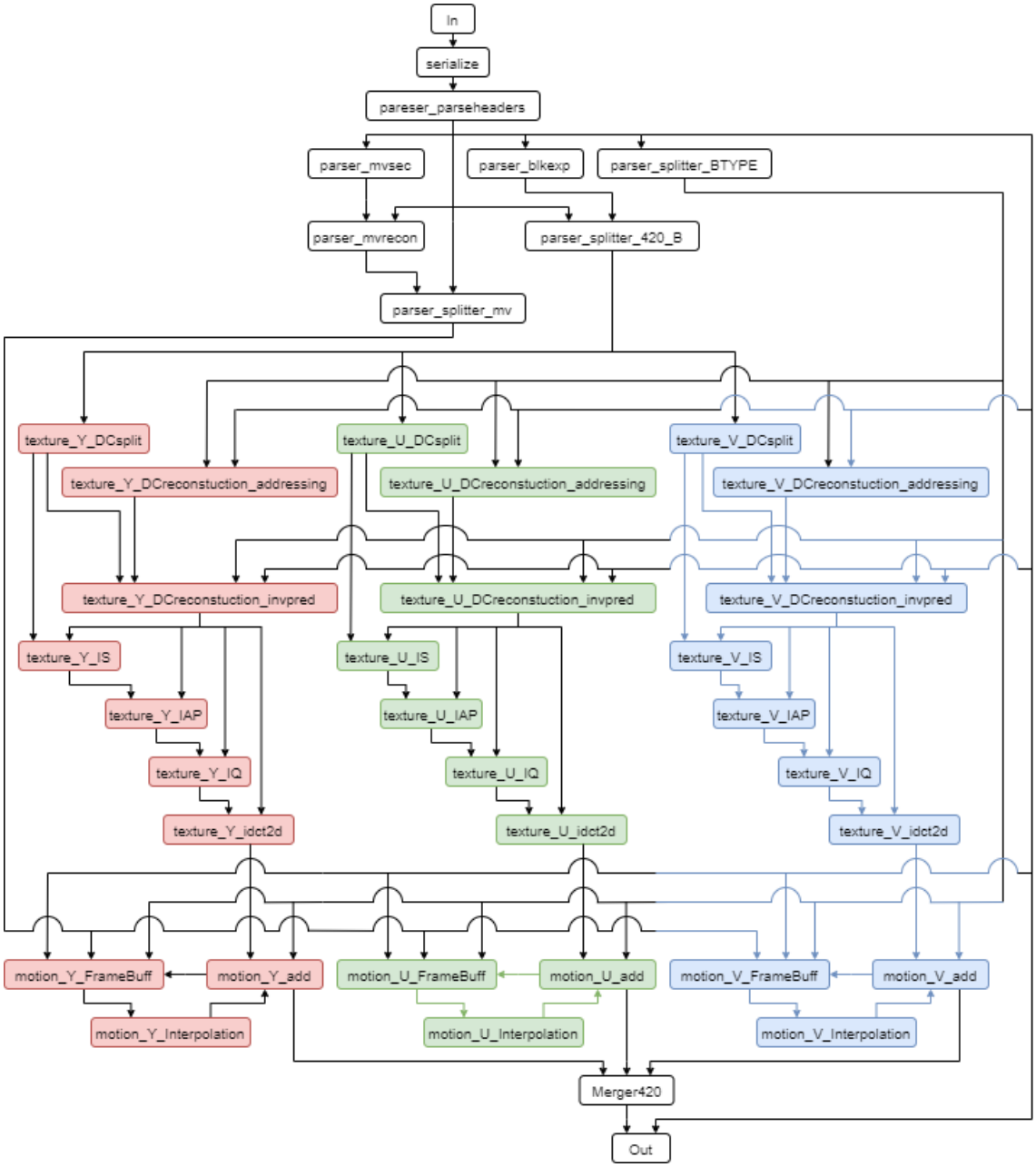

Figure 3.4: The manual actor layout.

3.5.1 MiniZinc Programming

In order to automate the actor mapping we had to decide on an implementation and a language to use. As we had interest in *Constraint Programming* (CP) and it seemed to fit the current problem well, we decided to use CP to build our program. When it comes to constraint programming there is no official standard language, and most solvers use their own modeling language [19]. We decided to use MiniZinc which is a open-source modeling language for combinatorial problems. These problems are then in turn solved by a backend solver [6]. Since we had previous experience with MiniZinc, and because the language allows one to smoothly adding additional constraints to a program, it seemed like an ideal choice.

We restricted the SPM usage for each core to the maximum amount allowed by the hardware, and then made use of constraints to define the relations between the actors on the core as well as the cores SPM usage. We let the program output an array with one core index per actor. A Python script was used to generate the data section used by the model. The data contained information about queue buffer sizes and which queues were connected to which actors. We now had a working base model that was capable of creating a functioning layout. The implementation could also work for other actor based programs. After rigorously testing edge cases to find and eliminate any residual bugs in the model, we moved on to different solve conditions (a solve condition in MiniZinc can for example be to minimize the total SPM usage) and heuristics in order to find more relevant layouts. While doing this we further expanded the model's input data with the execution time as well as the generated code size which we obtained with the help of live analysis.

Eventually we ended up with two heuristic and seven different solve conditions. One of the heuristics was to force any strongly connected sub-graphs of the actor system to be on the same core. The other one instead forced actors of similar "depth", which is decided by the longest path from the serialize actor, which is the first actor to use the input data from the in actor, to be on the same core. These heuristics were tested on the different solve conditions. These solve conditions are: Satisfy (no condition), Minimizing the peak execution time of the cores, Minimizing the total traffic between cores, Load balancing the traffic between cores, Maximizing the traffic between cores, Balancing SPM usage per core and Minimizing the code size used on each core. Most of these were chosen to observe how the aspect they are minimizing, or maximizing, affect the system as a whole and how different conditions impact them. This is discussed deeper in the result section 5.

3.6 Benchmarking

The benchmarking process had two purposes. The first was to evaluate the overall performance (frame rate) of different implementations where the actor to core assignment is the only variable. The other reason was to conduct live analysis to provide data to the MiniZinc algorithm such as execution times of the actors to enable load balancing optimization based on feedback. The host application was also briefly benchmarked to see if parallelization of the actor system scaled on the x86 host computer. Parallelization on the x86 platform was implemented by using pthreads. As this was made before the Patmos implementation we tried out several different scheduling and layout methods before eventually settling on the one which gave the greatest performance, which was to let each actor be on its own thread.

This is slightly unfair in the comparisons later on as the Patmos version instead had several actors on a core which it updated in a round robin fashion. However this was also tried on the x86 platform and had worse performance than giving each actor its own thread. These results can be found in table 5.7.

3.6.1 Actor Execution Time

The benchmark was done by measuring how many cycles were used to update each actor. An example pseudo code of how this was actually implemented can be found in listing 1. This code also describes the round robin style scheduling. The code ran on each core and the actor array contained the actors assigned to that core. Each actor also had its own cycle counter and update function. The cycles used by the actor were only counted if its state machine made any progress (useful computations). The actor update function returns false when the actor can not make progress due to limited space or token counts in the FIFO queues it interacts with. This allowed us to measure utilization and actor execution time independently of overall performance.

```
t = get_cycles()
start = t
useful_cycles = 0
while not done:
    for a in actors:
        progress = false
        while a.update():
            progress = true
        delta = get_cycles() - t
        if progress:
            useful_cycles += delta
            a.cycles += delta
        t += delta
total_cycles = get_cycles() - start
utilization = useful_cycles / total
```

Listing 1: Python pseudo code representing how the benchmarking was done.

Chapter 4
Implementation

This chapter describes the implementation details of our solution. It starts of with describing the memory usage as it is the main driving force behind almost all design choices we made. The following sections dive deeper into the FIFO queue implementation and why these design choices were made.

4.1 Memory Usage

The required memory for the original queues and variables was approximately 7 *MiB*. This was way too large to fit even on the Patmos processor external memory, not to mention the scratch pad memory. With the available scratchpad memory being 128 *KiB* equally divided amongst 4 cores and the external memory featuring a total of 2 *MiB*, it became apparent that we would need to trim the applications memory usage. After some research we found that the majority of the memory was used by 6 arrays: frameBuffer_n32v0, frameBuffer_n35v0, frameBuffer_n38v0, buf_n11v0, buf_n18v0 and buf_n25v0 which all related to frame buffers. As their memory usage depended on the maximum supported resolution the support for HD resolution was dropped in order to fit the application on Patmos. The maximum number of tiles supported was initially 121x79 tiles each using 768 bytes. With some extra margin allocated this added up to 6.62 *MiB*. Fortunately the supported resolution could be changed by modifying 3 actors: motion_Y_FrameBuf, motion_U_FrameBuf and motion_V_FrameBuf. Reducing the maximum number of tiles from 121x79 to 22x18, which is enough to decode the 4 smallest videos provided, we were now left with 317 *KiB* of memory to fit in the scratchpad memory. At this point we targeted the FIFO buffer sizes and immediately found much room for improvement. After decreasing the queue sizes the application was finally small enough to fit on the Patmos device.

4.2 FIFO Queues

A FIFO queue is a way of transferring data which arrive in the same order it was sent. It is also one of the, if not the most, important building block of any actor system. The decoder is no exception as it uses 143 FIFO queues. All of them are single producer single consumer type queues, which means that there is only one source placing items in the queue as well as only one reader processing queued items. The actors interacts with the queues using the operations found in table 4.1. These operations were used in the original actor files but were not originally implemented.

Table 4.1: FIFO operations used by the actors.

operation	description
SPACE	get max tokens that can currently be written
WRITE	put one token at the end
LOOP_WRITE	put N tokens at the end
TOKENS	get number of queued tokens
READ	read one token at some offset from front
LOOP_READ	read N token at some offset from front
CONSUME	remove N items from the front

4.2.1 Algorithm

A ring buffer is used to implement the FIFO buffer which the respective writers and readers interact with. Ring buffers are a simple fixed-size buffer of length L that wraps around itself to maintain the illusion of it being end-to-end connected. This is done to enable a design that assumes the buffer is infinite as long as some rules are followed. This does however mean that an array of tokens in the buffer can be split in two and appearing both at the end and the start of the buffer. A read and write index describes the queue state independently of the buffer size. Both the read and write indices are initially zero and increase during usage. The write index W may never overtake the read index R by more than the length of the buffer L. This means that all indices in the range between the R and W always map to a unique index in the ring buffer. The number of tokens T and the space S in the queue can be computed from W, R and L. An example state of a FIFO queue can be found in figure 4.1 where tokens $ABCDEF$ has been sent and the first 3 of them consumed. In this figure the row of squares and the ring to the right represent the same queue in two different perspectives, the yellow rectangles contain the read and write indices. Only the red and green boxes are actually stored in memory. The blue boxes represent consumed tokens. As for how the discussed alforithm was actually implemented, pseudo code can be found in listing 2 for anyone interested.

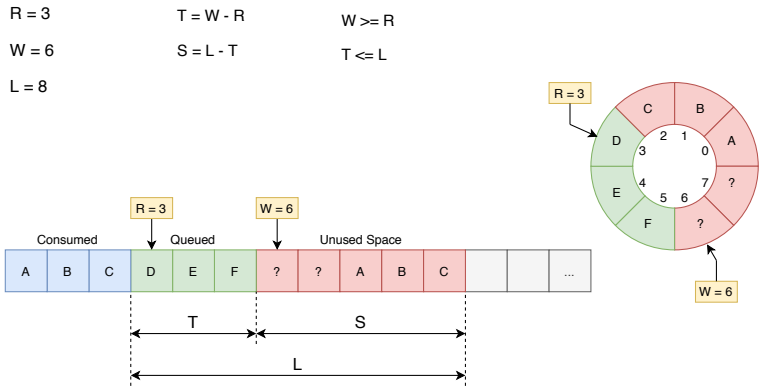

Figure 4.1: FIFO queue example.

```
L = 8
R = 0
W = 0
buf = [None] * L
def space():
    return L - (W - R)
def write(token):
    buf[W % L] = token
    W += 1
def tokens():
    return W - R;
def peek(i):
    token = buf[(R + i) % L]
    R += 1
    return token
def consume(n):
    R += n
```

Listing 2: Python code implementing the core FIFO queue algorithm.

4.2.2 Cross Core FIFO Queues

The design of the FIFO queues is influenced by the hardware design. Communication directly between different cores can only happen by copying a range of scratchpad memory from the sender core to another range of scratchpad memory at the receiver core. The granularity of the transfer is 32 bit blocks and they are written atomically. It is possible to write but impossible to read other core's scratchpad memory directly. The copy operation is initiated by the producer core by programming its DMA unit to copy the memory. The DMA unit then proceed to copy one token at a time to the receiver memory asynchronously. This

allow the producer and consumer core to do computations while copying. The DMA unit can only work on one copy operation at a time so it makes sense to send large number of tokens at the same time and do computation until the DMA unit is no longer busy. Since the communication is completely asynchronous each FIFO queue can only have one single producer while keeping the implementation lock less to avoid data race conditions.

4.2.3 Shadow State

The algorithm described in section 4.2.1 works fine when the producer and consumer actor are located on the same core but there will also be actors on different cores. In a multi-core setup the ring buffer is located at the consumer core. The state of the queue is completely described by W, R and L. L is just a constant but W and R are dynamic. The producer and consumer located on different cores need access to each others variables. The producer controls W and the consumer controls R. In order to allow the producer to read R it needs a local shadow of that variable. The same thing applies for the consumer who needs a local shadow of W. The shadow variables lag behind their counterparts but can still be safely used in their place as long as everything is done in the correct order. The producer must update the W shadow at the consumer core after the buffer at the consumer core has been written to. The consumer must update the R shadow at the producer core after the tokens have been consumed. This scheme is similar to how the *Transmission control protocol* (TCP) works where the shadow updates are the analog to acknowledge messages moving the transfer window forward [26]. An example of the FIFO protocol can be found in figure 4.2. This figure illustrates a bulk transfer of 3 tokens including shadow state update. The space between the vertical lines represent the link between two different cores. The horizontal dotted lines separate the state over time, time flows from top to bottom. The tilt of the arrows between the vertical lines illustrates the latency of transferred packets. Yellow squares represent the write index and purple squares the read index. The rows of squares to the right represent the a ring buffer length 4. Blue slots of the buffer represents pending received tokens, green consumable tokens and red unused space. The pseudo code of the modified algorithm with shadow state can be found in listing 3. The functions are divided in two blocks, the first three are only accessed by the producer and the last tree are only accessed by the consumer. The function dma_write can not be defined in Python but are meant to illustrate the programming of the DMA. Writing data not residing on the same core must be done using this function. Notice that the write function is split up in loop_write and produce. This enables optimizations discussed later.

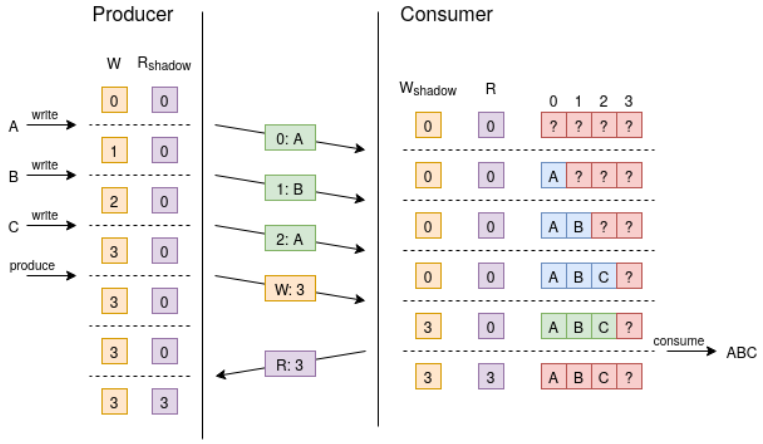

Figure 4.2: FIFO protocol example.

```
L = 8             # producer and consumer has its own copy
W = 0             # resides on producer core
W_shadow = 0      # resides on consumer core
R = 0             # resides on consumer core
R_shadow = 0      # resides on producer core
buf = [None] * L # resides on consumer core
def dma_write(dst,src,offset,count)
# only accessed by producer:
def space():
    return L - (W - R_shadow)
def loop_write(tokens):
    dma_write(buf,tokens,W % L,len(tokens))
    W += len(tokens)
def produce():
    dma_write(W_shadow,W,0,1)
# only accessed by consumer:
def tokens():
    return W_shadow - R;
def peek(i):
    token = buf[(R + i) % L]
    R += 1
    return token
def consume(n):
    dma_write(R_shadow,R,0,1)
```

Listing 3: Multi-core FIFO queue pseudo Python code.

4.2.4 Bulk Transfer

When sending one token at a time half of the bandwidth is used to update the W shadow assuming the tokens are the same size as the counters. This is not necessary for bulk transfer. If loop_write is called with many tokens as input the bandwidth is better utilized. For large chunks of tokens this almost reduces the bandwidth requirement by half. The only possible drawback is that the consumer may have to wait slightly longer. Another important reason to separate the update of the W shadow variable is that the DMA unit is busy for a while after calling loop_write. The most efficient way to send data is to call loop_write with a large chunk of tokens, do some computation and then call produce. By doing this it is less likely that the producer core is wasting time waiting for the DMA unit to be available in the produce function.

4.2.5 Overflow

One concern of this implementation is that the read and write indices overflow and breaks the algorithm. As the counters use 32-bit unsigned integers this happens when approximately 4 billion tokens have been sent. This correspond to 4 *GiB* assuming the case where on token is just one byte. The queue with the highest throughput is logically the last one that transfers the tiles of the video. In order for it to overflow it would have to saturate the output bandwidth of the 100 *Mb/s* Ethernet interface for 327 seconds. The largest of the videos provided produces a bit-stream of tiles that is just 265 *MiB*. Since the queues are reset before decoding a video the overflow problem, while definitely being solvable, will not cause any problem unless much longer videos are decoded which is not the case in this study.

4.2.6 Initialization

The FIFO implementation for the Patmos platform requires an initialization stage where the SPM addresses are decided for all components of all queues. The components involve the ring buffer, write index and read index for each queues as well as shadows of the indices for queues between actors assigned to different cores. The allocation of all components can be done in compile time instead of execution time by defining the addresses statically. We implemented this as a Python script generate_spm_addresses.py that generates a file, spm_addr.h, containing the addresses of all components. The file is then used by the C pre-processor. The Python script does this by reading a configuration file core.cfg that contains the layout assigning cores to actors and fifo_sizes.txt that contains the queue lengths. It then makes use of one memory arena allocator per core as the SPM use local address spaces. The only initialization required at run-time is to make sure that all read and write indices and their shadows are zero. This was done by simple clearing the entire scratchpad memory.

4.2.7 Power of 2 Optimization

When executing the peek, write and loop_write functions, see listing 2 and 3, The modulo (%) operation is used. This operation is significantly expensive as it is emulated by multiple instructions on the Patmos hardware. This operation would have a execution time comparable

to that of a integer division. This obviously is not feasible to do for every token produced and consumed as the performance impacts would be severe. However this is an issue that has been a research area a long time and has an elegant solution. The solution is to simply let the ring buffer length L be a power of 2. By doing so, the operation can be completed using a single bit-wise AND instruction with a constant mask [35]: $L = 2^N \implies x \bmod L = x\&(L-1)$.

Chapter 5
Results and Discussion

This chapter contains figures depicting the result we obtained by implementing the streaming application on Patmos, which is then followed by a small discussion which explains why these results are significant or interesting as well as our theories as to why we obtained these results.

5.1 Setup

In the following benchmarks the video akiyo_cif_s.m4v, which was provided alongside the original actor code, was used for all experiments. It was chosen as it was the smallest video of the provided videos. It is a video featuring an almost static background with a talking reporter. To conduct the experiments a host computer running Ubuntu Linux is necessary as it is the only operating system officially supported by Patmos. Three different hardware configurations of Patmos was synthesized for the FPGA: a standard 4 core version with 32 *KiB* of SPM per core used for most testing, a 8 core version with 8 *KiB* per core and a single core version with 64 *KiB* SPM. Furthermore, there was one additional subversion of the 4 core setup which made use of double FIFO queue sizes. Which setup was used for the experiment is presented alongside the result for clarity in the section below. For the testing process it is necessary to have the FPGA connected with an Ethernet cable, and a setup script, provided by the Patmos team, must be run on the host computer to allow the Patmos platform to connect to the host.

One of our solve conditions makes use of a table of actor execution times to determine the actor mapping. These execution times were obtained on a completely local single-core configuration on the Patmos platform with 64 *KiB* of scratchpad memory. This means that the NoC is not affecting the performance and there is no transfer bottlenecks. See figure 5.4 for the results. The benchmark is a live analysis that depends on the input data. It can be seen that the actor parser_parseheaders takes many cycles to compute on average. We suspect that this is due to the gigantic switch statement in the source code as it compiles to approximately

200 *KiB* of machine instructions, way more than the 4 *KiB* instruction cache.

The x86 machine had the 4-core CPU Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz.

5.2 Performance Based on Layout

The applications performance can be significantly improved or reduced depending on the chosen actor layout. This can be seen below in figures 5.1, 5.2 and 5.3 which shows the average time it took to produce a single frame for the different layouts generated from the solve conditions (see listing 4) and heuristics on different configurations. Figure 5.1 uses the standard configuration for this project, which is 4 cores with 32 *KiB* SPM per core. This configuration is also used in figure 5.3 also uses this configuration but now with double queue sizes for all FIFO queues, which in turn limits freedom for the constraint solver. As can be seen from these two figures (figure 5.1 and 5.3), the frame times are far lower for all the solve conditions when the queue sizes are increased. This means that the applications performance benefits greatly from the increase in queue size. In this example we only doubled all FIFO queues but some FIFO queues benefit more than others from the increased size. We discuss more about this in the future works section 6. In figure 5.2 an 8 core configuration with 8 *KiB* SPM per core is used. This is the maximum amount of SPM we were able to allocate for an 8 core configuration. As can be seen the performance drops substantially compared to the 4-core configuration. This stems from two factors. The first is when increasing the number of cores, there is also an increase of the slots in the time-multiplexing and the second is that the decrease in SPM in turn leads to lower FIFO queue sizes and less freedom when mapping actors to cores. For this project the lower SPM is by far the greater factor, but if this was implemented on a platform with an excess of memory we theorize that the time-multiplexing would become the dominant bottleneck. The tables 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6 shows the amount of idle cycles, the amount of cycles where no progress could be made. As can be seen these numbers vary greatly with different solve conditions, heuristics and hardware configurations. Furthermore when comparing the idle cycles to the frame times which is shown in graphs 5.1, 5.2 and 5.3 it becomes clear that the idle time has a huge impact on performance. The more idle time the less performance. Because of this we would like to balance the operations as much as possible between the cores. An example of this can be seen in our depth constrained heuristic. Our depth constrained heuristic cut the actor system into vertical slices and divided these slices between the cores. This was intended to make the actors be able to work in parallel as much as possible by cutting any long chains of dependency between the actors. This heuristic benefited the performance for every layout as can be seen in figure 5.1, where the yellow bars represent the depth constrained frame time, the red bars represent the strongly connected sub graph heuristic and the blue bars represent the original frame times. As mentioned in section 1.1.2 the cores form a bi-torus topology. This means that you cannot have any number of cores that cannot form a rectangle, such as prime numbers. For example you could not have two, three or five cores, but you could have four, six or eight cores.

The single-core configuration of Patmos had a frame time of 1486 *ms*. Since there were only one core to map to there were no solve conditions or heuristics applied as they would not do anything.

I. Load balancing the actor execution time amongst the cores.

II. Satisfy (no condition).

III. Balancing SPM usage per core.

IV. Minimizing the code size used on each core.

V. Load balancing the traffic between cores.

VI. Maximizing the traffic between cores.

VII. Minimizing the total traffic.

Listing 4: The seven solve conditions. These will be referred to by their respective roman numeral in the coming figures.

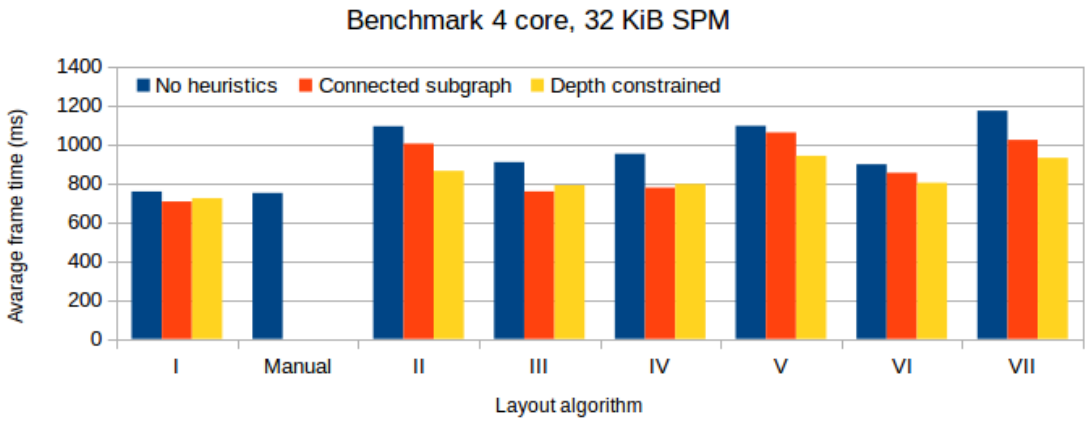

Figure 5.1: Frame Times for a 4 Core build with 32 *KiB* SPM per core.

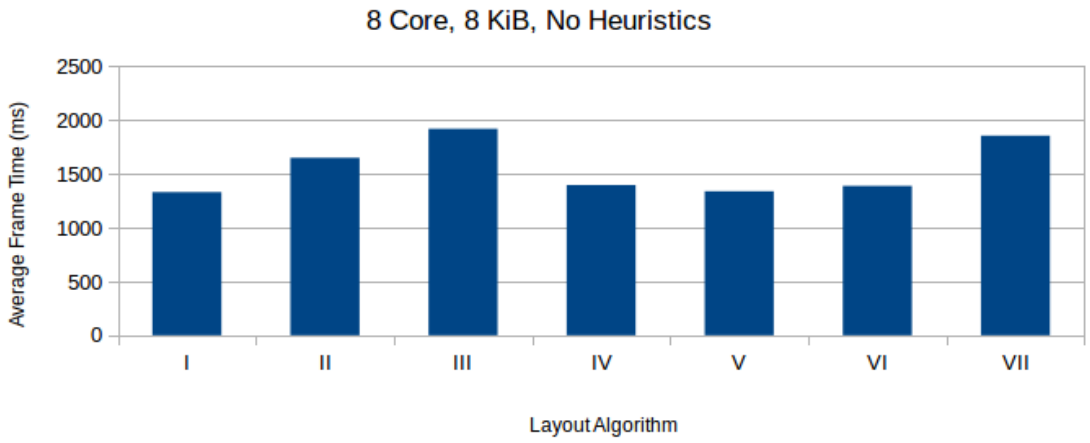

Figure 5.2: Frame Times for a 8 Core build with 8 *KiB* SPM per core.

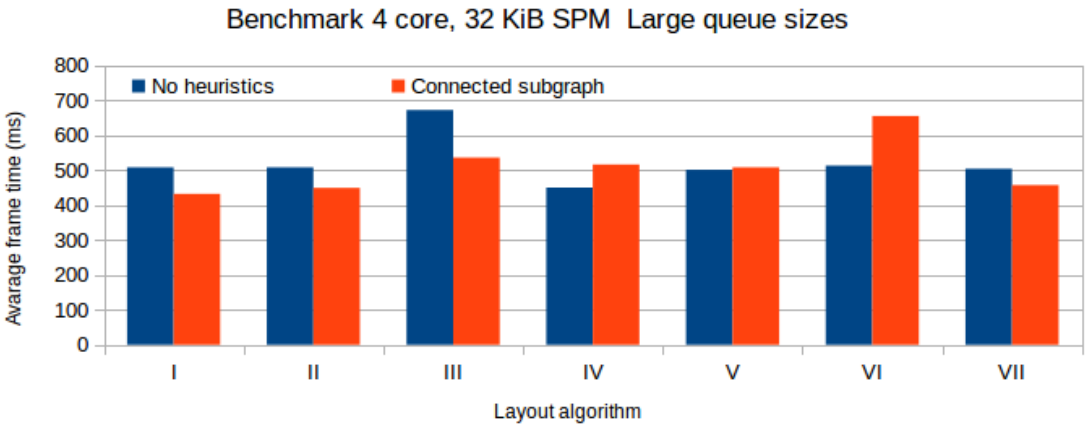

Figure 5.3: Frame Times for a 4 Core build with 32 *KiB* SPM per core with doubled FIFO queue sizes.

Table 5.1: Table of idle cycles in percentage for a 4 core layout with no heuristics.

Core Idle time (%) with 4 cores and no heuristics for each solve condition.								
Core ID	**I**	**Manual**	**II**	**III**	**IV**	**V**	**VI**	**VII**
1	47.00	70.00	86.00	68.00	20.00	28.00	70.00	55.00
2	38.00	30.00	52.00	77.00	57.00	93.00	91.00	100.00
3	34.00	68.00	20.00	20.00	59.00	47.00	35.00	81.00
4	53.00	13.00	91.00	49.00	79.00	77.00	6.00	21.00
Average idle time	43.00	45.25	62.25	53.50	53.75	61.25	50.50	64.25

Table 5.2: Table of idle cycles in percentage for a 4 core layout with the actor sub-graphs forcefully made local.

Core idle time (%) with 4 cores with connected sub-graphs for each solve condition.							
Core ID	**I**	**II**	**III**	**IV**	**V**	**VI**	**VII**
1	50.00	62.00	56.00	59.00	27.00	66.00	40.00
2	41.00	76.00	15.00	46.00	93.00	49.00	89.00
3	21.00	11.00	65.00	32.00	76.00	55.00	92.00
4	35.00	82.00	35.00	38.00	45.00	20.00	22.00
Average idle time	36.75	57.75	42.75	43.75	60.25	47.50	60.75

Table 5.3: Table of idle cycles in percentage for a 4 core layout with the depth constraint heuristic.

Core idle time (%) with 4 cores with each solve condition being depth constrained							
Core ID	**I**	**II**	**III**	**IV**	**V**	**VI**	**VII**
1	50.00	83.00	50.00	64.00	33.00	26.00	34.00
2	33.00	69.00	33.00	38.00	89.00	45.00	28.00
3	34.00	32.00	34.00	33.00	32.00	72.00	69.00
4	41.00	22.00	41.00	39.00	66.00	27.00	87.00
Average idle time	39.50	51.50	39.50	43.50	55.00	42.50	54.50

Table 5.4: Table of idle cycles in percentage for a 8 core layout with no heuristics.

Core idle time (%) with 8 cores and no heuristics for each solve condition.							
Core ID	**I**	**II**	**III**	**IV**	**V**	**VI**	**VII**
1	75.00	85.00	87.00	83.00	82.00	83.00	87.00
2	86.00	80.00	87.00	39.00	55.00	36.00	83.00
3	35.00	76.00	10.00	73.00	36.00	73.00	95.00
4	65.00	69.00	72.00	46.00	74.00	89.00	62.00
5	78.00	84.00	81.00	34.00	83.00	40.00	93.00
6	34.00	71.00	90.00	72.00	84.00	86.00	55.00
7	78.00	8.00	86.00	85.00	38.00	20.00	95.00
8	43.00	73.00	49.00	73.00	60.00	76.00	10.00
Average idle time	61.75	68.25	70.25	63.13	64.00	62.88	72.50

Table 5.5: Table of idle cycles in percentage for a 4 core layout with no heuristics but double FIFO queue sizes.

Core idle time (%) with 4 cores with double FIFO queue sizes and no heuristics for each solve condition							
Core ID	**I**	**II**	**III**	**IV**	**V**	**VI**	**VII**
1	42.00	56.00	69.00	19.00	61.00	51.00	61.00
2	16.00	10.00	58.00	48.00	8.00	44.00	53.00
3	45.00	33.00	65.00	19.00	42.00	18.00	19.00
4	13.00	40.00	6.00	10.00	21.00	8.00	11.00
Average idle time	29.00	34.75	49.50	24.00	30.25	30.25	36.00

Table 5.6: Table of idle cycles in percentage for a 4 core layout with the actor sub-graphs forcefully made local and the FIFO queue sizes doubled.

Core idle time (%) with 4 cores with double FIFO queue sizes and connected sub-graphs for each solve condition							
Core ID	**I**	**II**	**III**	**IV**	**V**	**VI**	**VII**
1	29.00	6.00	23.00	25.00	25.00	70.00	61.00
2	8.00	47.00	42.00	13.00	11.00	5.00	16.00
3	35.00	45.00	15.00	38.00	37.00	68.00	22.00
4	11.00	15.00	63.00	58.00	57.00	43.00	15.00
Average idle time	20.75	28.25	35.75	33.50	32.50	46.50	28.50

5.3 Actor Execution Times

Below in figures 5.4 and 5.5 the actor execution time for the single-core Patmos and x86 implementation respectively are shown. In these images it is shown that parser_parseheaders is the heaviest actor on Patmos while it instead is the out actor on the x86 implementation. We theorize that the parser_parseheaders actor has such a long execution time on the Patmos solution since the code size is too large to fit in the instruction cache. However as we do not know how to show the cache performance numbers for a run we cannot prove this. As the actor access data in an almost sporadic order this leads to that almost every call become a cache miss which force the actor to access the slower external memory. On the x86 solution a much larger part of this actor fits inside the instruction cache without any issues thanks to the high amount of available resources. The x86 platform also features branch prediction [31] that could likely help as well. The out actor takes up considerable more time on x86 than on Patmos as it writes a bit-stream to an actual file, while Patmos makes use of Ethernet to stream it back to the host instead. The speed of the x86 CPU compared to its hard-drive is much larger than the relative speed of a Patmos core compared to the Ethernet. We could have converted the x86 solution to use the Ethernet for the IO instead to make the tests more fair.

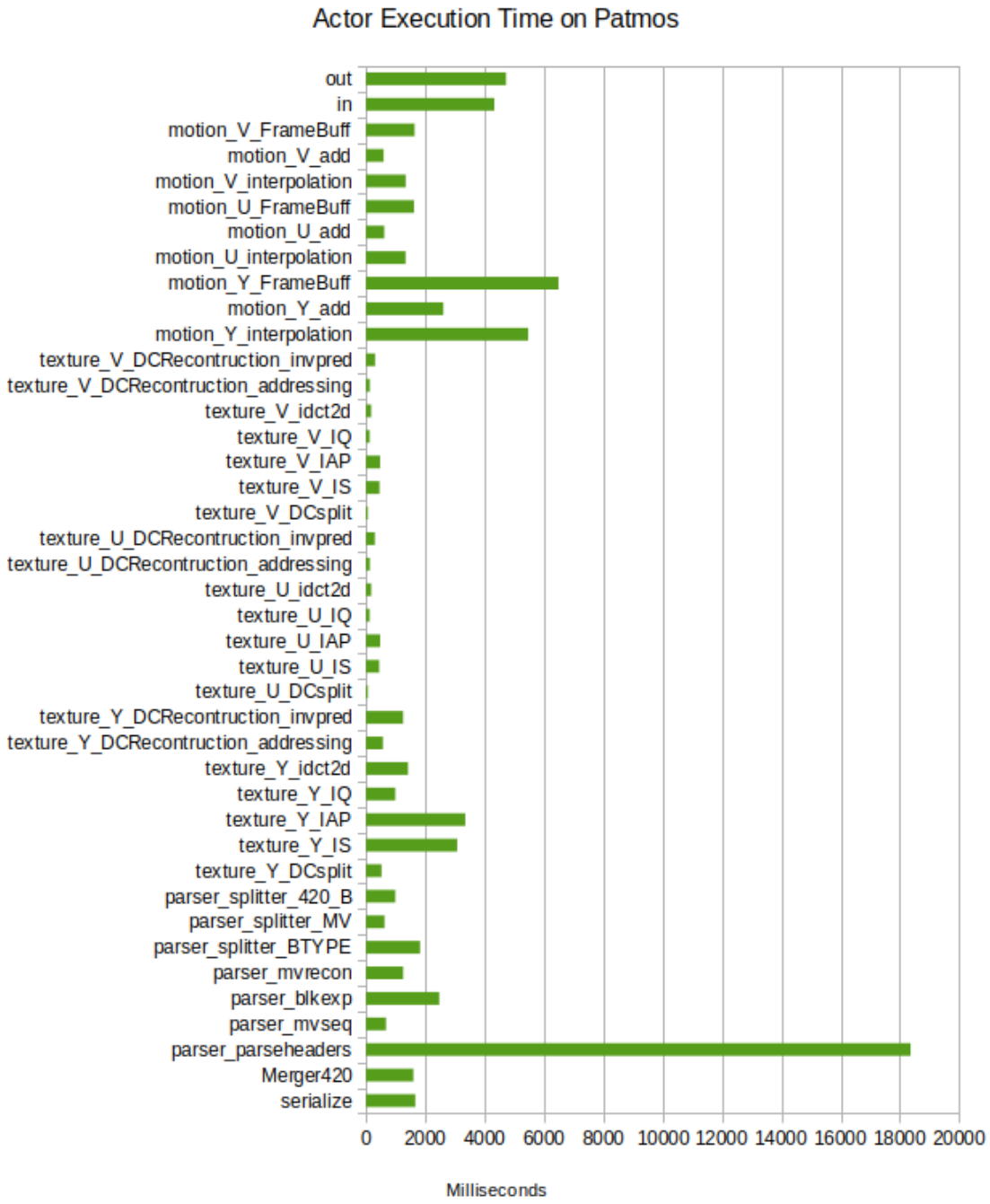

Figure 5.4: Actor execution times on the Patmos platform.

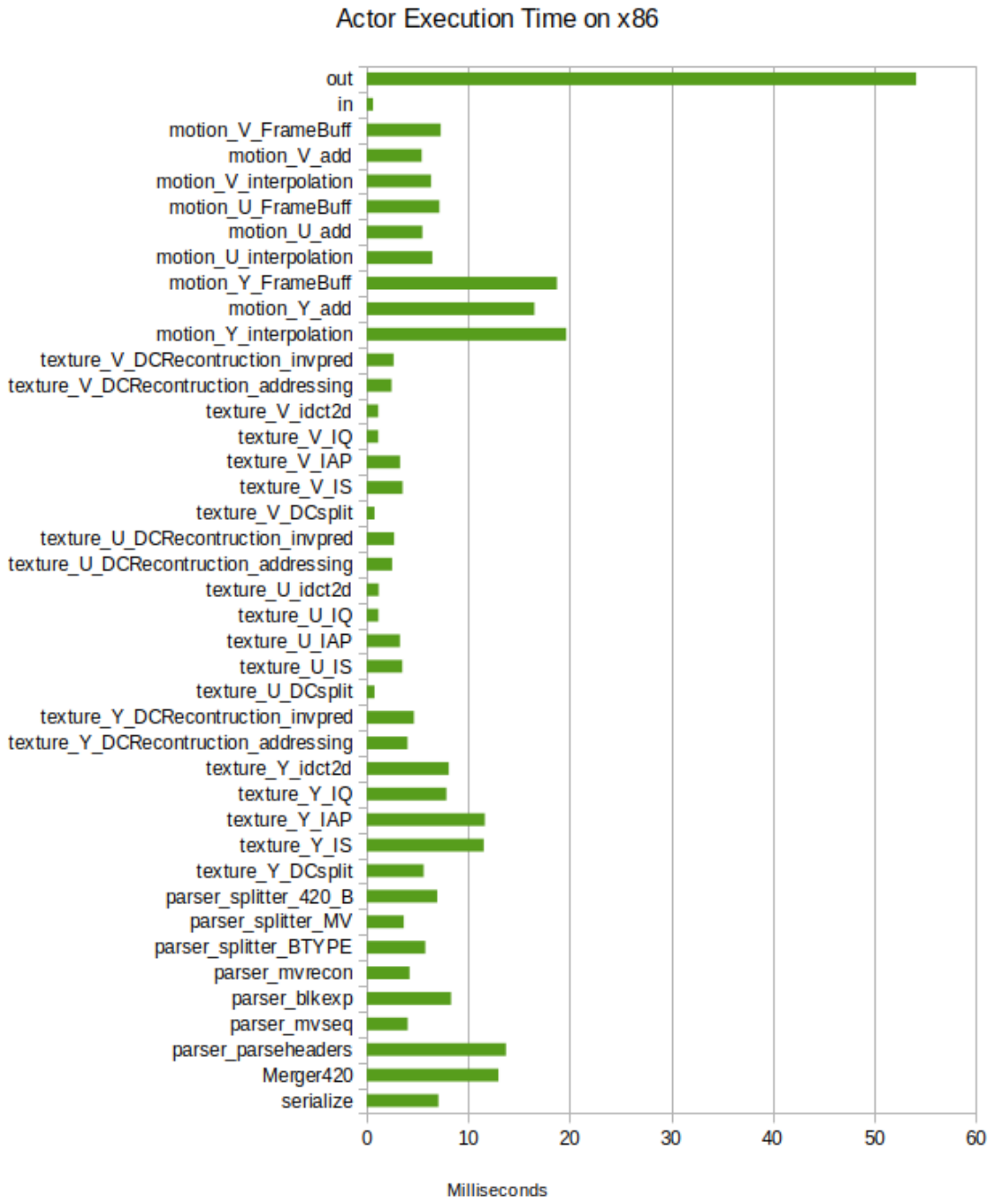

Figure 5.5: Actor execution times on a x86 platform.

Table 5.7: Table displaying the difference in execution time in seconds between the single-core and multi-core x86 program. As there is interference from the OS 5 executions were made and averaged.

Total execution time (s) of the x86 streaming application		
Execution ID	**Single-Core**	**Multi-Core**
1	0.332	1.660
2	0.300	1.856
3	0.324	1.688
4	0.328	1.792
5	0.304	1.760
Average:	0.3176	1.7512

5.4 Benefits and Drawbacks of Patmos

The Patmos architecture featured a few benefits over x86 when it came to the streaming application. The main advantage is the impressive scalability of 344% when comparing the best results for a single core and a 4 core solution, see the lowest bar in figure 5.3 (432 ms) and compare this with the average frame time of 1486 ms for the single core solution. This is in stark contrast to the scalability of x86 for this particular application, see table 5.7, which shows a degradation by a factor higher than 5. As stated in section 3.6 the multi-core x86 solution features one pthread per actor and the FIFO queues use atomic variables. This is not a good implementation on x86 as it does not account for various cache coherency problems [3] such as false sharing [32] and cache line alignment [20]. Swapping this over to the other version of having one pthread per core and letting the actors be updated in a round-robin fashion would be more fair, but as this was tried earlier on and found to perform even worse than having one pthread per actor, we decided not to as we only needed to show that the x86 platform suffered a performance degradation. Not having to worry about problems such as cache-coherence is another benefit of Patmos as far as actor based applications are concerned. The Patmos platform also benefited from the lack of an operating system, which would normally be interfering with the performance of the application. As a result of this all executions had a high degree of stability and rarely fluctuated between executions. This resulted in faster benchmarking as a single run was sufficient to evaluate the performance of a solution.

The drawbacks imposed by the Patmos architecture consisted of the lack of memory, the time-multiplexed access to the main memory and the lack of a receive buffer for the Ethernet. Patmos featuring 128 *KiB* of scratchpad memory and 2 *MiB* of external memory was completely incapable of running the original application, which at the time used 7 *MiB* of memory. As any access to the NoC was time multiplexed we had to create a balance between the communication division and the amount of cores, as all cores had to communicate on an even level in order to maintain performance. The memory limitations can not be blamed on Patmos as there is a hard limit on the available memory modules on the FPGA. Some of these drawbacks were a result of the unorthodox application chosen as it could not make any use of the WCET optimizations Patmos specializes in. The rest of the drawbacks, such as the lack of a receive buffer for the Ethernet may in the future be corrected since. Patmos is still a work in progress. Because of this some features were mutually exclusive, like the NoC and properly implemented Ethernet. Fortunately we could work around this as we received major help from the Patmos development team.

5.5 Ethical Aspects

Real-time optimized micro processors play a very important roll in today's society. They make sure that planes stay in the air and are responsible for many important parts of our infrastructure such as traffic lights and railways. Needless to say a failure of these processors could be devastating. In this thesis we have mentioned how safety-critical applications make use of platforms such as Patmos. However, when designing such an application for Patmos it is necessary to remember that the average case execution time, while being stable, is not guaranteed to always have the same execution time. It is therefore necessary to design a safety-critical application to make use of the worst case execution time instead. If a programmer implementing a safety-critical application was unaware of this they would end up making an application that was guaranteed to fail.

Chapter 6
Conclusions and Future Work

In this thesis we have investigated the benefits and drawbacks of implementing a streaming application on a Patmos processor. Now we can make use of the data collected and draw some conclusions. This is what we will be doing in this chapter. We start of by answering our research questions and then move on to future work. When we started this project we wanted to answer two research questions, but couldn't as we did not have the necessary data. However, now that we have access to the results presented in this report we can finally answer those questions. To answer our first research question, *What are the benefits and drawbacks to the Patmos platform viewed from the unconventional perspective of implementation of a streaming application?*, we can summarize the discussion presented in section 5.4. The most prominent benefit is the scalability provided by the Patmos platform. Achieving over three times the performance when switching from single-core to four core configurations. As can be seen in figure 5.3 the best performing four-core solution had a frame-time of 432 ms while the single core version had a frame-time of 1486 as presented in section 5.2. Comparing this to the x86 version which saw a performance degradation when swapping to a four core solution, going from a single-core execution time of 0.3176 *s* to a four-core execution time of 1.7512 *s*. Comparing the actual execution time of the Patmos implementation to the x86 version is not relevant as Patmos is running on an FPGA with severely limited resources while the x86 version is implemented on a computer with many times the resources. Furthermore, the x86 versions multi-core was implemented slightly different from the Patmos version as discussed in section 5.4. However, as the only relevant information to obtain here is that the x86 version had a degradation in performance when swapping to the multi-core implementation, this is not an issue. The second benefit to the Patmos platform was its stability. Since the platform is optimized to make the WCET analyzable it has also, as a side effect of these optimizations, made the average case execution time very stable. Combining this with having no interference from an operating system or background applications, all the Patmos executions would have almost the same frame time, varying only with a few ms. Because of this we could obtain a good average value in only a few iterations of a benchmark. The drawbacks imposed by the Patmos architecture were: the lack of memory, the time-multiplexed access

to the main memory and the lack of a receive buffer for the Ethernet. The lack of memory is always an issue in embedded systems, and can only be fixed by upgrading the hardware. The lack of a receive buffer for the Ethernet has also been fixed in the newer version of Patmos, but as of writing this report there were no stable versions that featured the buffer as well as a functioning NoC. The time-multiplexed access to the main memory, as well as the communication between cores, were small factors in this project. However, if the issue with the lack of memory was fixed, by for example upgrading the FPGA board to one with much more memory, this would become the main bottleneck.

When it comes to the mapping problem, the most obvious conclusions we can draw from the results is that the actor layout, or the mapping between actors and cores, have a significant performance impact, increasing it by over 50% at times, as can be seen in figure 5.1. To answer our second research questions, *Is it feasible to automatically map an actor system to the Patmos architecture without loosing performance compared to a manual mapping?*, it was possible to automatically map an actor system to the Patmos architecture. This can be seen from figure 5.1 as the manual layout actually performed worse than some of the automatic mappings, so we actually had a performance increase by letting the mapping be done automatically. Furthermore, the automatic mapping could easily compute solutions to even harder packing problems, such as with the increased queue sizes or with more cores, while making a manual mapping for these configurations can take several hours. The most successful algorithm we found for the layout was to load balance the individual actors execution time spread out amongst the cores. We also found that improving on the model by adding in heuristics such as constraining all members of strongly connected sub-graphs to be on the same core improved the performance further, as can be seen by comparing the values in figure 5.1. The automated solution even beat our best effort manual layout.

When it comes to future work, all optimizations ideas has not yet been tested. We theorize that perhaps a better performing solution could be created by performing live analysis of the queues to obtain the variances of their read and write rate. And then make use of this data with a new constraint program to dimension the queues to minimize stalls caused by queues being full or empty for extended duration's of time and letting queues with large variance in the write or read rate be larger. The advantage of longer queues is obvious when comparing figure 5.1 and figure 5.3.

Furthermore, there are still a lot of room for improvements and optimizations. We simplified our FIFO queues to only work with 32-bit tokens to save time and effort as the transfer granularity was 32-bit words. However as most FIFO queues only needs 16-bit integers, bytes or only bit tokens this could certainly be improved upon by implementing dedicated solutions for these queues. We theorize that this would translate to a significant performance increase as it would let the queue sizes be further increased, and again, as can be seen by comparing figure 5.1 to figure 5.3 larger queue sizes tend to increase performance. Another optimization that was discussed but never implemented was the deferred update of the writer shadow state when doing bulk transfers. In our current solution the actors can't do computations while the DMA is working and there is also a few inefficiencies when sending data from an actor attribute as the data is first copied to a staging buffer before the actual transfer can begin. This should not be necessary as the actors attributes are often already located in the SPM and can serve as the transfer source immediately.

All in all we can safely conclude that the Patmos platform shows significant potential alongside a helpful and dedicated development team. We have no doubts that Patmos will be

a great fit for any application that require scalability and predictability, once a stable version is released.

References

[1] Altera de2-115 development and education board. Taken 2020-05-18.

[2] Patmos a time-predictable processor for real-time systems. Taken 2020-01-22.

[3] James K Archibald. *The cache coherence problem in shared-memory multiprocessors*. PhD thesis, 1987.

[4] Russell Atkinson and Carl Hewitt. Synchronization in actor systems. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 267–280, 1977.

[5] Bhattacharyya, Shuvra S, Eker, Johan, Janneck, Jörn W, Lucarz, Christophe, Mattavelli, Marco, Raulet, and Mickaël. Overview of the mpeg reconfigurable video coding framework. *Journal of Signal Processing Systems*, 63(2):251–263, 2011.

[6] Gustav Björdal, Jean-Noël Monette, Pierre Flener, and Justin Pearson. A constraint-based local search backend for minizinc. *Constraints*, 20(3):325–345, 2015.

[7] Johan Eker and J Janneck. Cal language report: Specification of the cal actor language, 2003.

[8] Johan Eker and Jörn W Janneck. An introduction to the caltrop actor language, 2001.

[9] Terry J Fountain. Processor arrays: Architecture and applications. 1987.

[10] Essayas Gebrewahid, Mingkun Yang, Gustav Cedersjö, Zain Ul Abdin, Veronica Gaspes, Jörn W Janneck, and Bertil Svensson. Realizing efficient execution of dataflow actors on manycores. In *2014 12th IEEE International Conference on Embedded and Ubiquitous Computing*, pages 321–328. IEEE, 2014.

[11] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial intelligence*, 8(3):323–364, 1977.

[12] Intel. Quartus prime lite edition, 10 2019. Taken 2020-03-05.

[13] Wooyoung Kim. *ThAL: An actor system for efficient and scalable concurrent computing*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[14] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. *ACM SIGPLAN Notices*, 43(6):114–124, 2008.

[15] Ming-Huang Kuo. Method for processing an image using difference wavelet, January 13 2005. US Patent App. 10/604,265.

[16] Jerry Lindström and Stefan Nannesson. Implementing a streaming application on a processor array, June 2015. Taken 2020-01-22.

[17] Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. Approximating pareto optimal compiler optimization sequences—a trade-off between wcet, acet and code size. *Software: Practice and Experience*, 41(12):1437–1458, 2011.

[18] User Manual. Altera de2 board. *Altera Corporation*, 72, 2006.

[19] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[20] P Ranjan Panda, Hiroshi Nakamura, Nikil D Dutt, and Alexandru Nicolau. A data alignment technique for improving cache performance. In *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, pages 587–592. IEEE, 1997.

[21] Fernando CN Pereira, Fernando Manuel Bernardo Pereira, Fernando C Pereira, Fernando Pereira, and Touradj Ebrahimi. *The MPEG-4 book*. Prentice Hall Professional, 2002.

[22] Michal Podpora, Grzegorz Pawel Korbas, and Aleksandra Kawala-Janik. Yuv vs rgb-choosing a color space for human-machine interaction. *FedCSIS Position Papers*, 18:29–34, 2014.

[23] Charles Poynton. Chroma subsampling notation. *Retrieved June*, 19:2004, 2002.

[24] Peter Puschner and Alan Burns. A review of worst-case execution-time analyses. *REAL TIME SYSTEMS-AVENEI NJ-*, 18(2/3):115–128, 2000.

[25] James A Ross, David A Richie, Song J Park, and Dale R Shires. Parallel programming model for the epiphany many-core coprocessor using threaded mpi. *Microprocessors and Microsystems*, 43:95–103, 2016.

[26] Sergio Scaglia. *The Embedded Internet: TCP/IP Basics, Implementation and Applications*. Addison-Wesley Professional, 2007.

[27] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva,

Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

[28] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. *Technical University of Denmark, Tech. Rep*, 2014.

[29] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: A time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, 2018.

[30] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The patmos approach. In *Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, pages 11–21. OASICS, 2011.

[31] James E Smith. A study of branch prediction strategies. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 202–215, 1998.

[32] Josep Torrellas, HS Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[33] David Wang. Isscc 2005: The cell microprocessor. *Real World Technologies, Feb*, 2005.

[34] Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan. Software code generation for the rvc-cal language. *Journal of Signal Processing Systems*, 63(2):203–213, 2011.

[35] Marco Ziccardi. Modulo and division vs bitwise operations, 5 2015. Taken 2020-04-08.

Appendices

Appendix A
Installation

Patmos is a fairly large system which not unlike most systems have its own fair share of dependencies. Before there is any chance of getting Patmos running it is important to install all of them. If you are on an Ubuntu operating system you can follow these commands which have been taken from the Patmos reference handbook section 6 [28]. Firstly we recommend downloading Quartus from the Intel download page [12] and letting the download go in the background as it can take quite a while with a slower internet, see the Quartus section below for more detail. When it comes to actual commands we start of by enabling the sbt download distribution:

```
 echo "deb https://dl.bintray.com/sbt/debian /" | sudo
    tee -a \
/etc/apt/sources.list.d/sbt.list sudo apt-key adv --
   keyserver hkp://keyserver.ubuntu.com:80 \
--recv 2EE0EA64E40A89B84B2DF73499E82A75642AC823
 sudo apt-get update
```

After which we can install the necessary tools, minus quartus, with:

```
 sudo apt-get install git default-jdk gitk cmake make g++
     texinfo flex bison \
subversion libelf-dev graphviz libboost-dev libboost-
   program-options-dev \
ruby-full liblpsolve55-dev python zlib1g-dev gtkwave
   gtkterm scala sbt
```

A.1 Quartus

As Intel acquired Altera who owned Quartus, there is some confusion regarding which version of Quartus to acquire. You want to get the Quartus Prime Lite Edition which can be

found at the Intel download page[12] and not the web version. This is because of a name change that happened when Intel took over. After that all the other steps can be found in section 6.3 Quartus on Linux in the handbook [28].

A.2 Multi-Core

As of this report being written there are a lot of issues with the multi-core aspect on the Patmos master. These issues stem from an ongoing transfer from chisel 2 to chisel 3. To avoid these we need to checkout an older revision of the platform in order to get it working. To do so navigate to the Patmos directory (t-crest/patmos) and use the command:

```
git checkout baaa205c7d5c951613394105b57531f2a0c0d9ea
```

After that you can edit the configuration file for your fpga (for example the altde2-115 config file is t-crestpatmos/hardwareconfig/altde2-115.xml) to turn on multicore as shown in section 1.2.8 of the reference handbook [28]. There are actually even more useful configurations that be edited in the default.xml file in the same path, such as the DSPM size which is a measly 2kb by default.

Appendix B
Help Tools

B.1 Table Generated From Queue Analysis

sending actor	Packet Count In	Packet Count Out	receiver
parser_parseheaders	1	1	parser_mvrecon
parser_parseheaders	1	1	parser_blkexp
parser_parseheaders	1	1	parser_blkexp
parser_parseheaders	1	1	parser_blkexp
parser_mvseq	1	1	parser_mvrecon
parser_blkexp	1	64	parser_splitter_420_B
parser_mvrecon	1	2	parser_splitter_MV
parser_parseheaders	1	1	parser_mvrecon
parser_parseheaders	1	1	parser_mvseq
parser_parseheaders	1	1	parser_mvrecon
parser_parseheaders	1	1	parser_splitter_420_B
parser_parseheaders	1	1	parser_splitter_BTYPE
parser_parseheaders	1	1	parser_mvseq
parser_parseheaders	1	1	parser_splitter_MV
parser_parseheaders	1	1	parser_mvrecon
parser_parseheaders	1	1	parser_splitter_BTYPE
parser_parseheaders	1	1	parser_splitter_BTYPE
parser_parseheaders	1	1	parser_mvseq
parser_parseheaders	1	1	parser_mvrecon
parser_parseheaders	1	1	parser_splitter_BTYPE
parser_parseheaders	1	1	parser_mvseq
parser_parseheaders	1	1	parser_splitter_420_B
parser_parseheaders	1	1	parser_mvrecon
parser_parseheaders	1	1	parser_splitter_BTYPE

serialize	1	1	parser_parseheaders
texture_Y_DCsplit	63	1	texture_Y_IS
texture_Y_IS	1	1	texture_Y_IAP
texture_Y_IAP	1	63	texture_Y_IQ
texture_Y_IQ	1, 63	64	texture_Y_idct2d
texture_Y_DCRecontruction_addressing	1	1	texture_Y_DCRecontruction_invpred
texture_Y_DCRecontruction_addressing	1	1	texture_Y_DCRecontruction_invpred
texture_Y_DCRecontruction_addressing	1	1	texture_Y_DCRecontruction_invpred
texture_Y_DCRecontruction_invpred	1	1	texture_Y_IQ
texture_Y_DCRecontruction_invpred	1	1	texture_Y_IQ
texture_Y_DCRecontruction_invpred	1	1	texture_Y_IAP
texture_Y_DCRecontruction_invpred	1	1	texture_Y_IAP
texture_Y_DCRecontruction_invpred	1	1	texture_Y_IS
texture_Y_DCRecontruction_invpred	1	1	texture_Y_idct2d
texture_Y_DCRecontruction_invpred	1	1	texture_Y_IAP
texture_Y_DCRecontruction_invpred	1	1	texture_Y_IAP
texture_Y_DCsplit	1	1	texture_Y_DCRecontruction_invpred
parser_splitter_420_B	64	64	texture_Y_DCsplit
parser_splitter_BTYPE	1	1	texture_Y_DCRecontruction_invpred
parser_parseheaders	1	1	texture_Y_DCRecontruction_addressing
parser_parseheaders	1	1	texture_Y_DCRecontruction_invpred
parser_splitter_BTYPE	1	1	texture_Y_DCRecontruction_invpred
parser_splitter_BTYPE	1	1	texture_Y_DCRecontruction_invpred
parser_splitter_BTYPE	1	1	texture_Y_DCRecontruction_addressing
parser_splitter_BTYPE	1	1	texture_Y_DCRecontruction_invpred
texture_U_DCsplit	63	1	texture_U_IS
texture_U_IS	1	1	texture_U_IAP
texture_U_IAP	1	63	texture_U_IQ
texture_U_IQ	1, 63	64	texture_U_idct2d
texture_U_DCRecontruction_addressing	1	1	texture_U_DCRecontruction_invpred
texture_U_DCRecontruction_addressing	1	1	texture_U_DCRecontruction_invpred
texture_U_DCRecontruction_addressing	1	1	texture_U_DCRecontruction_invpred
texture_U_DCRecontruction_invpred	1	1	texture_U_idct2d
texture_U_DCRecontruction_invpred	1	1	texture_U_IQ
texture_U_DCRecontruction_invpred	1	1	texture_U_IQ
texture_U_DCRecontruction_invpred	1	1	texture_U_IAP
texture_U_DCRecontruction_invpred	1	1	texture_U_IAP
texture_U_DCRecontruction_invpred	1	1	texture_U_IS
texture_U_DCRecontruction_invpred	1	1	texture_U_IAP
texture_U_DCRecontruction_invpred	1	1	texture_U_IAP
texture_U_DCsplit	1	1	texture_U_DCRecontruction_invpred
parser_splitter_420_B	64	64	texture_U_DCsplit
parser_splitter_BTYPE	1	1	texture_U_DCRecontruction_invpred
parser_parseheaders	1	1	texture_U_DCRecontruction_addressing
parser_parseheaders	1	1	texture_U_DCRecontruction_invpred
parser_splitter_BTYPE	1	1	texture_U_DCRecontruction_invpred
parser_splitter_BTYPE	1	1	texture_U_DCRecontruction_invpred
parser_splitter_BTYPE	1	1	texture_U_DCRecontruction_addressing
parser_splitter_BTYPE	1	1	texture_U_DCRecontruction_invpred

texture_V_DCsplit	63	1	texture_V_IS
texture_V_IS	1	1	texture_V_IAP
texture_V_IAP	1	63	texture_V_IQ
texture_V_IQ	1, 63	64	texture_V_idct2d
texture_V_DCRecontruction_addressing	1	1	texture_V_DCRecontruction_invpred
texture_V_DCRecontruction_addressing	1	1	texture_V_DCRecontruction_invpred
texture_V_DCRecontruction_addressing	1	1	texture_V_DCRecontruction_invpred
texture_V_DCRecontruction_invpred	1	1	texture_V_idct2d
texture_V_DCRecontruction_invpred	1	1	texture_V_IQ
texture_V_DCRecontruction_invpred	1	1	texture_V_IQ
texture_V_DCRecontruction_invpred	1	1	texture_V_IAP
texture_V_DCRecontruction_invpred	1	1	texture_V_IAP
texture_V_DCRecontruction_invpred	1	1	texture_V_IS
texture_V_DCRecontruction_invpred	1	1	texture_V_IAP
texture_V_DCRecontruction_invpred	1	1	texture_V_IAP
texture_V_DCsplit	1	1	texture_V_DCRecontruction_invpred
parser_splitter_420_B	64	64	texture_V_DCsplit
parser_splitter_BTYPE	1	1	texture_V_DCRecontruction_invpred
parser_parseheaders	1	1	texture_V_DCRecontruction_addressing
parser_parseheaders	1	1	texture_V_DCRecontruction_invpred
parser_splitter_BTYPE	1	1	texture_V_DCRecontruction_invpred
parser_splitter_BTYPE	1	1	texture_V_DCRecontruction_invpred
parser_splitter_BTYPE	1	1	texture_V_DCRecontruction_addressing
parser_splitter_BTYPE	1	1	texture_V_DCRecontruction_invpred
motion_Y_interpolation	1	64	motion_Y_add
motion_Y_FrameBuff	81	1	motion_Y_interpolation
motion_Y_FrameBuff	1	1	motion_Y_interpolation
motion_Y_add	64	64	motion_Y_FrameBuff
motion_Y_add	64	256	Merger420
parser_splitter_MV	2	1	motion_Y_FrameBuff
parser_parseheaders	1	1	motion_Y_FrameBuff
parser_parseheaders	1	1	motion_Y_FrameBuff
parser_parseheaders	1	1	motion_Y_FrameBuff
parser_splitter_BTYPE	1	1	motion_Y_add
parser_splitter_BTYPE	1	1	motion_Y_FrameBuff
parser_splitter_BTYPE	1	1	motion_Y_add
parser_splitter_BTYPE	1	1	motion_Y_FrameBuff
texture_Y_idct2d	64	64	motion_Y_add
motion_U_interpolation	1	64	motion_U_add
motion_U_FrameBuff	81	1	motion_U_interpolation
motion_U_FrameBuff	1	1	motion_U_interpolation
motion_U_add	64	64	motion_U_FrameBuff
motion_U_add	64	64	Merger420
parser_splitter_MV	2	1	motion_U_FrameBuff
parser_parseheaders	1	1	motion_U_FrameBuff
parser_parseheaders	1	1	motion_U_FrameBuff
parser_parseheaders	1	1	motion_U_FrameBuff
parser_splitter_BTYPE	1	1	motion_U_add
parser_splitter_BTYPE	1	1	motion_U_FrameBuff

parser_splitter_BTYPE	1	1	motion_U_add
parser_splitter_BTYPE	1	1	motion_U_FrameBuff
texture_U_idct2d	64	64	motion_U_add
motion_V_interpolation	1	64	motion_V_add
motion_V_FrameBuff	81	1	motion_V_interpolation
motion_V_FrameBuff	1	1	motion_V_interpolation
motion_V_add	64	64	motion_V_FrameBuff
motion_V_add	64	64	Merger420
parser_splitter_MV	2	1	motion_V_FrameBuff
parser_parseheaders	1	1	motion_V_FrameBuff
parser_parseheaders	1	1	motion_V_FrameBuff
parser_parseheaders	1	1	motion_V_FrameBuff
parser_splitter_BTYPE	1	1	motion_V_add
parser_splitter_BTYPE	1	1	motion_V_FrameBuff
parser_splitter_BTYPE	1	1	motion_V_add
parser_splitter_BTYPE	1	1	motion_V_FrameBuff
texture_V_idct2d	64	64	motion_V_add
Merger420	256, 64	set()	out
parser_parseheaders	1	set()	out
parser_parseheaders	1	set()	out
in	set()	1	serialize

Appendix C
Project Code

The project will be provided on request to anyone interested.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-06-15

EXAMENSARBETE Evaluating a Real-time Multi-core Processor for Embedded Streaming
STUDENTER Linus Gudmundsson, Jacob Canbäck
HANDLEDARE Jörn W. Janneck (LTH)
EXAMINATOR Flavius Gruian (LTH)

Exploring a time-driven platform with the help of a data-driven application

POPULÄRVETENSKAPLIG SAMMANFATTNING **Linus Gudmundsson, Jacob Canbäck**

Processors designed to be time-driven and allow for worst case execution time analysis are a common thing in safety-critical systems. But what happens when a data-driven application, in this project a streaming application, is implemented on such a platform? In this project we will answer this question.

Today's modern processors are able to run just about any application with significant performance. But how is this possible? With so many different types of applications in existence, how can a single processor be compatible with all of them while also maintaining high performance? It is because the processor is designed and optimized for such general purpose use.

However, such optimizations are often not without the drawback of increasing the *Worst Case Execution Time* (WCET). Most of the time the impact from this is not noticeable, but once in a blue moon something goes wrong and even simple applications can take very long to complete. Furthermore, determining the WCET for an application on such general purpose processors can be a nightmare when it comes to complexity. This is where Patmos, a processor array designed for real-time systems to make it as easy as possible to calculate the WCET comes in. There is no question that such processors are necessary when it comes to safety-critical applications that simply are not allowed to fail, such as train controllers. But how does such a platform handle non time-driven or safety-critical applications?

In this project we tested this by implementing a data-driven streaming application. This streaming application is made up of an actor system, which creates a need to map these actors to the processor cores in some fashion. We further explored how much this mapping affected the application performance and if it was possible to automatically create the mapping without significantly dropping the performance.

This application was translated to a Patmos application by making use of a pipeline of Python programs we created. The biggest hindrance to the translation was the memory limits on the FPGA which Patmos ran on. As the application originally required far more memory than was available, we had to extensively reduce the memory usage. The mapping from actors to cores was done by modelling the problem in MiniZinc and letting the solvers determine a layout.

The results show that the application was able to run on Patmos without any major issues. When it came to scalability Patmos even beat the x86 platform as the parallel version of the application performed far better than the single-core version, while the opposite was seen on the x86 platform. Furthermore the results clearly displayed that there is a significant performance gain from optimizing the actor to core mapping, as the performance almost doubled between some layouts.