A Framework for Parallel Streaming Applications

Maik Nijhuis
This work was carried out in the ASCI graduate school.
ASCI dissertation series number 214.

This thesis was printed by the printing office of the Vrije Universiteit.

The raw material for the paper of this thesis comes from renewable sources, such as responsibly managed forests.

Copyright ©2010 by M.H.J. Nijhuis
A Framework for Parallel Streaming Applications
# Contents

## Acknowledgments

8

## 1 Introduction

9

1.1 Streaming applications

11

1.2 Architectures

12

1.3 Roles

13

1.4 Contributions

14

1.5 The SP@CE Framework

15

1.5.1 SP@CE properties

15

1.5.2 SP@CE structure

16

## 2 Related work

17

2.1 Streaming Frameworks

17

2.2 Component-based frameworks

20

2.3 Parallel programming

21

2.3.1 Skeletons and design patterns

21

2.3.2 Coordination languages

22

2.4 Heterogeneous frameworks

23

2.4.1 Multimedia MPSoC architectures

23

2.4.2 Network processors

23

2.4.3 CPU + GPU combinations

24

## 3 The Hinch run time system

25

3.1 Requirements

26

3.1.1 Components

27

3.1.2 Iterations

29

3.1.3 Parallelism

29

3.1.4 Communication

29

3.1.5 Dynamic reconfigurability

30

3.1.6 Ease-of-use

32

3.2 Design

33

3.2.1 Components

33

3.2.2 Parallelism

34
CONTENTS

3.2.3 Communication ................................................. 37
3.2.4 Reconfiguration and event handling ......................... 39
3.2.5 Iterations .................................................. 41
3.2.6 Ease-of-use ................................................ 41

3.3 Implementation .................................................. 42
3.3.1 Component tree ............................................. 43
3.3.2 Data flow graph ............................................. 46
3.3.3 Parallelism .................................................. 48
3.3.4 Communication streams ..................................... 51
3.3.5 Event communication ....................................... 52
3.3.6 Dynamic reconfigurability .................................. 53

3.4 Experiments ..................................................... 55
3.4.1 Applications ................................................. 55
3.4.2 Hinch overhead .............................................. 58
3.4.3 Parallel performance ........................................ 60
3.4.4 Reconfigurability overhead .................................. 62

3.5 Summary ........................................................ 64

4 The XSPCL coordination language .................................. 67
4.1 Required properties of XSPCL ................................... 68
4.2 XSPCL language specification ................................... 70
4.2.1 Header ...................................................... 70
4.2.2 Components ................................................ 72
4.2.3 Procedures ................................................ 75
4.2.4 Parallelism .................................................. 77
4.2.5 Alternating components ..................................... 82
4.2.6 Feedback loop .............................................. 84
4.2.7 Reconfigurability ........................................... 85

4.3 Implementation .................................................. 88
4.3.1 Reconfiguration ............................................. 88

4.4 Experiments ..................................................... 89
4.4.1 XSPCL Overhead ............................................ 90
4.5 Summary ........................................................ 93

5 Gordon: A Cell run time library ...................................... 95
5.1 The Cell Processor ............................................... 96
5.1.1 Problems ................................................... 97
5.2 Related programming environments .............................. 100
5.3 Gordon .......................................................... 102
5.3.1 Application Programming Interface ......................... 102
5.3.2 Buffer management ......................................... 103
5.3.3 SPE function library ........................................ 105
5.3.4 Job chaining ................................................ 107
5.3.5 Notification approaches .................................... 109
CONTENTS

5.3.6 Implementation ........................................... 111
5.4 SP@CE applications on the Cell processor .................... 114
  5.4.1 Gordon components ..................................... 114
5.5 Experiments .................................................. 118
  5.5.1 SPE functions ........................................... 118
  5.5.2 Notification approaches ................................ 121
  5.5.3 DMA overhead and multi-buffering .......................... 122
  5.5.4 Job chaining ............................................. 124
  5.5.5 Persistent data ......................................... 125
5.6 Summary ...................................................... 126

6 Applications and Evaluation .................................. 127
  6.1 Application descriptions ................................... 127
    6.1.1 Picture-in-Picture .................................... 129
    6.1.2 Filter ................................................ 130
    6.1.3 Rotator .............................................. 131
    6.1.4 Correlator ........................................... 132
    6.1.5 Edge Detector ....................................... 133
    6.1.6 JPEG Matrix Display .................................. 135
  6.2 Experiments ................................................ 140
    6.2.1 Parallel performance .................................. 141
    6.2.2 Heterogeneity ........................................ 143
  6.3 Summary .................................................... 145

7 Conclusion ..................................................... 147
  7.1 Summary ..................................................... 147
    7.1.1 Research questions .................................... 147
    7.1.2 Contributions ......................................... 150
    7.1.3 Applications ......................................... 152
  7.2 Future work ................................................ 152
    7.2.1 Distributed memory / Grid computing ..................... 153
    7.2.2 Load balancing heterogeneous resources .................. 153
    7.2.3 Real time systems ..................................... 154
    7.2.4 Automatic component combining .......................... 154
    7.2.5 Challenging applications ................................ 155
    7.2.6 Graphical front-end ................................... 155

Index ............................................................. 156

Bibliography ...................................................... 157

Nederlandse samenvatting (Dutch summary) ......................... 169
Acknowledgments

This thesis would not have been written without the help of many people. My first and biggest thanks go to my supervisors Henri and Herbert, for their continuous support. Besides my supervisors, the thesis committee consisted of Marco Danelluto, Henk Corporaal, Henk Sips, Willem de Bruijn and Rob van Nieuwpoort. I thank them for taking the time for reading this thesis and for giving their valuable comments.

The research for this thesis was performed in collaboration with a number of people. Thanks to Henk Sips for writing the SCALP project proposal, getting it accepted, and managing the project. Thanks to Ana Varbanescu, for the collaboration within the project. Thanks to Arturo González-Escribano for the lively discussions about XSPCL. Thanks to Cédric Augonnet for bootstrapping my work on the Cell processor.

During the first part of my PhD period I worked with Philips Research in Eindhoven. I want to thank everybody from this group for their cooperation. I especially want to thank Paul Stravers for his guidance in this first period.

The experiments on the Cell blade and Niagara platforms were performed on systems at the College of Computing at Georgia Institute of Technology. I would like to thank David Bader for giving me access to these systems. Without these systems, most experiments would not have been possible.

At the Vrije Universiteit I had many roommates during my PhD period, although I never moved myself. Patrick, Srijith, Michel, Reza, Reinout, Ela, Mauro, and Ana: Thank you for keeping me company during these years. An extra thank you goes to Michel for using his \LaTeX{} bible, and another one to Srijith, for being his paranymph. Besides my roommates, I also want to thank my other colleagues at the Vrije Universiteit.

Even though they haven’t performed their duties as of this writing, I want to thank Arjen and Roelof for being my paranymphs. Furthermore, I want to thank the many people outside academia who supported me.

Finally, thanks go to ALDI supermarket for supplying methyltheobromine, 2-aminoethanesulfonic acid, and cyanocobalamin on an as-needed basis.
Chapter 1

Introduction

Streaming applications process continuous streams of data. For example, multimedia applications continuously read data from a file or a network connection, decode it, and send it to one or more output devices, such as speakers for audio data and screens for video data. Network packet processing is also an important streaming application. Network packets continuously enter the application, which inspects the packets and performs operations on them, such as filtering, encapsulation, decapsulation, splitting and merging. Finally, the application sends zero or more packets to one or more output channels. Streaming also occurs in scientific applications, for example processing the data from radio telescopes that continuously scan for signals.

Developing streaming applications is challenging for multiple reasons. The bandwidth and compute power requirements are substantial for these applications. Encoding a 720x480 MPEG-2 video stream at 30 frames per second requires 1.5 MB/s of incoming bandwidth and multiple billions of operations per second \[7\], \[77\]. High-definition streams with resolutions up to 1920x1080, higher frame rates, and more advanced compression techniques, require an order of magnitude more bandwidth and compute power. Network packet processing applications nowadays handle multiple data streams up to 10 Gb/s, which requires similar compute power. Scientific applications may even require supercomputers for data processing. For example, the LOFAR distributed radio telescope requires 192 10 Gb/s connections and tens of TFlops for processing the data streams from various locations with radio antennas \[105\].

The complexity of the operations performed in streaming applications still increases. Analog television broadcast systems are being replaced by digital ones, which use advanced compression techniques. Internet web sites increasingly offer video streams besides text and images. High definition streams are becoming increasingly popular. Even mobile devices support streaming. For example, when making a phone call, a mobile phone processes incoming and outgoing audio streams. In the future, video telephony will also require these devices to process complex audio/video streams.
Because of the compute power requirements, using a single processing unit does not suffice for streaming applications. The application has to exploit parallelism. Fortunately, streaming applications contain many forms of parallelism. However, exploiting parallelism is difficult, as the application has to be split into different parts that communicate and synchronize regularly. Support for parallelism is a key component in any system supporting streaming applications.

Streaming applications typically combine several operations within a single application, which means the application has to transfer data between different subsystems. When the subsystems have incompatible interfaces, connecting them can be difficult, as the application has to convert the data. To address this problem, a streaming application can best have a common interface for all subsystems and provide wrappers that convert the native subsystem interface to the common interface. This approach has the advantage that subsystems can easily be reused across different applications that use the same internal interface. Moreover, a generic interface allows reusing subsystems within a single application. However, designing and implementing a generic internal interface is difficult, since it should support any kind of subsystem.

Since the challenges in building streaming applications are similar across different streaming applications, one should ideally create a generic system that addresses these challenges for all streaming applications. However, building such a system is more challenging than building streaming applications, since the system should ideally support any streaming application on any architecture. The system should support multiple forms of parallelism, allow application-specific optimizations, and provide a means for constructing applications out of different components, amongst others. The system interface should be as simple as possible, to aid the developer. The system should also have low overhead, to make using it worthwhile. In this thesis we discuss the design of such a system and describe how it addresses the various challenges posed by streaming applications. We will present an implementation of the resulting design and a performance evaluation on three different parallel architectures.

In this introduction we will first describe streaming applications in more depth in Section 1.1. Section 1.2 describes the architectures at which we run these applications. Section 1.3 discusses the user classes involved in developing and using streaming applications, which we will use throughout this thesis. Section 1.4 lists the research questions we answer in this thesis, which form the contributions of this thesis. Finally, Section 1.5 gives an overview of our framework for developing streaming applications. It describes the relation between the systems we have developed and provides an outline of the remainder of this thesis.
1.1 Streaming applications

In a streaming application new data continuously enters the application, for example from an Internet video stream. The application processes this data, e.g., it decompresses the video stream. Finally, it sends the result to some output device like a screen.

After outputting the data, the application discards it. Sometimes, an application contains a feedback loop in which it reuses output data. For example, many video compression techniques use previously decompressed images for decompressing the current image\[52, 127\]. This situation merely delays the discarding of data; the application discards the data as soon as it has processed the limited set of following images.

Processing a stream typically consists of multiple stages, which are handled by different independent kernels or components. In this thesis, we use the term component for a stream processing stage. In the Internet video stream example, one component manages the Internet connection, one component performs decompression, and one component manages the output device. Components can be composed of other components. For example, JPEG decompression consists of bit stream decoding and Inverse Discrete Cosine Transformations (IDCTs)\[124\]. Separating an application into components has various advantages:

- When the requirements for components are clearly specified, components can be independently developed and tested.
- When components adhere to a standard interface, components can be reused in different applications, which reduces development time.
- The application can automatically exploit parallelism by running independent components concurrently.

A stream is an unidirectional FIFO channel between one or more producing components and one or more consuming components. A component does not have to know to which other components it is connected, since it only interacts with the streams it is connected to. This abstraction further enables component reuse.

A streaming application typically uses both external and internal streams. External streams are connected to resources external to the application, for example, Internet connections can be used for external input and output streams when the application runs on a single host. Internal streams are used for connections between components within the application. They are typically implemented using efficient techniques, such as shared memory. However, when the application runs on multiple distributed hosts, internal streams must use less efficient techniques, such as network connections, for communication between components at different hosts. Ideally, all internal streams have the same interface towards components and hide the implementation details from components.

Input and output components perform the conversion between external and internal streams. Input components read data from an external stream and write data
to one or more internal streams. Output components perform the opposite operation: they read data from internal streams and produce data for external output devices. Other components read and write internal streams only, and perform some operation on the data in these streams. A component may be connected to multiple internal streams. For example, a component that performs YUV to RGB color conversion has one stream for each color component which yields three input streams and three output streams.

Streaming applications often have to respond to events, such as user input or sudden changes in the availability of resources. For example, a user may instruct an application to view a different video stream from the Internet. An application could also detect that the available network bandwidth suddenly drops, and switch to a low bandwidth stream with reduced quality. In both cases, the application has to reconfigure itself while it keeps running.

When an application requires reconfiguration, the actions in the application vary from simple to complex. When the application switches to a different video stream from the Internet that has the same properties as the original stream, the input component that reads the Internet stream only disconnects the old stream and connects the new stream. All other components and the internal streams require no modifications in this case. When the new video stream has different properties, for example when it uses a different compression scheme, the application has to replace the components that perform decompression. This kind of reconfiguration is complex, since the application structure changes considerably. The application also has to delete streams between the old components and create new streams between the new components. It also has to rewire existing streams towards the new components.

\section{Architectures}

Besides streaming applications themselves, the architectures on which they run are also complex. Current CPUs have multiple cores, which requires parallel programming. Moreover, heterogeneous architectures are becoming increasingly common, for example combining a general-purpose CPU with a special-purpose Graphics Processing Unit (GPU). For optimal performance, streaming applications have to be written as a parallel application and handle synchronization, communication, and load balancing, which increases their complexity.

In this thesis, we use three different platforms for evaluating streaming applications, namely a node of the DAS-3 supercomputer, a Cell blade server, and a SPARC Enterprise T5120 server. Besides these architectures, our streaming applications also run on a simulator for the SpaceCAKE architecture for embedded systems, developed by Philips\cite{114}. As our systems are mainly written in platform-independent C and XML code, porting our applications to other general-purpose and/or embedded architectures requires little effort. We will now explain the three evaluation platforms in detail. Table\cite{111} summarizes their main differences.
1.3. ROLES

<table>
<thead>
<tr>
<th>Name</th>
<th>Architecture</th>
<th>Heterogeneous</th>
<th>Cores</th>
<th>Threads per core</th>
<th>Total threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAS-3</td>
<td>x86-64</td>
<td>no</td>
<td>2x2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Cell blade</td>
<td>PowerPC+SPE</td>
<td>yes</td>
<td>2+16</td>
<td>2+8</td>
<td>20</td>
</tr>
<tr>
<td>Niagara</td>
<td>SPARCv9</td>
<td>no</td>
<td>8</td>
<td>8</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 1.1: Architectures used for evaluating streaming applications

- DAS-3. DAS-3 is a compute cluster that is distributed over several universities in the Netherlands\(^1\). We use DAS-3 nodes at the Vrije Universiteit in Amsterdam, which contain two dual-core 2.4 GHz AMD Opteron 280 processors per node. This processor uses the 64-bit x86_64 instruction set. The architecture thus supports four homogeneous threads per node. Our applications always run on a single node, since the current implementation does not support distributed memory.

- Cell blade. A Cell blade server contains two 3.2 GHz Cell processors\(^2\). Each Cell Processor contains a Power Processing Element (PPE) core, which has the 64-bit PowerPC instruction set and supports two hardware threads. The architecture supports shared memory between these four threads. Besides in a Cell blade, which is a general-purpose environment, the Cell processor is also used in embedded environments, such as the PlayStation 3 gaming console.

  Cell is a heterogeneous architecture. Besides the PPE, a Cell processor contains eight Synergistic Processing Element (SPE) cores\(^6\). The SPEs have a distributed-memory programming model. Each SPE has 256 kilobytes of local memory and a DMA engine for transferring data between main memory or other SPEs and its local memory. Section \(5.1\) describes the Cell architecture and the challenges it poses in detail.

- SPARC Enterprise T5120. This server contains an 1.167 GHz UltraSPARC T2 ‘Niagara’ processor, which uses the 64-bit SPARCv9 instruction set. The processor has eight homogeneous cores which each support eight hardware threads. The architecture supports shared memory between these 64 threads. We will use the term ‘Niagara’ for referencing this platform.

1.3 Roles

In the development of streaming applications, we distinguish different user classes. We provide a clear distinction between these user classes by using the following descriptions throughout this thesis:

- An end-user uses the application and needs no technical knowledge of the application or the hardware platform. These users interact with the application using a simple interface, such as a remote control.
A component developer develops components for streaming applications. A component uses the interface of the runtime environment for common functions, such as accessing a stream it is connected to. All components adhere to a generic interface, which the runtime environment uses for initializing, running and reconfiguring the component, amongst others. Section 3.2 describes our design of these interfaces.

A component developer writes components in an implementation language, such as C. Knowledge about the hardware platform is only required if the component developer wants to use hardware-specific optimizations, for example, Single Instruction, Multiple Data (SIMD) primitives. When the component uses external libraries, the component developer is responsible for interfacing with these libraries.

An application developer develops streaming applications using a high-level coordination language, which has primitives for building an application out of components. The coordination language can be different than the implementation language for the components. Application developers are fully aware of the high-level application structure, including the interface of the components that are used in the application. These users do not require knowledge about the implementation of components or about the hardware platform.

1.4 Contributions

The contributions of this thesis fall into two categories. The main contribution is a requirement analysis and design of a framework for building parallel streaming applications. The second contribution is the design of a simple interface for using heterogeneous resources. We verify our designs using prototype implementations. We will measure the overhead of our systems compared to using hand-written code. We will also evaluate the efficiency of the prototype implementation when using multiple cores. When combined, both contributions provide a valuable framework for building streaming applications for heterogeneous architectures. The main research questions we will answer using these contributions are:

1. How can one abstract the various challenges of building parallel streaming applications behind a simple interface, with low overhead?
2. How does one design and implement a high-level coordination language for parallel streaming applications?
3. How can one abstract the various difficulties of using heterogeneous resources with distributed memory semantics behind a simple interface, with low overhead?

Designing simple interfaces and not limiting functionality seem conflicting goals. Our systems therefore use an incremental approach. An application developer can
use only basic functionality for rapidly building prototype applications. Then the application developer can stepwise increase the prototype, for example by adding support for reconfigurability or by improving performance.

We do impose some restrictions on developers. For example, a component developer has to adhere to a fixed component interface. However, this interface is kept as simple as possible. Moreover, the component developer does not have to implement the full interface as many items are optional. Again, the component developer can first build a simple prototype component, and later extend this component with advanced features.

The next section will describe the top-level design of our framework, which consists of five subsystems. Three of these subsystems address the research questions posed above. Chapters 3, 4, and 5 describe these three systems in detail.

1.5 The SP@CE Framework

The solutions we provide for efficiently developing parallel streaming applications adhere to the SP@CE framework, which we developed in collaboration with Delft University of Technology and Universidad de Valladolid[121]. SP@CE consists of several interconnected systems, which address different problems related to developing these applications. We have focused on the run time system for these applications. The researchers from Delft University of Technology focused on performance prediction. We based the XML layer that couples the various systems in SP@CE on work by González-Escribano at Universidad de Valladolid[50].

We will use a top-down approach when describing SP@CE. First we will describe the global properties of SP@CE, which are common to all subsystems. Then we will describe the global structure of SP@CE. While describing the subsystems, we will also give an outline of the remainder of this thesis.

1.5.1 SP@CE properties

Following the description of streaming applications in Section 1.1, a SP@CE application consists of components that communicate using streams or by sending asynchronous events. SP@CE organizes the components into a data flow graph. The application runs as a series of iterations of the data flow graph, in which each component runs once. In an iteration, a component reads data from its input streams, performs some computation and writes the result to its output streams.

SP@CE provides various grouping constructs for building the data flow graph, including various task- and data-parallel constructs. New constructs can easily be added. An application developer can build dynamically reconfigurable applications by declaring parts of the data flow graph as optional. SP@CE enables or disables these parts in response to external or internal events. SP@CE runs the application in parallel by exploiting both task- and data-parallelism in the data flow graph and by running multiple iterations of the data flow graph concurrently.
CHAPTER 1. INTRODUCTION

1.5.2 SP@CE structure

SP@CE consists of five parts, as shown in Figure 1.1. An application developer specifies the data flow graph in a high-level coordination language using the Front-End, which compiles this coordination language into an intermediate XML-based language called XSPCL. The Front-End currently only exists in the design of the SP@CE framework. The implementation of the Front-End is subject to future work.

The XSPCL layer performs various transformations and optimizations on the intermediate representation. This layer includes a compiler which transforms an XSPCL representation into an executable object that is linked against the Hinch run time system (RTS). A performance prediction layer, called PaM-SoC, can also read XSPCL representations and provide feedback to the user and to SP@CE about optimization decisions\[122\]. Performance prediction is outside the scope of this thesis, however. Chapter 4 describes the XSPCL language and its implementation in detail.

The Hinch run time system performs all generic run time tasks for SP@CE applications, such as scheduling the data flow graph and stream management. This way, Hinch abstracts the component developer from low level communication and synchronization details. Hinch includes a component library with the available SP@CE components. Chapter 3 describes the requirements, the design, and the implementation of Hinch in detail.

SP@CE supports the heterogeneous Cell architecture using the Gordon run time library, which provides a generic interface for using the SPE coprocessors in this architecture. A SP@CE component uses this interface for offloading computations to the SPEs, which provide the bulk of the processing power in the Cell. Programming these SPEs is difficult, since the architecture has a distributed memory model. Chapter 5 describes the Cell architecture, the Gordon run time library, and the usage of Gordon within a SP@CE component in detail.

We show that SP@CE works for advanced applications in Chapter 6, which presents multiple advanced applications that were built in XSPCL, and that use Hinch and Gordon. We will show the efficiency of the applications on all platforms mentioned in Section 1.2 and we evaluate design choices of heterogeneous Cell applications.

Chapter 7 summarizes the Hinch run time system, the XSPCL coordination language, the Gordon run time library, and the results we obtained using these systems. It will also give directions for future work regarding the SP@CE framework and its subsystems.
Chapter 2

Related work

Many systems address the difficulties of programming complex applications. In this chapter, we give an overview of these systems. Since this field is very broad, we consider only systems that resemble our systems. In Section 2.1 we will discuss related frameworks for streaming applications. Section 2.2 discusses component-based systems. In Section 2.3 we will describe the relation to other parallel programming systems, focusing on design patterns, algorithmic skeletons, and coordination languages. Section 2.4 describes other heterogeneous platforms besides the Cell architecture and the frameworks for streaming applications on these platforms. We will describe the Cell architecture and mention related work regarding this processor in Sections 5.1 and 5.2, respectively, since Chapter 5 focuses on this architecture.

2.1 Streaming Frameworks

Several frameworks for developing parallel streaming applications have emerged. Table 2.1 lists these systems and compares them against our combined systems. We consider three types of parallelism, which are named differently in different systems. In Table 2.1, 'pipe' parallelism means exploiting pipeline parallelism by running multiple iterations of the application concurrently. The active components in each iteration are typically different. However, some systems exploit parallelism by running multiple iterations of a single component concurrently. Task parallelism occurs when multiple different components run concurrently within a single iteration. Data parallelism occurs when multiple similar components process the data in one iteration in parallel. The events column indicates if the system supports communication using asynchronous events, besides synchronous streaming communication.

Table 2.1 only compares the features of the various other systems to our systems. Because each system has a different set of accompanying applications, we can not directly compare these systems in terms of performance. A direct comparison would require porting our applications to the other systems and vice versa. Besides the huge development and performance measurement effort, we believe this approach
would still yield an unfair comparison, as the various systems were designed for
different kinds of streaming applications. We will now briefly describe the systems
in Table 2.1.

The StreamIt language allows expressing streaming applications using sequen-
tial pipeline, parallel split/join, and feedback loop primitives [117]. The StreamIt
primitives are fixed, unlike our approach, which allows adding new grouping primi-
tives. StreamIt supports events and reconfiguration using teleport messaging and a
re-initialization mechanism, respectively [118, 117].

Although using a new language has some advantages, using existing well-estab-
lished languages such as C and XML has the advantages that the user does not have
to learn a new language, that interfacing with existing libraries is easy, and that
mature tools such as compilers and debuggers are available. However, several tools
have been developed for StreamIt as well. The StreamIt Development tool assists
in visualizing and debugging StreamIt applications [80]. The StreamIt compiler can
detect parallelism in the application and map it onto homogeneous architectures [57].
This mapping is performed statically.

The Nizza framework [115] processes streaming multimedia applications using
data flow, similarly to our Hinch run time system. Nizza supports pipeline and task
parallelism but has no special support for data parallelism. Nizza has a dynamic
scheduler that favors tasks in the oldest iteration.

Nizza supports reentrant components. When a component is a bottleneck, the
component developer can make it reentrant by removing sequential dependencies
between different iterations. Nizza can then run multiple iterations of the component
currently, removing the bottleneck. Our Hinch run time system also supports
reentrant components, as we will explain in Section 3.3.3.

The support for dynamic reconfiguration is relatively basic in Nizza. The applica-
tion can only halt the entire data flow graph if it wants to perform reconfiguration
and restart Nizza afterwards. To our knowledge, Nizza does not support handling
user events.

<table>
<thead>
<tr>
<th>System</th>
<th>Parallelism</th>
<th>Load balancing</th>
<th>Reconfiguration</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>StreamIt</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Nizza</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MIRTIS</td>
<td>yes</td>
<td>yes</td>
<td>static</td>
<td>no</td>
</tr>
<tr>
<td>Decklight</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>ASSIST</td>
<td>yes</td>
<td>yes</td>
<td>dynamic</td>
<td>yes</td>
</tr>
<tr>
<td>GCM</td>
<td>yes</td>
<td>yes</td>
<td>dynamic</td>
<td>yes</td>
</tr>
<tr>
<td>System S</td>
<td>yes</td>
<td>yes</td>
<td>dynamic</td>
<td>yes</td>
</tr>
<tr>
<td>Streamware</td>
<td>yes</td>
<td>yes</td>
<td>static</td>
<td>no</td>
</tr>
<tr>
<td>Hinch</td>
<td>yes</td>
<td>yes</td>
<td>dynamic</td>
<td>yes</td>
</tr>
<tr>
<td>XSPCL</td>
<td>yes</td>
<td>yes</td>
<td>dynamic</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of related streaming frameworks
The Model Integrated Real-Time Imagine processing System (MIRTIS) models parallel image processing applications using data flow graphs. The application developer specifies the application structure using a graphical front-end. The MIRTIS interpreter then maps this structure statically onto a network of DSPs. Dynamic load balancing and run time reconfiguration are therefore not supported. MIRTIS supports pipeline, task and data parallelism. Events do not occur in the MIRTIS applications.

Decklight attacks the problem of integrating different multimedia frameworks and libraries. A Decklight application consists of a graph of components that interact with the lower level multimedia frameworks. The components communicate using streaming and event based primitives. Decklight supports dynamically reconfigurable applications. Since the components act as wrappers around existing frameworks, the types of parallelism Decklight exploits are limited to pipeline and task parallelism. A Decklight application only exploits data parallelism when the existing lower level framework already supports data parallelism.

The components in a Decklight application are relatively coarse-grained, as they perform the function of an entire existing framework. Since most existing frameworks do not exploit parallelism, the amount of parallelism in the application is limited. When existing frameworks do exploit parallelism, dynamic load balancing is difficult, if not impossible, since the existing frameworks exploit parallelism independently.

ASSIST and the Grid Component Model (GCM) are component-based models for Grid applications, which feature generic constructs for expressing parallelism and communication using events and streams. They support all forms of parallelism this way. ASSIST supports dynamic reconfiguration using a hierarchically structured application manager, which reconfigures the application based on availability of Grid resources. In GCM, components can contain Autonomic Managers that automatically perform reconfiguration based on dynamic measures, such as current performance. Using this automatic reconfiguration, ASSIST and GCM applications perform dynamic load balancing.

Since ASSIST and GCM are application models, they mainly focus on global application structure at the Grid level, whereas we also focus on more fine-grained aspects of stream programming, such as the various styles of parallelism and optimizations within a single compute node. The grain size of ASSIST components is also considerably larger than in our systems. What we consider an entire application, may be a single Grid component in the ASSIST model. GCM does not enforce a particular grain size, however, since a GCM component is the unit of distribution in a Grid environment, the grain size of GCM components is similar to ASSIST components. Our applications could very well fit within the ASSIST and GCM models, since these systems are designed with the same primitives in mind as our systems.

D-Stampede provides streaming communication primitives for distributed cluster applications. It is based on Space-Time Memory, which supports location transparency and performs distributed garbage collection of streaming data. Since D-Stampede does not address scheduling parallel tasks or managing the task graph, we have omitted it from Table 2.1.
System S focuses on data mining of streaming data\cite{2}, using the SPC run time system and the SPADE coordination language\cite{12, 54}. It schedules the application on a distributed architecture using a data flow graph, exploiting task and pipeline parallelism. SPADE exploits data parallelism by replicating components and automatically determines a static load balance at compile time. Although the system supports reconfiguration by dynamically creating streaming connections based on type information, it does not support handling events, to the best of our knowledge.

Streamware is similar to our Hinch run time system, as it targets streaming applications on general purpose CPUs\cite{61}. Streamware focuses on exploiting memory locality. It supports pipeline and task parallelism using a generic dependency construct. Data parallelism and dynamic load balancing are also implemented, however, reconfiguration and events are not.

Table 2.1 clearly shows that other frameworks for streaming applications only partly satisfy the demands for these applications. Our combined Hinch and XSPCL systems form the only framework that supports all parallelism types, dynamic load balancing, dynamic reconfiguration and events. Moreover, since our systems are built using a modular approach, they can easily be extended with other features, as we will explain in the remainder of this thesis.

2.2 Component-based frameworks

Many other projects agree that a component-based approach is necessary when developing complex systems and applications. Component models are used in varying domains, ranging from low-level hardware design to high-level high-performance distributed applications. All streaming frameworks mentioned in the previous section use components, since streams, by definition, flow between components. This section describes other component frameworks, and their relationship to our work.

Our component model resembles that of the Common Component Architecture (CCA)\cite{9}, which targets high performance computing (HPC) applications. The main difference is the granularity of components. We use fine-grain components and exploit parallelism within an application. CCA, on the other hand, focuses on course-grain components that may represent complete (parallel) applications.

Dryad\cite{70} uses a data flow graph with compute elements with communication ports, which is similar to our approach. These ports can be connected using various communication mechanisms, including files, shared memory FIFOs and TCP pipes, which all provide a form of streaming communication. The main difference is that Dryad targets coarse-grained data-parallel applications, where a single job processes gigabytes or more. In contrast, we primarily support fine grained parallelism, where the job input size is less than a megabyte. Another difference is that we also support multiple forms of task parallelism besides data parallelism. Furthermore, dynamic reconfigurability is not supported in Dryad, although plans exist to add support for this feature\cite{70}.

Fractal is a component model targeted at structuring software development in general\cite{27}. Key features are support for recursive component groups, sharing com-
2.3. PARALLEL PROGRAMMING

Components between different component groups, and reconfiguration. Our systems also support these advanced features, however, we do not support sharing components directly. In our approach, multiple instances of a component may share global state, which accomplishes the same. Fractal has been implemented using both Java and C [28, 100]. Koala is a component model targeted at embedded consumer electronic software [99]. Like Fractal, it focuses on structured software engineering only, and does not have special support for streaming or other communication types. DRRTS [113] is a component-based paradigm for complex embedded systems software.

2.3 Parallel programming

Many systems have been developed for supporting parallel programming. Danelutto and Skillicorn et al. give good overviews of current parallel programming techniques and requirements for parallel programming frameworks [45, 111]. Unlike our systems, which support an extensible range of parallelism types, the range of supported parallelism types is typically limited in other systems. Many systems only exploit data parallelism [30, 85, 108]. Some systems support a fixed set of different parallelism types [75]. Bal et al. describe several systems that combine task and data parallelism [19]. Only few systems allow extending the range of supported parallelism types [37, 44].

2.3.1 Skeletons and design patterns

Our systems have a clear relation to design patterns and algorithmic skeletons for parallel programming. The main advantages of using design patterns are code reuse, genericness, compositionality, the possibility of hierarchical refinement, separation of concerns, flexibility and extensibility [58]. Several design patterns are identified in [14, 87, 88]. In our systems, grouping components implement these patterns using the data flow primitives of our run time system. These components are building blocks for a hierarchical component tree. The range of supported design patterns can easily be extended by adding new grouping components.

Algorithmic skeletons, as defined by Cole [42], appear in our systems in two ways. First, each grouping component acts as a skeleton around its children. Second, a leaf component may act as a skeleton. Our systems support instantiating components using different parameters. A component developer can thus build template components that have one or more functions as parameters that define the behavior of the component. For example, a template component could provide generic support for single pixel operations. An application developer then gives a function implementing a specific single pixel operation as a parameter to the template component.

There are several parallel programming frameworks based on design patterns or skeletons, among which CO2P3S [86], eSkel [41], Lithium [6], P3L [17], and the Kuchen Skeleton library [78]. Since these frameworks focus on parallel programming in general, they have minimal support for streaming applications. For example, they lack
a generic interface for streaming communication that can be used throughout the application. The support for streaming applications consists of allowing patterns with streaming characteristics, such as a pipeline. Only the SkIE system mentions support for streaming\[18\].

Besides these generic frameworks, there are skeleton-based frameworks that focus on specific application domains. The SmartCam and Skipper projects have applied skeletons in the (embedded) image processing domain[32, 109]. Google uses the MapReduce skeleton for analyzing large data sets[10].

### 2.3.2 Coordination languages

Similarly to our SP@CE framework, several other systems address parallel programming using separate coordination and implementation languages. A component developer uses the implementation language for writing components. The implementation language is typically an imperative language, such as C, Fortran, or Java. While writing components, the component developer does not have to handle parallelism. An application developer uses the coordination language for combining the components into an application. A generic run time system then exploits parallelism by running multiple components concurrently. Using separate coordination and implementation languages thus provides the means to easily build reliable parallel applications[83].

Systems that use coordination languages include the ASSIST component-based framework, which uses the ASSISTcl coordination language[7]. The Fractal component model uses the architecture description language (ADL) for assembling components[100]. ADL contains elements of a coordination language. CODE and Hence are programming environments that are based on the data flow paradigm[25]. Using a graphical interface, the developer specifies a dependency graph of various sequential elements of the application, which also resembles using a coordination language. The Click system uses a coordination language for describing packet processing applications[76].

UML is often used as a high-level coordination or modeling language, for example in Rhapsody by I-Logix[65] and AndroMDA[13]. Rhapsody targets reliable embedded systems, for example for the aviation industry. AndroMDA targets enterprise applications. These tools generate the application from its UML description. Rhapsody also uses UML for verifying the correctness of the application and for preventing design flaws.

Two notable exceptions to using a separate coordination language are the X10 and Threaded-C languages[37, 116]. X10 is based on Java, while Threaded-C has its roots in C. Both languages include various high-level primitives for expressing parallelism. Coordination primitives have thus been incorporated into the implementation language, instead of having a separate coordination language. Although these languages provide valuable abstractions for parallel programming, they do not provide the higher abstractions that XSPCL and other coordination languages provide. Threaded-C applications can efficiently exploit parallelism using the EARTH
2.4 HETEROGENEOUS FRAMEWORKS

Although much work has been put into the design of X10, we are unaware of any performance figures that show the efficiency of X10 applications on parallel hardware.

2.4 Heterogeneous frameworks

The systems we mentioned above focus on homogeneous architectures. However, streaming applications are also an important class of applications on heterogeneous architectures. In this section, we will discuss systems for developing streaming applications for three different classes of heterogeneous architectures, namely multi processor system-on-chip (MPSoC) architecture targeted at multimedia applications, network processors, and CPU + GPU combinations. The systems typically address heterogeneity by designating a main processor that controls the other resources in the architecture.

Depending on the architecture, the communication mechanisms between the heterogeneous resources differ. Some systems take hardware design into account and use a similar communication mechanism between all resources by design. Other systems focus on software only and have to support different communication methods within one application. Our systems fall in the latter category.

2.4.1 Multimedia MPSoC architectures

Philips Research has developed many systems that address the difficulties of designing applications for heterogeneous MPSoC platforms, including TTL [129], YAPI [48] and C-HEAP [94]. These systems model an application as a Kahn Process Network (KPN) [73], which is a number of independent tasks that communicate using FIFO channels, which correspond to communication streams. The tasks and FIFO channels can be implemented in hardware or software, using shared or distributed memory. As these systems include hardware design, all resources use the same communication primitives. These systems mainly support task parallelism. They support data parallelism using a Single-Program-Multiple-Data approach. Load balancing is mostly done statically by mapping the tasks to fixed resources. TTL and C-HEAP have special primitives for reconfiguring the task graph and the communication streams at run time [62, 94].

The heterogenous Cell architecture is also an MPSoC architecture targeted at multimedia applications. We will discuss the various programming environments for this architecture and our approach in Chapter 5 which focuses on this architecture.

2.4.2 Network processors

Network packet processing is a challenging streaming application. Different types of packets from different network connections continuously enter the application at varying rates. The goal of the application is maximizing throughput.
Specialized network processors have been developed for these applications, such as the Intel IXP2xxx processor series [69]. These processors have complex heterogeneous architectures with general-purpose and special-purpose cores. The general purpose cores coordinate the processor and perform infrequent but complex packet processing tasks. The special-purpose cores may be fully programmable, or may only perform fixed tasks, such as encryption.

Several systems for network packet processing have been developed, including NP-Click [110], NEPAL [89], Shangri-La [38], Netbind [34], and the nameless systems described in [103] and [120]. Similarly to our systems, these systems split the application into different components, which in this case perform different packet processing tasks. Unlike our systems, which supply generic streaming primitives, these systems are domain-specific as they only support packet processing applications. Since packet processing consists of fine grained operations, these systems perform optimizations at the machine instruction level, whereas we focus on coarser-grained operations and optimizations.

Streamline is an optimized stream-based communication subsystem for operating systems [46]. It focuses on communication between the application, the operating system, and hardware devices, such as network devices. In contrast, we focus on streaming communication within an application. Actually, our systems and Streamline may be a good match, when input/output components built using our systems use Streamline communication primitives.

2.4.3 CPU + GPU combinations

Although Graphics Processing Units (GPUs) were originally designed for speeding up graphics output and had limited programmability, recent GPUs are increasingly programmable and support high performance applications. Using interfaces such as CUDA from Nvidia [98] and ATI Stream Technology [11] the CPU can use the GPU as a coprocessor and exploit its performance, which is typically orders of magnitude more than the CPU, since GPUs contain many parallel cores. This performance comes at a price, however: GPUs only support regularly structured algorithms.

Besides the above-mentioned vendor-specific interfaces, various systems have emerged that provide generic support for using GPUs. These systems include OpenCL [75], StarPU [10], OpenVIDIA [59], and Brook for GPUs [29]. Only Brook for GPUs has special primitives for streaming applications. StarPU models the application as a graph of communicating components. Unlike streaming applications, these components process single data items and not streams of data items. A component in a streaming application can however use StarPU, OpenCL and OpenVIDIA for improving performance, similarly to how we use the SPE coprocessors in the Cell processor, as we will explain in Section 5.4.
Chapter 3

Hinch: A run time system for parallel streaming applications

In order to attack the complexity of programming MPSoC architectures for streaming applications, we have developed the Hinch run time system, which shields the programmer from difficulties of the parallel architecture, such as load balancing, synchronization, and communication. Advanced constructs, such as end-user event handling and dynamic reconfiguration are fully supported. Due to its modular design, Hinch can easily be extended for supporting even more advanced applications in the future.

While it is well known how to support the individual properties mentioned above, to the best of our knowledge, Hinch is the first system that supports all, while greatly simplifying the MPSoC programmers’ task. Moreover, Section 3.4 will show Hinch efficiently exploits parallel architectures. The major difficulties we encounter are combining task- and data-parallelism\[19\], supporting dynamic reconfiguration, and handling asynchronous events.

Hinch targets streaming applications, as these applications are commonly run on MPSoC architectures. Streaming applications are typically multimedia applications, such as video decoding, or networking applications, such as packet processing. Many other applications can also be expressed with streaming, including high-performance applications. In the future, we expect that the complexity of these architectures and applications will increase for several reasons:

- Streaming applications increasingly require complex processing, for example new multimedia coding algorithms.

• The processing is applied to increasing amounts of data per time-unit, such as multichannel HDTV.

• Streaming applications often exhibit much potential parallelism. Task parallelism is available by extracting different tasks from the application, such as decompression and filtering. Independent tasks can always run concurrently while a pipeline approach allows running dependent tasks concurrently on multiple data items. Data parallelism is often available within each task by splitting the input data into multiple parts.

• Hardware vendors have already opted for multi-core processors, such as the Cell[72], Network processors[69], Xeon, and Opteron, as the key to speed, because it is hard to crank up clock speeds at the same rate as in the past.

By carefully studying several applications, we obtained a core set of properties that a run time system for streaming applications should cater to. We derived requirements for Hinch from these properties. From these requirements, we have developed a flexible run time system which efficiently runs complex applications on MPSoC platforms.

Within the SP@CE framework, which we described in Section 1.5, Hinch is used as a lower layer. Other layers with an even higher abstraction level are built on top of Hinch. The application developer thus typically is not aware of the full Hinch API as most of it is abstracted by the higher layers.

In this chapter we will first describe the requirements for Hinch in Section 3.1, followed by a detailed description of the design and implementation of the various features of Hinch in Sections 3.2 and 3.3. Section 3.4 presents a performance evaluation of Hinch, in which we examine its overhead and parallelization efficiency. Finally, Section 3.5 summarizes and concludes this chapter.

3.1 Requirements

The requirements for Hinch are composed of three parts. First, streaming applications require that Hinch provides certain features, such as communication primitives and means for composing applications out of several components. Second, MPSoC architectures have special properties, such as multiple compute cores, which Hinch should address. Third, Hinch should be easy to use for both application and component developers. Table 3.1 summarizes the requirements for Hinch derived from the application, the architecture, and the developer perspective.

Some requirements overlap, for example both the applications and the MPSoC architectures require support for parallelism and communication. Therefore we have grouped the requirements into six categories, as shown in Table 3.2. Some requirements, such as having a simple and abstract API, apply to multiple categories. In the following subsections, we will describe the requirements we have identified for Hinch by category.
3.1. REQUIREMENTS

<table>
<thead>
<tr>
<th>Perspective</th>
<th>Contains / Uses</th>
<th>Requirement for Hinch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Compute kernels</td>
<td>Components</td>
</tr>
<tr>
<td>Application</td>
<td>Tree structure</td>
<td>Grouping structures</td>
</tr>
<tr>
<td>Application</td>
<td>Task &amp; Data Parallelism</td>
<td>Express Parallelism</td>
</tr>
<tr>
<td>Application</td>
<td>Streaming communication</td>
<td>Support communication</td>
</tr>
<tr>
<td>Application</td>
<td>Event communication</td>
<td>Support communication</td>
</tr>
<tr>
<td>Application</td>
<td>Multiple iterations</td>
<td>Schedule iterations</td>
</tr>
<tr>
<td>Application</td>
<td>Dynamism</td>
<td>Support Reconfigurability</td>
</tr>
<tr>
<td>Architecture</td>
<td>Portability</td>
<td>Abstract API</td>
</tr>
<tr>
<td>Architecture</td>
<td>MPSoC</td>
<td>Exploit parallelism</td>
</tr>
<tr>
<td>Architecture</td>
<td>Communication mechanisms</td>
<td>Support communication</td>
</tr>
<tr>
<td>Developer</td>
<td>Hinch run time system</td>
<td>Simple API</td>
</tr>
<tr>
<td>Developer</td>
<td>Optimizations</td>
<td>Allow optimizations</td>
</tr>
</tbody>
</table>

Table 3.1: Requirements for the Hinch run time system

<table>
<thead>
<tr>
<th>Category</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
<td>- Components</td>
</tr>
<tr>
<td></td>
<td>- Grouping structures</td>
</tr>
<tr>
<td>Iterations</td>
<td>- Schedule iterations</td>
</tr>
<tr>
<td>Parallelism</td>
<td>- Express parallelism</td>
</tr>
<tr>
<td>Communication</td>
<td>- Support communication</td>
</tr>
<tr>
<td>Reconfigurability</td>
<td>- Support reconfigurability</td>
</tr>
<tr>
<td>Ease-of-use</td>
<td>- Allow optimizations</td>
</tr>
</tbody>
</table>

Table 3.2: Requirement categories

3.1.1 Components

A streaming application consists of several kernels that perform a specific operation, such as a geometric transformation of an image or an image filter. Managing large numbers of kernels requires wrapping kernels inside components with a generic interface. Figure 3.1a shows an example of a streaming application that applies a filter to a video file containing successive RGB images. The arrows indicate streams, and the ellipses are components. The input component reads an input file from disk and outputs R, G, and B streams. Three filter kernels, that are wrapped inside components, apply the filter. Finally, the output component writes the combined data from the R’, G’, and B’ streams to disk.

Since the generic component interface is implemented by the component developer, and used by the application developer, it is kept as simple as possible. On the other hand, it must be advanced enough to support all possible component interactions. Ideally, it should also be flexible and accommodate the requirements of
Managing large applications with many components requires support for grouping components into higher-level components. The resulting application is organized hierarchically like a tree with kernels at the leaf nodes. Each tree node specifies the scheduling dependencies between its children. This approach resembles using algorithmic skeletons [42], as grouping components correspond to skeletons.

The component tree thus represents a data flow graph. Figure 3.1b shows the grouping components and the data flow graph of the application in Figure 3.1a. The arrows indicate data flow dependencies. The sequential group creates sequential dependencies between its children: The Input component runs before the Filter components, which in turn run before the output Component. The parallel group creates independent data flow paths which allows running the children of the group concurrently. Figure 3.1c shows the corresponding component tree.

The grouping interface should be as simple as possible, as the application developer uses it for composing applications. On the other hand, it should support advanced constructs such as dynamic reconfigurability, as we will explain in Section 3.1.5. Moreover, Hinch should not impose limits on the structure of groups. A grouping component developer should be able to build groups that closely match the application structure.

Figure 3.1: Example streaming application that filters RGB images
3.1. REQUIREMENTS

3.1.2 Iterations

A streaming application runs by executing iterations of the data flow graph, in which each component and therefore each kernel runs once. Hinch must therefore schedule the execution of the data flow graph, taking into account the scheduling order that corresponds to the component tree. In the example application of Figure 3.1, one iteration would process one image from the input video stream.

Normally the application halts when it runs out of input data, or when the end-user terminates it. The application of Figure 3.1 would terminate at the end of the input file, for example. For performing benchmarks, running a fixed number of iterations should also be possible. Unlike most other applications, which have only one fixed mode of operation, Hinch should support both an infinite as well a finite iteration count.

Hinch must also handle multiple different termination conditions. The application can either terminate when a fixed number of iterations have run, or when it detects that the end of input is reached, or when a (user) event signals the application to stop. Hinch must detect these conditions and gracefully terminate the application in all cases.

3.1.3 Parallelism

Streaming applications exhibit both task- and data-parallelism. Task parallelism occurs when independent subtrees of the component tree can be scheduled concurrently, for example, the Filter components in Figure 3.1. Also, multiple iterations can be scheduled concurrently resulting in pipeline parallelism. Data parallelism is available when the computations inside a component are independent, for example, when the kernel in a component repeatedly performs the same operation for each pixel in an image. Multiple instances of these components can be run in parallel, where each instance performs part of the computation. For example, the application of Figure 3.1 could replicate the Filter components and configure the replicas to call the kernel only for a part of the input image.

Hinch must be able to map the application on the MPSoC architecture such that both forms of parallelism are exploited. However, combining task- and data parallelism is difficult [19]. Hinch needs to perform load balancing, synchronize parallel tasks, and allow communication between tasks that concurrently use different resources. As we will show in Section 3.2.2, Hinch supports all these requirements.

3.1.4 Communication

In streaming applications components primarily communicate using streams, which act as FIFO channels. In each iteration, a component consumes data from its input streams and produces data in its output streams. For example, the topmost filter component in Figure 3.1 consumes the R stream and produces the R’ stream. When a component reads data from sources external to the application, such as the input component in Figure 3.1, the component only has output streams. Similarly,
a component that writes data to external resources only has input streams, that contain the data the component has to write. When an application performs different computations on the same data, multiple consuming components access the same stream.

Some components access data that was also used in previous iterations. For example if the component performs motion estimation on an image stream, it needs to access the previous image and the current image in the stream. This access type is similar to the peek functionality in StreamIt\cite{117}.

After the data has been consumed by all consuming components, it can be discarded and the memory occupied by the data can be freed or reused in a later iteration. Hinch should perform memory management and synchronization between the different components that access a stream.

In one application streams may use different communication mechanisms, depending on the communication capabilities of the hardware. For example, when shared memory is available, Hinch can store the data in a stream in a shared memory buffer. With distributed memory, Hinch has to transfer the data between connected components using a communication network.

Supporting these functional requirements in a single system is not easy, because they influence each other. For example, while performing memory management Hinch must be aware that a component accesses data from previous iterations.

Besides the functional requirements, Hinch should provide a simple interface to the application developer for creating streams and connecting them to components. Towards the component developer, it should provide a simple interface for reading from and writing to streams, that hides the underlying communication structure. The combined functional and interface requirements make supporting streaming communication a challenging task.

Besides streaming communication, which is predictable as it occurs in every iteration, unpredictable events occur in streaming applications. For example, a component informs another component when a certain situation occurs or the end-user generates an event by pressing a key. Hinch should provide a simple abstract interface for sending and receiving events and hide the underlying communication structure(s) from the component developer.

### 3.1.5 Dynamic reconfigurability

Many streaming applications need support for reconfiguration, which usually occurs after user interactions, e.g., the user wants to add a picture-in-picture to the television screen. Resource availability can also trigger reconfiguration. For example, quality is scaled down when less bandwidth is available. This requirement especially holds for applications with real-time constraints, for example when an application displays a video stream at a constant rate. When an application does not meet its deadlines because of limited resources, it should detect it and act accordingly, for example, by reducing output quality.
3.1. REQUIREMENTS

Reconfiguration can be performed by adjusting parameters of components (component reconfiguration) or by adding and removing components while the application is running (application reconfiguration). Hinch should thus allow the application to modify its behavior at run time, which entails four subtasks:

- Hinch must have some mechanism for signaling that reconfiguration is required.

- Individual components must be reconfigurable. For example, the filter components in Figure 3.1 could be reconfigured to use different filter parameters. Many image filters contain threshold values, which can be altered, for example. Hinch has to be able to tell the component it must alter its parameters, in case the end-user wants to. Besides the functional interface for running the component, the component interface must therefore include a non-functional part that supports reconfiguration. Since each component has different parameters, the interface should be generic and extendible to the requirements of future components. To aid the component developer, the interface should also be as simple as possible.

- The whole component tree must be reconfigurable. For example, the end-user might want to add a second output component to the application in Figure 3.1 that displays the output on a viewscreen. Hinch must allow the application developer to create and destroy components, and to modify the component tree at run time. The modified parts can be single components or entire subtrees of components. The extra non-functional interface that provides this functionality, should be as simple as possible for the application developer.

- When components are added, Hinch must connect communication streams to these components. These streams can be either existing streams, which get new connections, or newly created streams. When components are removed, the opposite holds. Hinch must thus support the creation and destruction of streams at run time, which again requires extra non-functional interfaces besides the functional interfaces. A similar argument holds for components that communicate using events.

Supporting dynamic reconfigurability thus affects all parts of the run time system. It requires support by individual components, components groups, streams, and the internal scheduling mechanism that manages the parallel resources, amongst others. All dynamic structures require non-functional reconfiguration interfaces, besides their functional interface. All parts of Hinch thus have to be designed with support for dynamic reconfigurability in mind.
3.1.6 Ease-of-use

When a new architecture becomes available, the application developer should be able to quickly deploy applications on this new architecture. Hinch should therefore have an abstract platform-independent API that hides the low-level details of the architecture. Like all APIs, the API should be as simple as possible, to aid the developer that uses the API.

However, Hinch should still allow using low-level optimizations, for example architecture-specific SIMD instructions, as these optimizations may significantly increase performance\cite{104}. We therefore need to seek a balance between platform independence and low-level optimizations.

The same argument also holds for Hinch itself. Internally, Hinch should be mainly platform independent, which makes porting it to other architectures easy. Platform independent code also avoids duplicate code, as one would have to re-implement and maintain the code for each platform when the code is platform dependent. However, for optimal performance, Hinch has to exploit architecture-specific optimizations where possible. We need to seek a balance between platform independence and using low-level optimizations for Hinch as well.

Since writing parallel applications is difficult, Hinch should hide as much parallelism from the developer as possible behind its simple API. Ideally, the application developer only expresses what can be run in parallel, and Hinch takes care of how the application is run in parallel. The component developer ideally only writes sequential code.

However, this approach puts restrictions on the application structure and the amount of parallelism that can be exploited. For example, a component can only be active in one iteration at a time, because Hinch can not assume that the sequential component code is safe for parallel execution. Ideally, Hinch should therefore also have methods for overcoming these restrictions, for example an interface call which the component uses to tell Hinch that the component can execute another concurrent iteration. The drawback of using these methods is that the developer is made aware of parallelism again. Therefore using these methods should be optional and not mandatory.

<table>
<thead>
<tr>
<th>Requirement category</th>
<th>Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
<td>Grouping components, Data flow process network</td>
</tr>
<tr>
<td>Parallelism</td>
<td>Grouping components, Job queue</td>
</tr>
<tr>
<td>Communication</td>
<td>Stream module, Event queue</td>
</tr>
<tr>
<td>Reconfigurability</td>
<td>Manager group, Event queue, Component interface</td>
</tr>
<tr>
<td>Iterations</td>
<td>Loop group, Manager group, Event queue</td>
</tr>
<tr>
<td>Ease-of-use</td>
<td>Modular system, Optional advanced functions</td>
</tr>
</tbody>
</table>

Table 3.3: Correspondence between requirements and design
3.2 Design

This section explains the design of Hinch and how it addresses the requirements described above. The design closely follows the requirements. Table 3.3 shows the correspondence between the requirement categories we identified in the previous section and the design. It also serves as a summary of the design of Hinch.

3.2.1 Components

A Hinch application consists of components that are actors in a data flow process network. This approach fits streaming applications well, as it provides many advantages:

- A data flow network is an intuitive way of splitting a large application into multiple small components that can then be developed independently.
- Splitting the application into components allows reuse of code, because a component can be instantiated multiple times in one data flow graph.
- The data flow dependencies are easily derived from the hierarchical component tree.
- Hinch can exploit parallelism by running multiple independent paths in the data flow graph concurrently.
- A data flow network allows full dynamic load balancing on a parallel architecture, as we will explain in Section 3.2.2.
- A data flow network can be modified at run time, which is required for dynamic reconfiguration. We explain the design of dynamic reconfiguration in detail in Section 3.2.4.
- A component in a data flow network does not require a complex interface. The component interface only needs basic functions for creating, destroying, and running components. The interface should also provide basic functions for connecting and disconnecting streams to the component.

The application runs by executing iterations of the data flow graph. In an iteration each actor fires once. One firing corresponds to running one iteration of a component. In a video processing application, a component typically contains an image processing kernel. One iteration then consists of processing one image frame from the video stream. A graph iteration begins by scheduling the component(s) at the start of the data flow graph. The other components are scheduled as soon as their predecessors in the data flow graph have finished executing.

The data flow graph is built using special grouping components, which are the intermediate nodes in a component tree that represents the application. Grouping components contain child components and a data flow graph that specifies the
scheduling dependencies between these children. These groups can be nested recursively. When Hinch runs an iteration of a grouping component, Hinch runs an iteration of the data flow graph inside the grouping component. When this inner iteration has finished, the successors of the grouping component in the higher level data flow graph are scheduled.

Figure 3.2 shows an example data flow graph. The boxes and circles indicate (grouping) components, which contain a number that indicates the scheduling order. Grouping component 1 thus contains components 2, 3, and 6, where component 3 is another grouping component. The arrows indicate data flow dependencies. Note that components 4 and 5 may run concurrently, since they are on independent paths in the data flow graph.

All components have a generic interface that provides an abstraction of the component to Hinch. Using this interface, Hinch handles all components equivalently, regardless of their function. The interface contains functions for creating, configuring and destroying instances of the component, getting its properties, and running an iteration of the component. Besides these functions, a component contains references to the streams it is connected to. A grouping component has extra functions for adding and removing child components.

The component interface is kept as simple as possible, which allows quick incremental development of components. A component developer can first implement the basic functionality of a component and test it, before adding more advanced features, such as support for data parallelism and other configuration options.

Even though the component interface is simple, it supports all necessary features, including advanced constructs such as dynamic reconfiguration, as we will show in Section 3.2.4.

The Hinch application model resembles that of Koala. A Hinch component iteration corresponds to the execution of a Koala task. As in Koala, components can be recursively grouped, and components can only be coupled if their interfaces match, as we will explain in Section 3.2.3.

3.2.2 Parallelism

The data flow graphs inside the grouping components express the parallelism in the application. The most common data flow graph types are shown in Figure 3.3. The circles indicate components and the arrows indicate data flow dependencies.
Parallelism types

When the data flow graph in a grouping component indicates that the children run sequentially, Hinch exploits pipeline parallelism by running multiple iterations concurrently. For example, in Figure 3.3a, component A could be active in the second iteration while B is still performing its first iteration.

A data flow graph with independent paths for each child component expresses task parallelism. In Figure 3.3b, components C and D have no dependency and can therefore be concurrently active in the same iteration.

A grouping component can support data parallelism by creating multiple copies of its children. Using the component configuration function, the grouping component configures each copy to do only a part of the computation. Data parallelism only works with components that support these configuration commands. The grouping component then puts these children in a data flow graph that is similar to the normal task parallel data flow graph, as shown in Figure 3.3b.

When a data parallel component accesses its streams, it has access to all data in these streams, although it only processes part of this data. This approach has the advantage that the data does not have to be split up into multiple different streams when data parallelism is used. All data parallel components simply access the same stream. Non-data parallel components and data-parallel components can easily be combined this way.

Another major advantage of this approach is that it allows a component to easily access data outside its own part of the input. For example, in an imaging application, a convolution kernel can run in parallel by having each data-parallel kernel process a slice of the input image, which is a number of consecutive rows. The kernel also
Building a data parallel component requires some effort from the component developer. The component requires a cloning function, which the grouping component uses for creating copies of the component. As the grouping component tells the component it has to perform part of the computation, the component also needs to support this configuration command. Finally, the running function of the component must perform only part of the computation, corresponding to the component’s configuration.

One might argue that data parallelism can also be obtained by reducing the computation granularity of an iteration, and then running multiple iterations in parallel. When a data-parallel kernel needs input data that is produced in other iterations, building such a kernel will be very difficult, if not impossible. For example, the input data for a data-parallel convolution kernel would reside in three different data buffers. One of these buffers belongs to the next iteration, therefore complicated dependencies need to be generated between different iterations. In our approach, developing a data-parallel convolution kernel is easy since the whole image is in a contiguous memory area.

Hinch exposes as little parallelism to the component developer as possible. In our design, a component developer does not have to write any parallel code. However, for optimal performance, Hinch provides some features in which the component developer is aware of parallelism. For example, when writing a data parallel component, the developer must be confident that the computation can be split into independent parts. Still, the implementation of a data parallel component only requires sequential code, without any explicit synchronization commands.

Since the application runs in parallel, Hinch enforces some restrictions on the component developer. The restrictions include for example that multiple components can not share global data structures, as Hinch can not enforce exclusive access to these structures and race conditions could occur. Components are also not allowed to enforce exclusive access themselves, for example using mutual exclusion primitives (locks). These locks might interfere with the locking strategy in Hinch, resulting in deadlock, for example.
3.2. DESIGN

Job queue

When the application runs on a shared memory MPSoC architecture, a central job queue accomplishes automatic load balancing. Distributed memory machines use a distributed job queue, for example using the cluster-aware random stealing algorithm\[95\]. A parallel application consists of multiple threads that continuously execute jobs from the queue, and add new jobs to the queue that are ready to run. We avoid expensive context switches by bounding the number of threads to the number of threads supported by the architecture. For example, on a quad-core CPU, Hinch uses no more than four threads.

A job in Hinch consists of running a dataflow actor, which corresponds to one iteration of a component. While an actor runs, it uses the data flow graph to send tokens to other actors which enables the other actors to run. When an actor is ready, Hinch puts a job that runs the actor onto the job queue. This way, Hinch exploits parallelism automatically: When multiple actors are ready, the job queue contains multiple jobs. Different threads remove these jobs and execute them in parallel. Hinch performs load balancing automatically: Since all threads are capable of running all jobs, they can simply pick the first job from the queue. As long as the job queue contains jobs, all threads will be busy executing jobs and load balancing will be optimal.

A job queue is a powerful mechanism as it allows exploiting any type of parallelism. It perfectly fits the data flow model and it allows a separation of concerns regarding the exploitation of parallelism. Hinch is responsible for synchronizing all accesses to the queue. The data flow graph ensures that a job is only put onto the queue when it is ready to be run. For debugging purposes, a job queue can easily be configured to run an application sequentially and deterministically, by using a single thread instead of multiple threads of control.

The Hinch API hides the job queue from the application and the basic component developers. A group component developer creates dependencies between jobs and components and therefore needs some knowledge about jobs and dependencies. By hiding the details of scheduling parallel jobs, and performing automatic load balancing, the job queue is a valuable tool for developing parallel applications.

3.2.3 Communication

Components can communicate using streams. The Hinch API for accessing streams provides similar streaming functions as the StreamIt\[117\] language and a communication abstraction similar to Space-Time Memory\[102\]. Each component has a fixed number of input and output ports. An application developer connects streams to these ports. A component does not know with whom it is communicating. It only knows which streams are connected to its ports. This way, an application developer can easily reuse a component in another part of the application or in a different application.

Multiple producing or consuming components are connected to a single stream when a set of components is configured for data parallelism. Hinch must support
CHAPTER 3. THE HINCH RUN TIME SYSTEM

Figure 3.5: Dependencies between successive sliced components

this feature and keep track of the status of each component. The component status includes the most recently accessed data buffer, for example. The stream module must also store the number of data buffers from previous iterations that a component accesses, to determine when a buffer is no longer in use.

The stream module also supports multicast streams by connecting multiple components to the same stream. This way, multicast is efficient because the data does not have to be copied for each component. All components simply access the same data buffer from the same stream. Beltway buffers have a similar zero-copy design[47].

When both the producer and the consumer of a stream run in parallel using slicing, care must be taken to apply the correct dependencies between the producing and consuming components. Often the slicing layout is equal, which means both the producer and the consumer use the same data distribution. When the slicing layout is equal, and the consumer only accesses the slice that is produced by the producer that is assigned to that slice, there can be a simple producer-consumer dependency, as shown in Figure 3.5a. When the slicing layout is equal, and the consumer also accesses data from its neighboring slices, dependencies must be added from the neighboring producers, as shown in Figure 3.5b. When the number of slices differs between the consumer and the producer, a similar complex dependency structure has to be used. A solution that will always work is introducing a synchronization point between the producer and the consumer, as shown in Figure 3.5c. Each consumer will only be scheduled when all producers have finished.

In our design we have chosen not to handle these complex dependencies in the stream module. This approach would add complexity which is unnecessary because data flow dependencies are already handled by grouping components. The dependencies of Figure 3.5a can be handled by embedding a sequential group, that processes one slice, within a parallel group, that instantiates the sequential group for each slice. The dependencies of Figure 3.5c are handled by embedding two data parallel groups within a sequential group. For the dependencies of Figure 3.5b a special grouping component has to be created that enforces these dependencies. Section 3.3.3 will describe this grouping component.

A component accesses a stream using a simple interface, which is equal for all access types, including reading and writing. At the start of an iteration, it requests
new data buffer from the stream. The stream module returns the address of the buffer in memory. For input streams, the data buffer contains input data that was produced by a component that was scheduled earlier. For output streams, the data buffer contains room for the output data of the component. After the component has finished accessing the buffer, it releases the buffer. When all components that are connected to a stream have released a buffer, Hinch frees the buffer’s memory or reuses it for a later iteration.

Hinch handles asynchronous events by buffering them in an abstract event queue. When a component runs, it may put new events in an event queue, for example, when it detects the user has pressed a key. A component may also poll an event queue for new events. If an event is in the queue, it is removed and returned to the polling component. Otherwise, Hinch informs the component that the queue is empty. A component can retrieve all pending events in an event queue by polling until the queue is empty.

### 3.2.4 Reconfiguration and event handling

Hinch supports reconfigurability throughout the whole run time system. All structures, such as components, streams, and event queues, are designed with reconfigurability in mind. Hinch does not only have functions that create new instances of these structures, but also functions that reconfigure and destroy existing instances. Furthermore, all internal structures can handle dynamic changes, which often means that Hinch has to allocate resources, such as memory, dynamically. For example, each stream contains a list with the status of the connected components, whose size must be increased when the list is full and Hinch wants to connect a new component to the stream.

Especially in reconfigurable applications, in which many structures are created and destroyed while the application is running, it is important that each structure has a clean destruction function that frees all used resources, including any dynamically allocated resources. Creating a clean destruction function adds to the complexity of dynamic reconfigurability, as these functions are typically not necessary in non-reconfigurable applications. Supporting dynamic reconfiguration is therefore difficult because the various structures and the accompanying management functions are more complex than in static applications.

Besides the internal Hinch structures, the component interface also supports dynamic reconfigurability. Components can be dynamically created, destroyed, grouped, and connected at run time. Using the configuration function, an application developer and/or Hinch can send (re)configuration commands to a component. A component developer implements this functionality.

The grouping component interface has a replace function that replaces a child component by a component that has the same I/O interface. Without the replace function, replacing a component would have to be accomplished by removing all data flow dependencies to and from the old child, removing the child, adding the new child, and creating these dependencies to and from the new child. Similarly, all
streams have to be disconnected from the old child, and new connections have to be made to the new child. With the replace function, the dependencies and streaming connections do not have to be destroyed and created. The grouping component can simply transfer the dependencies to the new child. Similarly, it can transfer the streaming connections as the old and the new child have the same I/O interface.

Implementing an optimized replace function is optional. A grouping component developer can choose to use a generic default replace function, which uses the add and delete functions of the grouping component. For simple grouping components, with few dependencies, implementing an optimized replace function is often not worth the effort. In this case, the default replace function suffices. This default replace function is also useful for debugging and prototyping purposes: A grouping component developer can first implement the basic functionality of the grouping component, and add optimizations, such as a replace function, later.

A special manager grouping component manages a reconfigurable subgraph in the application, as shown in Figure 3.6. Each manager listens to an event queue, and reconfigures its contained subgraph in response to the events it receives from this event queue. The manager keeps track of the number of active subgraph iterations and suspends execution of new iterations when reconfiguration is required. This way, it can make its subgraph idle.

Upon reconfiguration, the manager calls an application-specific reconfiguration function. This function reconfigures the application by adding and removing components, or by sending reconfiguration commands using the component configuration interface. The application developer supplies this function to the manager. This way, we achieve a desirable separation of concerns: The manager handles all generic difficulties regarding reconfiguration, including halting and restarting the application. The application-specific commands are in a separate function. Since the manager provides a generic means for building reconfigurable applications, it corresponds to a behavioral skeleton. Such skeletons are also used in the Grid Component Framework[8].

To avoid race conditions, reconfiguration is only performed when all components in the contained subgraph are idle. Although it might be possible to reconfigure non-idle graphs, thereby increasing performance, the various synchronization difficulties that would arise in this case have led to the decision not to do this. It would expose
many difficulties to the application developer, which is undesirable, because of the ease-of-use requirement. Moreover, as reconfiguration typically occurs infrequently, the effort of handling these difficulties does not outweigh the possible performance gains.

### 3.2.5 Iterations

A Hinch application runs multiple iterations by scheduling a new iteration as soon as an iteration has finished. In our design, a loop grouping component performs this task. This component is at the top of the component tree.

As explained above, the manager grouping component can temporarily suspend execution of iterations in case reconfiguration is required. A manager can also stop the application by permanently suspending all iterations. For example, when a component detects the end of input, it can send an event to the manager to suspend all future iterations, which effectively halts the application.

All functionality related to running iterations is thus provided by Hinch and two grouping components. The application and component developer use the simple event queue interface for influencing how iterations are scheduled.

### 3.2.6 Ease-of-use

Hinch has a modular design, in which each module provides specific functionality to the other modules, the application and the components. A modular design with a clear interface towards the application and component developer for each module simplifies the effort for the developer. Many modules have optional features. A developer can thus incrementally develop applications and/or components, by first building an initial prototype, and then adding more advanced features, which for example improve performance.

For ensuring portability, most modules do not use any architecture-specific functionality. Whenever different architectures have different means of performing a certain function, a separate module abstracts the architectural details behind a generic interface. Architecture-specific low-level optimizations are only performed in these modules. The porting effort of Hinch itself is reduced to a minimum this way and a good balance between platform independence and using low-level optimizations is achieved.

Hinch is very flexible towards the use of architecture-specific optimizations in components. A component developer can always choose to build a platform independent component and not use any architecture-specific optimizations. These components will be highly portable but they will be relatively slow. When the compiler supports it, a component developer can also include architecture-specific optimizations, thereby giving up portability. Hybrid components, that use a different implementation depending on the target architecture, are also possible. By giving the component developer this freedom, we achieve a good balance between platform independence and using low-level optimizations.
Table 3.4: Modules in the implementation of Hinch

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>Component management</td>
</tr>
<tr>
<td>Grouping component</td>
<td>Grouping component management</td>
</tr>
<tr>
<td>Stream</td>
<td>Streaming communication</td>
</tr>
<tr>
<td>Event queue</td>
<td>Event communication</td>
</tr>
<tr>
<td>Thread</td>
<td>Low-level multi-threading primitives</td>
</tr>
<tr>
<td>Timing</td>
<td>Low-level timing using cycle counter</td>
</tr>
<tr>
<td>Job</td>
<td>Individual job management</td>
</tr>
<tr>
<td>Dependency list</td>
<td>Manage data flow dependencies between jobs</td>
</tr>
<tr>
<td>Job queue</td>
<td>Run multiple jobs in parallel using multi-threading</td>
</tr>
</tbody>
</table>

3.3 Implementation

Following its design, Hinch consists of several modules, which are listed in Table 3.4. A module is a self-contained object that provides some functionality to the developer and to other parts of the runtime system using an interface that abstracts the difficulties of the provided functionality. The platform-independent modules are written in C, which allows running Hinch on a variety of architectures and operating systems. Currently, Hinch has been successfully run on a simulator for the SpaceCake architecture [114] which uses the eCos operating system, on the SPARC architecture, using Solaris, and x86 and Cell architectures, with the GNU/Linux operating system. All components are also written in C, although they can be developed in any language as long as they can be linked to the application and adhere to the proper interface.

The only two modules that differ between architectures are the threading module and the timing module. The threading module provides generic functions for creating and synchronizing threads. On eCos, the module maps these functions onto the eCos threading primitives. On Linux and Solaris, it uses the pthreads library. For investigating the overhead of the multi-threading primitives, Hinch can also be compiled without multi-threading support. The application is then run sequentially and the functions from the threading module are either empty or trigger an error when they are called.

The timing module uses architecture-specific cycle-counter registers for measuring execution time. Using these registers, Hinch gathers performance statistics at low overhead. In case the architecture does not have a cycle-counter register, the module falls back to using system calls that return time information.

The main structures in Hinch are the component and the job. The component structure includes the functions provided by the component and the streams connected to it. Jobs are the nodes that build the data flow graph, as we will explain in Section 3.3.2.

Although C does not support object-oriented features such as interfaces, classes, and inheritance, all modules are implemented in an object-oriented manner. For
example, a component contains a function that destroys that instance of the component, and a job contains a function for running the job.

Hinch could have been implemented in an object-oriented language such as Java. However, these languages have other disadvantages:

- The developer should be able to apply low-level optimizations, such as SIMD optimizations, for achieving optimal performance. In high-level object-oriented languages such as Java, including these optimizations is difficult, if not impossible.

- The garbage collector in object-oriented languages causes unpredictable behavior in the application. Streaming applications often require a runtime system that is predictable. This requirement especially holds for multimedia applications, which need to timely output audio and video data.

In the domain of high performance computing, the disadvantages of using an object-oriented language are less prevalent. Using a similar design, a runtime system for high performance streaming applications can be created. For example, Hinch could be implemented on top of the Ibis communication library, which provides efficient communication primitives for Java applications [96].

The application developer must adhere to certain rules when using the Hinch API. For example, the destruction function of a structure managed by Hinch may only be called when the application does not actively use the structure. While developing the application, the application developer can enable several checks in Hinch, which raise an error and terminate the application if the Hinch API is incorrectly used. When the application fully works, the application developer can disable these checks for a small performance benefit.

In the remainder of this section, we describe the various Hinch modules in detail. Section 3.3.1 describes component and grouping component structures, and how they build the component tree. Section 3.3.2 describes how jobs and dependency lists form the data flow graph of the application. Section 3.3.3 describes how Hinch exploits parallelism, by giving more details about components, grouping components, and dependencies. It also describes the job queue, which is a key structure for exploiting parallelism. The implementation of the stream and event communication mechanisms are discussed in Sections 3.3.4 and 3.3.5 respectively. Finally, Section 3.3.6 describes the implementation of dynamic reconfigurability.

3.3.1 Component tree

An application developer builds the component tree in an object-oriented manner using component structures, which contain all information about a component that is relevant to Hinch. Because C does not support object-oriented features such as inheritance, a single data structure represents all components, including grouping components. This structure allows treating all components indifferently. Hinch contains a component module and a grouping component module which manage the
generic parts of the component. The component developer is responsible for the other parts of the component, including:

- Component-specific static data. To assist the component developer, each component has a reference to static data which can be used to describe the state and configuration of the component. A component initializes this data when it is created and updates the data when it is reconfigured. When it is run, the component can also modify the data, for example to count the number of iterations executed by the component. A component is responsible for freeing all allocated memory for this data when the component is destroyed.

  For avoiding race conditions due to mutual access to the static data, Hinch ensures that only one thread is active in a specific component instance at a time, which means only one thread accesses the static data at a time. The component developer thus does not have to handle mutual exclusion. There is one exception to this rule: The run function may be reentrant, which Section 3.3.3 will explained.

- I/O streams. Each component has a number of I/O ports, to which streams are connected. The number of I/O ports depends on the component type. For each I/O port, Hinch keeps track which stream is connected to it. The component developer does not use these fields directly. A component can easily access streams using the port number, which Hinch maps to the appropriate stream.

- I/O port properties. These properties determine the kind of stream that can be connected to the port and the connection type. The connection type specifies whether the component is reading from or writing to the stream and the number of previous iterations that the component accesses. Some components need to access data they read or produced in previous iterations.

  A component initializes these properties when it is created, which allows Hinch to connect proper streams to each I/O port. Hinch only connects streams between compatible I/O ports.

- new function. This function creates a new instance of the component. It is the only function whose signature differs between components. The new function uses the component module in Hinch to create and initialize a generic component structure. Then it initializes the static data and the I/O port properties. It also sets the various other functions in the newly created component structure to their respective implementations.

- clone function. This function creates a copy of the component. It is mainly used for exploiting data parallelism, as will be described in Section 3.3.3. When a grouping component clones itself, it recursively clones its children and adds them to its clone.
3.3. IMPLEMENTATION

- **destroy** function. This function destroys the component and releases all resources, including allocated memory, that are occupied by the component. When a grouping component is destroyed, it recursively destroys its children.

- **configure** function. This function (re)configures the component. Each configuration request includes an ignore-flag, a type and some type-specific values. It resembles the Linux IOCTL system call as it is a simple and generic interface for a component-specific function[3]. For example, a picture-in-picture blending component accepts a 'move' type which includes new coordinates for the picture-in-picture. A grouping component forwards all configuration requests to its children. This way, entire subtrees can easily be reconfigured at once.

A component does not need to support all configuration types. It may even omit the configure function, which means none are supported. When the ignore-flag is set, the components that support the configuration type reconfigure itself while the other components ignore it. It is an error if the ignore-flag is clear and the component does not support the given configuration type. This happens, for example, when an application developer uses a component that does not support data parallelism within a data parallel environment.

- **run** function. This function runs an iteration of the component, which typically consists of three simple stages, as shown in Figure 3.7. First, the component prepares itself for its main computation stage by acquiring input data buffers from its input streams and output data buffers from its output streams. A component also uses its static data in this phase. Second, it performs the main computations. Third, it releases the input and output buffers it had acquired in the first stage.

The first stage is also called the *synchronous stage*, as only one invocation can be active in this stage. The second and third stage do not have this limit. Section 3.3.3 gives more details about concurrent invocations.

Most components do not need any resources beyond CPU cycles and memory access, which are implicitly assigned to the component as its run function is called. These components always run to completion and Hinch guarantees that the application will never deadlock. However, when multiple components need external resources, deadlocks and livelocks could occur because of resource contention. The component developer has to prevent deadlocks and livelocks in this case.
• add, delete, replace and getChild functions (grouping components only).

These functions manage the children in a grouping component. Each grouping component has a number of slots that hold child components. All functions use a specific slot, and add, remove, replace, or return the child at that specific slot, respectively. The grouping component has to update the internal data flow graph when a child is added or removed.

The replace function has been added for optimization purposes. When a component is replaced, this can often be done efficiently because the data flow dependencies within the group do not have to be altered. The grouping component can transfer them from the old to the new component. The alternative is first removing the old component, and then adding the new component. In this case, the grouping component first removes all dependencies, after which it creates them again. By having a separate replace function, the grouping component developer can choose to build an advanced replace function, which preserves dependencies. The alternative is using a default implementation of this function provided by Hinch, which uses the add and delete functions.

3.3.2 Data flow graph

Each application contains a data flow graph, which specifies the dependencies between the components in the application. The data flow graph contains jobs and dependencies. Jobs are the actors in the data flow network and dependencies specify token channels between these jobs. A job runs when a token is available at each of its input token channels, which is called a firing of the actor in data flow terminology. When a job runs it inserts tokens at its output token channels. Hinch forwards these tokens to the input token channels of the jobs that depend on the job.

Each component has an associated job, which executes one iteration of the component by invoking the component’s run function. A component also has dependencies, that specify which jobs receive a token when the component has finished an iteration. The grouping component that contains the component manages these dependencies.

A grouping component has two sets of jobs and dependencies. The head job and the tail job are the entry and exit point of the grouping component, respectively. The head job uses the head dependencies in the group to send tokens to the jobs of the children in the group, and/or to the tail job of the group. The tail job sends tokens to the successors of the grouping component using the tail dependencies of the group.

Figure 3.8 shows an example of a grouping component and its jobs and dependencies. From the head job, the grouping component has created two head dependencies to the jobs of children A and B. From A and B, it has created dependencies to the jobs in C and D, respectively, using the standard dependencies and jobs in each component. When A has finished, Hinch will schedule C. The same holds for B and D, respectively. Finally, C and D have a dependency to the tail job of the grouping component, which causes the tail job to be scheduled when both C and D have
finished. The tail dependencies are attached to the tail job, however, the grouping component shown in Figure 3.8 does not manage these dependencies. A higher-level grouping component uses the tail dependencies to connect the grouping component to other (grouping) components.

Grouping components and non-grouping components have the same data flow interface towards the run time system and towards other grouping components, as the head job and the tail dependencies of a grouping component correspond to the job and dependencies of a non-grouping component. The head job corresponds to the job of a non-grouping component, as Hinch schedules these jobs when the component can be run. The tail dependencies correspond to the dependencies of a non-grouping component, as both indicate the successors of the component in the data flow graph. The difference lies in what happens inside the component: A non-grouping component performs some computation, and directly sends tokens to its successors. A grouping component runs its children, and sends tokens to its successors when its children have finished.

Having a single data flow interface for both grouping and non-grouping components has the advantage that the implementation of the run time system and grouping components becomes simpler. Moreover, with this interface, grouping components can be arbitrarily nested.

Dependencies are stored in a dependency list which has functions for adding and removing dependencies, amongst others. The grouping component developer can easily manage dependencies using these functions and implement reconfigurable groups. A job, in turn, uses these dependencies for sending tokens to other jobs. The management and the usage of dependencies is typically done by different components: A grouping component manages dependencies, whereas the managed component uses them. The main exception to this rule are the head dependencies in a grouping component, which are both managed and used by the same grouping component.

Running multiple iterations

The loop grouping component, which Figure 3.9 shows, runs multiple successive iterations of a data flow graph. It contains a single child with a head dependency to it. The loop grouping component also creates a loop dependency from its child to its child. Hinch will therefore schedule the child when it has finished.

The loop grouping component can also run a fixed number of iterations instead
of scheduling new iterations indefinitely. When it is initialized with an iteration count and the specified number of iterations have started, the loop grouping component removes the loop dependency and Hinch therefore does not schedule the child anymore. We mainly use this functionality for performing benchmarks. When it is initialized to run indefinitely, a loop can only stop using a different mechanism. For example, the manager grouping component can stop a loop, which we will explain in Section 3.3.6.

At the beginning of the application, Hinch adds the job that runs the top level grouping component to the job queue, as if it had received all its tokens. Typically, this job belongs to a loop grouping component which starts a new iteration when one has finished. Hinch runs multiple concurrent iterations of the application by adding this job multiple times to the job queue. When the job queue is empty and all threads are idle, Hinch will detect this condition and terminate the application.

3.3.3 Parallelism

Hinch exploits parallelism by running multiple components, or multiple iterations of a single component, in parallel. The data flow graph specifies which components can run in parallel. Hinch automatically runs components in parallel when they are on different paths in the data flow graph, if the necessary resources are available. The task parallel grouping component supports task parallelism, as shown in Figure 3.3b. It creates dependencies from its head job to each child and from each child to its tail job.

The data parallel grouping component supports data parallelism. The application developer initializes it with the desired number of parallel components and the data distribution type. Often, multiple types are available. For example, two-dimensional data allows both a row-based and a column-based distribution. Data parallelism can be nested by using a different distribution type at each nesting level. For example, a column-based distribution could be used within a row-based distribution to process image blocks in parallel.

An application developer assigns a single child component to a data parallel grouping component. The grouping component replicates this child \( n - 1 \) times using clone function of the child component, where \( n \) is the level of parallelism. Using the configure function of the child component, the grouping component configures the resulting \( n \) copies of the child for processing part of the input data. When the child component is also a grouping component, the child grouping component recursively configures its children for data parallelism. Data parallelism is only possible if the child has a clone function and supports the given parallelism type(s). If the child is a grouping component, these requirements apply to all components at all grouping levels within the group. The dependencies within a data parallel grouping component are the same as in the task parallel grouping component.

Other grouping components that implement more advanced parallel schemes can easily be added to Hinch. For example, the cross-dependencies grouping component implements the dependency structure of Figure 3.5b. Like the data parallel grouping
3.3. IMPLEMENTATION

component, the application developer initializes it with the desired level of parallelism and the parallelism type. It supports multiple children, which are replicated and configured in the same manner as in the data parallel grouping component. It replicates all children an equal number of times. The cross-dependencies grouping component creates dependencies from its head job to all copies of the first child. From the copies of the last child, it creates dependencies to the tail job. Between two replicated children, it creates dependencies from each child replica to replicas of the next child at the previous, equal, and next level.

Multiple iterations of a component can run in parallel when the component supports it. By default, components are non-reentrant, which means that Hinch will never call a function of a component when the component is already executing any function. However, a component can allow parallel execution of multiple iterations by making its \texttt{run} function reentrant. In this function, the component has to call a special re-entrance function in Hinch, which ends the synchronous stage, as shown in Figure 3.7. This approach has the advantage that the component developer can always write components using sequential code, without knowledge about parallel programming. Building parallel components is also easy, as mutual exclusion of the synchronous stage of the \texttt{run} function is handled by Hinch. During this stage, the component can freely access its static data. To avoid race conditions, it has to acquire input and output buffers from its streams in this stage. Furthermore, it has to copy all data that is used in the remaining stages into local variables in the synchronization stage.

Besides the dependencies we described in Section 3.3.2, a component also has \textit{asynchronous dependencies}, which specify that tokens are sent to another job when the component has finished its synchronous stage. For non-reentrant components, the asynchronous dependencies are used when the component finishes its \texttt{run} function. For reentrant components, these tokens are sent when the component calls the re-entrance function.

Using asynchronous dependencies, Hinch supports special grouping components. For example, the alternator grouping component uses a round-robin scheme for executing a different child in each iteration. Without asynchronous dependencies, running multiple concurrent iterations of this component is not possible, as race conditions could occur when two components in different iterations access the same resources, for example, the same communication stream(s). The alternator grouping component avoids these race conditions by creating asynchronous dependencies between children in successive iterations that access the same resources. This way, a component only runs when all components that access the same resources in the previous iteration of the alternator group have finished their synchronous stage.

\textbf{Job queue}

Hinch uses a job queue for running the application using multiple threads on a parallel architecture. The job queue stores all jobs that are ready to run. Jobs are not bound to a specific thread. All threads are equal and support all job types.
The threads in the application continuously remove jobs from the job queue and run them. As we explained above, a ‘source’ job sends tokens to other ‘target’ jobs while the source job is running. When all tokens of a target job are available, the thread that runs the source job puts the target job onto the job queue. Another thread can then remove the target job from the job queue and run it concurrently with the source job.

Although using a job queue combined with a data flow system incurs some overhead, it provides many advantages over a hand-written application that statically assigns resources. First, a job queue dynamically balances the application load over the available threads. As long as jobs are available, all threads will be busy and load balance is optimal. Second, reconfigurable applications are easily supported, as reconfiguration does not require changing the scheduling algorithm. With statically scheduled reconfigurable applications, a separate schedule has to be designed for each configuration. Third, the number of threads can easily be adjusted to the target hardware platform, as there is no static assignment of computations to a specific thread. The job queue thus causes the application to be very flexible regarding the target hardware platform. It even allows malleable applications, which adapt to the availability of resources.

For reducing overhead, the job queue consists of two parts. The main queue is a central queue in shared memory, which all threads can access. Besides the central job queue, each thread has a private queue which can contain one job. Using the private queue has an advantage over using the central queue: Accessing the private queue does not require synchronization with the other threads. The central job queue does require synchronization, as all threads access it concurrently.

The private queue has priority over the central queue when the thread adds or gets a job from the queue. As a job adds new jobs to the job queue, the first job goes to the private queue. The following jobs go to the central queue, which allows other threads to execute these jobs concurrently. When the current job finishes, the thread immediately continues executing the job from its private queue. This scheme thus avoids synchronization with the central queue for any job that adds one or more jobs to the job queue.

Using a private queue for each thread may affect performance, because jobs in a private queue do not run until the associated thread finishes its current job, which limits parallelism. If the private job would be put in the central queue, another thread could execute it concurrently. However, as long as the application generates enough jobs, this limit does not apply. Moreover, a job typically adds jobs to the queue after it has finished its main computations, since the added jobs typically have a data flow dependency on the current job. Executing the first added job by another thread therefore does not result in parallel execution of the core computations of the application.

We have limited the size of the private queue to one to allow executing the jobs beyond the first added job concurrently with the first added job. The original thread executes the first added job, which it receives from its private queue. The other threads execute the other jobs, which they receive from the central queue.
3.3. IMPLEMENTATION

3.3.4 Communication streams

As mentioned in Section 3.3.1, each component has a number of I/O ports, which are connected to streams. A stream acts as a FIFO channel with fixed-size data buffers between two or more components. Future implementations could also support variable-sized data buffers. Streams are implemented in Hinch using a generic stream module, which mainly performs buffer management. All components have a fixed number of I/O ports, which the component developer sets upon initialization of the component. Streams are fully reconfigurable, as the stream module supports connecting and disconnecting components at run time. The stream module provides the following functions:

- Create a new stream. This parameterless function allocates memory for a new stream structure, and initializes this structure with default values.

- Set I/O properties. This function sets the type of the stream, which determines the buffer size, amongst others. If this function is not called, the type of the stream is set when the first component is connected.

- Connect a component. Each stream maintains a list of components that access the stream. When connecting a component, Hinch passes the I/O properties of the respective component port to the stream module, which compares them to the I/O properties of the stream and verifies if the port and the stream are compatible. The stream module also determine the access type of the component using the port’s I/O properties. A component reads and/or writes the data in a stream.

- Disconnect a component. For reconfigurability purposes, components can be disconnected from streams.

- Synchronize a component. Before a component accesses a stream, it should be synchronized to the other components that access the stream. Synchronization ensures that the component receives the same data as the other components at the next access to the stream. The application developer may only call this function when no components are accessing the stream, because otherwise different components may be accessing different data and synchronization is impossible.

Connecting and synchronizing components are two separate actions for reconfigurability purposes. Separating these functions allows connecting new components while the application is running and other components are still accessing the stream. Synchronization occurs when the application is idle for reconfiguration.

- Destroy a stream. This function frees any memory that was allocated for the stream. It is an error if this function is called while a component is still accessing the stream.
• Start reading/writing. When a component wants to access a new data buffer from the stream, it calls this function. A component typically calls this function at the beginning of its run function, as shown in Figure 3.7.

The start reading/writing function returns a buffer identifier, which the component uses to access the actual data. Before a component can read a buffer, one of the writing components must have finished writing it. The data flow graph should ensure that writing components are scheduled before reading components.

• End reading/writing. The component has to call this function when it has finished accessing a data buffer. It should supply the buffer identifier it received when it started reading or writing, to allow the streaming module to distinguish between multiple concurrent accesses by the same component. When all components have finished accessing a buffer, the streaming module will free the buffer or reuse it in a later iteration. This function is typically called at the end of a component’s run function, as shown in Figure 3.7.

The implementation of streams uses an efficient zero-copy protocol. A component writes to a stream after acquiring an output data buffer from the stream. When it is finished writing, it commits the buffer. A component that reads this stream then reads the data from the same buffer. The zero-copy protocol is only possible at shared memory machines. Distributed memory machines, require a different implementation, which is subject to future research.

By default, the stream module allocates memory space for the data buffers that are used in the stream. To reduce memory copying, a writing component can also choose doing this allocation by itself and simply store a data reference in the stream. This functionality is useful when the data that is written already resides in memory. In our applications, it is used for components that read a file from disk and write the contents to a stream. To avoid I/O overhead at run time, the file is read from disk at initialization time. At run time, the writing component stores a data reference in the stream, instead of copying the data. At the end of the input file, the component loops to the beginning of the file, which simulates an infinite input stream.

3.3.5 Event communication

Besides using communication streams, Hinch supports communication between components using events. An event is a short message that consists of a type identifier and a type-specific value. For example, when the type is ‘key’ the value holds the key code. An event module manages events and stores them in an event queue. Multiple event queues can be present in an application, which allows independent communication between different sets of components.

When components need to communicate using events, the application developer has to create the event queue before creating the components. After creating the event queue, the application developer initializes the components with a reference
3.3. IMPLEMENTATION

Normal

normal operation

reconfiguration

required

idle

Stopping

not idle

end

program

Halted

Processing

reconfiguration complete

Figure 3.10: Manager state diagram

to the queue. In future implementations, the configure function of the component may be used to dynamically assign event queues to components.

The event queue module provides functions for creating and destroying event queues, putting a new event at the tail of the queue and getting the event at the head of the queue. The event queue thus has FIFO semantics. The event module avoids race conditions by serializing all accesses to an event queue.

The get function of the event queue module is used to poll the queue for new events and get the oldest event from the queue. When the queue is empty, the get function returns an event with the special type 'none'. Otherwise, it removes the oldest event from the queue and returns it. To drain the event queue, the get function should be called until it returns an event with type 'none'.

3.3.6 Dynamic reconfigurability

As mentioned in Section 3.2.4, a manager grouping component manages a reconfigurable subgraph. This grouping component is part of a standard set of grouping components that comes with the Hinch run time system. The manager receives events and calls subgraph-specific reconfiguration functions when reconfiguration is required. All structures that might be modified support reconfiguration, as we have shown previously in this section. These structures include both the internal Hinch structures and the additional (grouping) components used by the application.

The manager grouping component has a single child, which represents the contained subgraph. This child typically is another grouping component. A manager avoids race conditions by only performing reconfiguration when its child is idle. Upon initialization, the application developer supplies a manager with an event queue. The manager listens for reconfiguration commands from this queue.

A manager’s main task is making its child idle when it needs reconfiguration. It maintains a counter of the number of active iterations in its child. Whenever the manager schedules a new iteration in its head job, it increments the counter. When an iteration is complete, the manager’s tail job decrements the counter.

The manager can be in four different states, which determine its actions. Figure 3.10 shows the state transition diagram for the manager. By default, a manager is in the 'normal' state. In this state, it schedules an iteration of its subgraph when the manager itself is scheduled. It polls the event queue for new events in every iteration. When it is in the 'stopping' state, its subgraph needs reconfiguration but the subgraph is not idle yet. The manager therefore suspends new subgraph iter-
tions, by telling Hinch that it is not ready to be run when Hinch schedules it. Hinch then puts the manager’s head job at the end of the job queue, and automatically schedules the manager later.

When a manager is in the 'processing' state, it is processing events. A manager avoids race conditions by serializing all event processing. Because most events will trigger a reconfiguration, the manager suspends new iterations similarly to the 'stopping' state. When a manager has processed all events, it returns to the 'normal' state and Hinch automatically restarts suspended iterations when their jobs are run from the job queue.

When a user requests that the application should stop, the component that received this request sends a special 'halt' event to the manager. After processing this event, the manager enters the 'halted' state instead of the 'normal' state after the 'processing' state. In the 'halted' state, the manager’s head job reports successful execution to Hinch, however, it does not sends tokens to the job of its child anymore. This way a manager consumes all tokens in its part of the data flow graph. When the manager is part of the application loop, the application will eventually go idle as the loop is broken. Hinch will detect this situation, because the job queue will become empty, and terminate the application.

The manager grouping component is generic and therefore does not have knowledge about the structure of its child, or about which events trigger reconfiguration. The application developer therefore has to supply two functions to a manager that are specific to the application. These functions perform the actual reconfiguration. A manager avoids race conditions by serializing all calls to these functions. To relieve the application developer from writing these functions, they can be automatically generated from an XML specification of the application, as we will show in Section 4.3.1.

The manager calls the detect function when it gets a new event from the event queue. This function decides whether reconfiguration is required. It tells the manager whether it should keep running (stay in the 'normal' state), whether it should make the child idle (go to the 'stopping' state), or whether the manager should be halted (go to the 'halted' state via the 'stopping' and 'processing' states).

The manager calls the idle function when the detect function indicated that it had to make its child idle, and the manager has entered the 'processing' state. This function then performs the actual reconfiguration, for example by adding components to the child.

To reduce the latency of the reconfiguration by the idle function, the detect function can perform the part of the reconfiguration that does not require the child to be idle. For example it can create and initialize the components and streams that have to be added to the subgraph. The idle function can then skip these steps and only needs to add these structures to the child. This optimization also allows exploiting more parallelism as it shortens the sequential parts of the application, in which the child is idle.
3.4 Experiments

For evaluating Hinch, we use a synthetic benchmark application and two simple real-life applications. Since run time behavior is most important in streaming applications, all measurements do not include initialization, which mainly consists of allocating static resources and building the initial component tree.

We performed measurements on DAS-3, Cell blade, and Niagara platforms, which we described in Section 1.2. On DAS-3, we repeated each measurement eleven times and took the average of these measurements. Since a Cell blade is considerably slower than a DAS-3 node, gathering Cell blade measurements took more time and effort. On the Niagara gathering measurements took even more time and effort, since we only had access to a single Niagara server. Moreover, since the Niagara supports more threads than the other architectures, we performed more measurements on the Niagara. On the Cell blade and the Niagara, we therefore repeated each measurement five times instead of eleven times. On the DAS-3 and Cell blade clusters, these repeated measurements typically ran on different nodes. Unless specified otherwise, the standard deviation for a set of repeated measurements is always less than 6% of the average measurement.

3.4.1 Applications

We use three applications for evaluating Hinch: Benchmark, Filter, and PiP. Benchmark is a synthetic benchmark application that performs a fixed amount of computation in every iteration. Filter applies a convolution filter to a series of input images. PiP is a picture-in-picture application that blends reduced size video streams into a background video stream.

For all applications, we have created a hand-written version, that does not use Hinch, besides the Hinch version. In Section 3.4.2 we will evaluate the overhead of Hinch by comparing the hand-written version to the Hinch version.

Filter and PiP use files with uncompressed 720x576 YUV images with 4:2:2 color subsampling as input. Since we are interested in the performance of the computational parts of the application, and since I/O overhead may hide inefficiencies in Hinch, the application avoids I/O overhead during the measurement by reading all input files into memory at initialization time. Our measurements of execution time do not include this initialization time. At run time, the application thus reads the input data from memory. When it reaches the end of an input file, it loops to the beginning of the file, simulating an infinite input stream.

In the Hinch versions of the applications, an input component handles file input. It has an output stream for the luminance data (Y) and an output stream for the packed chrominance data (UV). In each iteration, it writes one image frame to both output streams.

The output components have Y and UV input streams from which they read the data that they sent to the output device. When debugging or giving demonstrations, the applications use output components that write the output to a file or display it
on the screen, respectively. When performing measurements, the applications avoid I/O overhead by using output components that discard the output.

Besides input and output components, the applications have components that transform the input. We will now explain the applications and their components in more detail.

**Synthetic Benchmark**

The synthetic benchmark application is based on a synthetic benchmark component that simulates a real component. In each iteration, it reads one data buffer from an input stream, executes a fixed amount of computation and writes another data buffer to an output stream. It accesses all data in these buffers.

We have created four versions of this application, as shown in Table 3.5. The hand-written version does not use Hinch and does not exploit parallelism. The pipeline, slicing, and mixed versions differ in the way they exploit parallelism. The pipeline version splits the computation into equal parts, with one synthetic benchmark component for each part. It exploits pipeline parallelism by connecting these components in a pipeline. The slicing version exploits data parallelism by splitting the input into slices, as explained in Section 3.2.2. The number of slices is equal to the number of pipeline stages in the pipeline version. The mixed version exploits both pipeline parallelism and data parallelism, by nesting a data-parallel computation inside each pipeline stage.

On DAS-3 and the Cell blade, the level of parallelism is 16 in the application. Since the Niagara supports 64 threads, the level of parallelism is 64 on this architecture. In the pipe and sliced versions, the level of parallelism corresponds to the number of pipeline stages or slices, respectively. In the mixed version, the number of pipeline stages and slices are equal to the square root of the level of parallelism, which results in an equal amount of parallelism for all versions.

By default, Hinch schedules four concurrent iterations. When the number of pipeline stages is larger than four, Hinch can not exploit all possible pipeline parallelism. In this case, we therefore set the number of concurrent iterations to the number of pipeline stages.

<table>
<thead>
<tr>
<th>Version</th>
<th>DAS-3 / Cell blade</th>
<th>Niagara</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pipeline stages</td>
<td>Slices</td>
</tr>
<tr>
<td>Hand-written</td>
<td>N/A (sequential)</td>
<td>64</td>
</tr>
<tr>
<td>Pipeline</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>Slicing</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>Mixed</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 3.5:** Synthetic Benchmark versions
3.4. EXPERIMENTS

Filter

The Filter application is based on a component that applies a convolution kernel to a stream of input images, creating a stream of output images. This application uses a zero-derivative Gaussian kernel, which blurs the image. The convolution filter component supports data parallelism using slicing. The application supports three modes of operation:

1. Filter-1D. In one dimensional mode, the application uses two convolution filter components. Figure 3.11 shows the components and communication streams in this application. The first component applies a horizontal kernel of size 3. The second component applies a vertical kernel, which is identical to the horizontal kernel except for its orientation. A cross dependencies grouping component combines the convolution filter components, as explained in Section 3.3.3

2. Filter-2D. In two dimensional mode, the application uses a single convolution filter component which applies a two-dimensional kernel. The kernel matrix is the outer product of the vectors used in the two one-dimensional kernels, which results in a 3x3 matrix. The result of applying the two-dimensional kernel is therefore similar to applying the two one-dimensional kernels.

3. Filter-reconf. The application switches between 1-D and 2-D modes at run time. After executing 32 iterations in one mode, it switches to the other mode. The application uses a manager grouping component for halting the application for reconfiguration, as explained in Section 3.3.6. The application has an extra component that sends a reconfiguration event to the manager every 32 iterations.

Picture-in-Picture

The picture-in-picture (PiP) application combines multiple input video streams into a single output stream. It displays the main input stream unmodified at the background. The other input streams contain picture-in-picture streams. The application
scales the images in these streams down and blends them into the background image, resulting in a picture-in-picture display. Section 6.1.1 will explain the component structure of this application in detail. For our experiments, we run PiP in three modes:

1. PiP-1. The application has one picture-in-picture. All components related to processing the second picture-in-picture are disabled in this mode.


3. PiP-reconf. The application switches between one and two pictures-in-picture at run time, by enabling or disabling the components for the second picture-in-picture. After executing 32 iterations in one mode, it switches to the other mode. The application uses a manager grouping component for halting the application for reconfiguration, as explained in Section 3.3.6. The application has an extra component that sends a reconfiguration event to the manager every 32 iterations.

3.4.2 Hinch overhead

We analyze the efficiency of Hinch by comparing hand-written versions of the applications, which do not use Hinch, to the respective Hinch versions. Both versions perform the same computations. The hand-written versions are sequential applications that perform all computations in one go. In the Hinch versions, the application spreads the computations over multiple parallel components. The Hinch versions therefore have overhead in scheduling components and communication between components. This overhead is absent in the hand-written versions.

Besides the hand-written version and the parallel Hinch version, we also compare a sequential Hinch version. The sequential Hinch version differs in one aspect from the parallel Hinch version: All synchronization primitives for locking and unlocking mutexes are omitted from the sequential Hinch version. The performance difference between the sequential Hinch version and the parallel Hinch version thus gives insight in the locking overhead of Hinch.

Figure 3.12 shows the overhead of Hinch for the applications on DAS-3, Cell blade and Niagara architectures. Since the hand-written and sequential Hinch versions do not exploit parallelism, we compare the Hinch overhead using a single thread. The overhead percentages are relative to the performance of the hand-written version. For example, an overhead of 10% means the Hinch version is 10% slower than the hand-written version.

On all architectures, the PiP applications have the highest overhead. The components within these applications perform few computations compared to the other applications. The overhead of Hinch is therefore relatively high in the PiP applications and reaches 24% on DAS-3.

The difference between the sequential and parallel Hinch versions is typically very small, which indicates low locking overhead. There are a few exceptions: On the
3.4. EXPERIMENTS

Figure 3.12: Hinch overhead for sequential applications
Niagara, the parallel Hinch version performs significantly worse than the sequential Hinch version with the PiP application, which already has a relatively high overhead on this architecture. On the other hand, the parallel Hinch versions of the Filter and PiP applications perform better than the sequential Hinch versions on the Cell blade. The parallel Filter applications even perform better than the hand-written versions. Adding the locking code thus has a positive impact on performance in these cases. The changed memory layout and explicit synchronization statements apparently have a positive impact on cache performance, which outweighs the cost of the extra locking code.

On the Niagara, the Hinch versions of the Benchmark and Filter applications are slightly faster than the hand-written versions. This behavior is also best explained by better cache performance.

On all architectures, the Pipeline version of the Benchmark application performs best. In the Pipeline version, each job adds one job to the job queue, which corresponds to the next stage of the pipeline. The application therefore benefits from the low-overhead private job queues, which we explained in Section 3.3.3. In the Slicing version, the application adds the jobs for all slices at once to the job queue, which means it uses the central job queue for most jobs, which is less efficient than using the private queues. Moreover, the Slicing version has more coarse-grain synchronization compared to the Pipeline version. In the Slicing version, the application waits for completion of all slices before starting a new iteration with new computations. In the Pipeline version, the next pipeline stage with the new computations only has to wait for completion of the current pipeline stage. As would be expected, the performance of the Mixed version lies between the performance of the Pipeline and Slicing versions.

### 3.4.3 Parallel performance

On all architectures, we analyze the performance of the parallel Hinch version and measure the speedup relative to the fastest sequential version, which is either the hand-written version or the parallel Hinch version. We do not compare against the sequential Hinch version because it is not intended for production use: We only use this version for debugging purposes and for measuring locking overhead, as mentioned in the previous section.

Figure 3.13 shows the speedup of Hinch for the applications on DAS-3, Cell blade

<table>
<thead>
<tr>
<th>Arch.</th>
<th>Application</th>
<th>Threads</th>
<th>Std. dev.</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAS-3</td>
<td>PiP-1</td>
<td>4</td>
<td>9,4 %</td>
<td>Load imbalance</td>
</tr>
<tr>
<td>Niagara</td>
<td>Benchmark-Slicing</td>
<td>24</td>
<td>12,1 %</td>
<td>Varying idle time</td>
</tr>
<tr>
<td>Niagara</td>
<td>Benchmark-Slicing</td>
<td>32</td>
<td>8,6 %</td>
<td>Varying idle time</td>
</tr>
<tr>
<td>Niagara</td>
<td>Benchmark-Slicing</td>
<td>64</td>
<td>26,0 %</td>
<td>Varying idle time</td>
</tr>
</tbody>
</table>

Table 3.6: Parallel measurements with high standard deviation
3.4. EXPERIMENTS

Figure 3.13: Hinch parallel performance

and Niagara architectures. The applications use the maximum number of threads supported by the architecture, which is 4 for DAS-3 and the Cell blade, and 64 for the Niagara.

In some cases, the standard deviation between repeated executions of a measurement is relatively high. Table 3.6 lists all cases in which the standard deviation exceeds 6% of the average measurement. PiP-1 on DAS-3 suffers from load imbalance, which varies between the eleven repeated executions of the measurement and thereby causes the high standard deviation. In the Benchmark-Slicing experiments on the Niagara with 24 or more threads, the threads are often idle, waiting for jobs. Although the total time spent in calculations remains equal, the total idle time varies considerably, which causes the high standard deviations.

On DAS-3, both Filter applications achieve near-perfect speedups. The Benchmark applications also perform well, while the PiP applications perform relatively bad. These results correspond to the overhead statistics of Figure 3.12a. When the overhead is high, the application spends much time in Hinch, which internally
synchronizes the various threads using mutual exclusion primitives. The relatively high amount of synchronizations causes a low speedup, as shown in Figure 3.13.a.

The Cell blade has two general-purpose cores (PPEs), which support two hardware threads each. We do not use the SPE coprocessor cores for these measurements. On the Cell blade, all applications obtain similar speedups. With two threads, the applications use two dedicated CPU cores and speedup is nearly optimal. With three and four threads, multiple hardware threads share one CPU core, and the performance increase is therefore limited.

On the Niagara, a similar effect occurs. The Niagara has eight cores which support eight hardware threads each. The PiP applications, which have relatively high overhead, as shown in Figure 3.12.c achieve speedups of only 9.5 using 64 threads, because a high overhead means Hinch performs relatively more synchronizations. The Filter applications achieve decent speedups of 31 using 64 threads. Benchmark-Pipeline achieves an even better speedup of 39 using 64 threads.

These numbers show that Hinch can efficiently exploit parallelism: Even with simple applications, with relatively fine-grained parallelism, and architectures in which multiple threads share a single CPU core, the parallel performance of these applications is high.

3.4.4 Reconfigurability overhead

In this section we evaluate the overhead of dynamic reconfigurability by comparing reconfigurable applications against corresponding static versions. We compare the performance of Filter-reconf to the average performance of Filter-1D and Filter-2D, since Filter-reconf processes half of its input in 1D mode and the other half in 2D mode. Similarly, we compare the performance of PiP-reconf to the average performance of PiP-1 and PiP-2.

When the application pauses for reconfiguration, the amount of parallelism in the application drops until the application runs sequentially. Reconfigurable applications can thus exploit less parallelism than their non-reconfigurable equivalents. This effect increases with the number of parallel nodes. We therefore expect that the reconfigurable applications perform relatively worse, especially on larger numbers of nodes.

Figure 3.14 shows the relative overhead of Filter-reconf and PiP-reconf on all architectures. On DAS-3 and the Cell blade, we compare the performance using 1 and 4 threads. On the Niagara, we also include measurements using 16 and 64 threads. We use the parallel Hinch versions in all cases, as this version is intended for practical use, and the sequential Hinch version is not.

On DAS-3 and the Cell blade, dynamic reconfigurability hardly incurs any overhead. The reconfigurable applications sometimes perform better than the average of the non-reconfigurable versions, however, the differences are typically not significant.

There is one exception to this rule. On DAS-3, PiP-reconf does perform significantly better than the average of PiP-1 and PiP-2 when using 4 threads. PiP-reconf even performs better than PiP-1 in this case. Detailed statistics show that PiP-1 and
3.4. EXPERIMENTS

PiP-2 suffer from load imbalance, while PiP-reconf does not. Regularly pausing the application for reconfiguration apparently makes the application more deterministic. Thereby it removes load imbalance and consequently performance improves.

On the Niagara, dynamic reconfiguration comes at a price. PiP-reconf has an overhead of up to 4.9%. Filter-reconf has an overhead of 30% with 64 threads. Detailed performance statistics of Filter-reconf show that the 64 threads are idle 34% of the time, while they are idle for less than 1% of the time in Filter-1D and Filter-2D. Filter-1D and Filter-2D therefore achieve good speedups, as shown in Figure 3.13c. In Filter-reconf the threads go idle when the application pauses for reconfiguration, which happens every 32 iterations. Contrary to Filter-1D and Filter-2D, Filter-reconf therefore can not continuously exploit all possible parallelism, which results in relatively poor performance compared to Filter-1D and Filter-2D. In PiP-reconf this overhead is not visible, since PiP-1 and PiP-2 do not exploit much parallelism, as shown in Figure 3.13c.

Figure 3.14 shows the average reconfiguration frequency in terms of reconfigurations per second, which we compute by dividing the number of reconfigurations by the execution time of the application. Since the performance of the application in terms of iterations per second varies across architectures and applications, and the applications reconfigure every 32 iterations, the number of reconfigurations per second also varies. PiP-reconf, which has less compute intensity than Filter-reconf, executes more iterations per second and therefore performs more reconfigurations per second. The reconfiguration frequency for PiP-reconf varies between 0.95 and 17.9 reconfigurations per second. For Filter-reconf, the frequency varies between 0.13 and 3.2 reconfigurations per second.

We conclude that the overhead of dynamic reconfiguration is low. Even when the applications reconfigure themselves multiple times per second, the overhead typically stays below 7%. If an application exploits a high degree of parallelism, recon-
configuration clearly incurs overhead, as the application executes sequentially during reconfiguration. However, we have shown worst-case behavior, as most applications do not reconfigure as often as every 32 iterations. When reconfiguration occurs less often, the corresponding overhead also drops.

3.5 Summary

This chapter describes Hinch, a run time system for parallel streaming applications. We first derived the requirements from Hinch from the perspective of the streaming application, the target architecture and the developer. We categorized these requirements into six categories. Hinch should support components with data-flow dependencies between them. Hinch should exploit parallelism in the architecture by supporting multiple forms of parallelism in the application. Furthermore, it should support streaming and event communication between components. It should also support running multiple iterations of the data-flow graph. A challenging requirement is supporting dynamically reconfigurable applications, which impacts the whole run time system. Finally, towards the component and application developer, Hinch should be easy to use by supplying simple interfaces, amongst others.

Supporting these requirements individually is difficult. Integrating support for all requirements together is even more challenging. We have addressed this task using a modular design based on data flow process networks. Components are organized into a data flow graph that is managed by grouping components. A loop grouping component runs multiple iterations of the data flow graph. A stream and event module provide abstract communication primitives to the components.

Hinch exploits multiple forms of parallelism using a grouping component for each kind of parallelism. The range of supported types of parallelism can easily be increased by adding new grouping components. Hinch achieves load balancing using a central job queue, which dispatches work to idle threads.
The modular design ensures a separation of concerns and thereby gives Hinch a clear interface towards the developers. One important feature is that the component developer only needs to write sequential code. Hinch automatically exploits parallelism between different components. The interface contains some optional features, for advanced developers. Using the basic interface, a developer can quickly build and test a fully working application. Using the advanced features, a developer can optimize performance, for example by adding parallelism to the application.

Hinch supports dynamic reconfigurability throughout the whole run time system by allowing structures to be modified at run time. Reconfiguration can be performed both at the component and the application level. A component can be reconfigured using a configuration interface, which should be implemented by all reconfigurable components. At the application level, entire subgraphs can be changed by adding and removing components and communication streams, amongst others. The manager grouping component ensures that reconfiguration is only performed when a subgraph is idle.

A performance analysis using three different shared-memory architectures shows that Hinch typically has low overhead. When the application has many fine-grained components, which do few computations, the overhead of managing these components naturally becomes relatively high. Hinch can efficiently exploit parallelism, even when the parallel components are relatively fine-grained. The overhead of dynamically reconfiguring applications at run time is only high when the application exploits a great degree of parallelism, because the application runs sequentially during reconfiguration. However, as reconfiguration typically does not occur often, this overhead is limited.
Chapter 4

XSPCL: A coordination language for streaming applications

When using the Hinch API, as described in Chapter 3, the application developer manually initializes components and connect them using streaming channels and/or event queues. This approach is viable for simple applications, for example for testing the functionality of a single component. However, maintaining an overview of more complex applications with multiple different components is difficult, especially if these applications are reconfigurable and the component layout is dynamic.

The XSPCL coordination language (pronounced as x-special) aids the application developer in building these complex applications, by providing an abstraction of the Hinch application development API. XSPCL is a generic intermediate language for streaming applications. Figure 4.1 shows the position of XSPCL within the SP@CE framework, which we described in Section 1.3. On top of XSPCL, SP@CE has a Front-End, in which the application developer expresses the application using a graphical interface. Below XSPCL, an XSPCL application is either compiled into an executable, or it is fed into a performance prediction tool. When using XSPCL in a streaming application development platform, the application developer is ideally not aware of the XSPCL layer.

XSPCL is based on SPC-XML, which is a specification

\footnote{The work in this chapter has been published in M. Nijhuis, H. Bos and H. E. Bal. A Component-based Coordination Language for Efficient Reconfigurable Streaming Applications. In Proc. Intl. Conf. on Parallel Processing (ICPP), September 2007.}
language for parallel programming, based on the Series-Parallel and Contention (SPC) model\textsuperscript{55, 56}. XSPCL supports task- and data parallelism, dynamically reconfigurable applications, and interactive applications that respond to asynchronous user events. As XSPCL is based on XML, it is extensible. An application developer can easily explore different parallelization strategies using XSPCL without modifying the underlying component code.

We have implemented a prototype XSPCL compiler that implements these features. In this chapter, we will use the term \emph{XSPCL application} for an executable application generated by the XSPCL compiler.

The XSPCL compiler transforms an XSPCL specification into an executable program with calls to the Hinch application development API. This program is linked to the components and to the Hinch run time system. The overhead of XSPCL is negligible because the generated code only runs at initialization time, or when the program is reconfigured. During normal operation, the components and the run time system determine the application performance.

XSPCL has been designed for including performance prediction details. In future implementations compiler analysis and/or performance prediction tools add these annotations to an XSPCL specification. The XSPCL compiler can then use these annotations and choose optimal parallelization strategies.

In this chapter, we present the XSPCL coordination language for streaming applications. First we give an overview of the required properties of XSPCL in Section 4.1. Section 4.2 describes the coordination language and how it meets its requirements. Section 4.3 describes the implementation of the XSPCL processing tool. In Section 4.4 we investigate the overhead of using XSPCL by comparing XSPCL applications to equivalent applications that directly use the Hinch run time system. Finally, we conclude the chapter in Section 4.5 and provide future work directions.

4.1 Required properties of XSPCL

Following the design of SP@CE, as described in Section 1.5, XSPCL focuses on streaming applications, because many challenging applications fall into this category. XSPCL must handle the following aspects of these applications:

1. Individual components. XSPCL must provide primitives for expressing the basic components in the application. A component has formal parameters that define its behavior. These parameters are different for each component. An application can contain multiple instances of a single component. Each instance may have different actual parameters.

2. The task graph of components. XSPCL must provide primitives for expressing sequential series of components, as well as various parallel constructs. Ideally, XSPCL supports all grouping constructs of the lower layer Hinch run time system, as mentioned in Section 3.3.3. XSPCL should also be extensible and support new task graph constructs as they become available in Hinch.
A streaming application runs multiple iterations of the task graph. XSPCL should support this behavior, including different termination conditions. The application may run a fixed number of iterations, and/or until a certain event occurs.

3. Procedural abstraction. To avoid specifying similar sub graphs multiple times, XSPCL must provide primitives for encapsulating these sub graphs into procedures. The application then instantiates these procedures at multiple positions in the application. A procedure may have parameters which configure a specific instance of the procedure. Parameters make the procedural abstraction applicable in a wide range of cases.

4. Communication between components. Communication is abstract in the sense that a component does not know to which other component(s) it is connected. It only knows the communication channel it is connected to, but not which component is on the other end. Different component instances are typically connected to different communication channels. Hinch only allows communication between compatible components. XSPCL must provide primitives for two forms of communication:

   (a) Streaming communication. Streams provide a synchronous communication primitive for possibly large pieces of data. Each component has a number of I/O ports to which streams are connected. When the component runs, it reads the data at its input ports. This data has been written to the stream by another component that has been scheduled earlier in the iteration. Similarly, the component writes data to its output ports. This data will be read by other components that will be scheduled later in the iteration.

   (b) Event communication. Events provide an asynchronous communication primitive for small pieces of data. A component can send events to other components whenever it is active, for example when it detects the user has pressed a key. Applications can also use events to respond to special input values. A component can send any number of events, including zero. Similarly, a receiving component receives zero or more events when it is active, depending on how many events other components sent to it.

5. Reconfigurability. Parts of the application may be enabled or disabled at run time, for example in response to an event. Reconfigurability yields two requirements for XSPCL:

   (a) XSPCL must provide primitives that declare optional sub graphs.

   (b) A graph with optional sub graphs must have a special container that specifies the reconfigurable part of the application. This container is needed to keep the contained sub graph in a consistent state. For example, nodes in the optional sub graph can be connected using streams to other
nodes in the contained sub graph. When these connections are made, the nodes in the optional sub graph have to synchronize with the other nodes in the contained sub graph. As we will show in Section 4.2, the container is also a convenient place to define the relation between events and the sub graphs that are be modified in response to these events.

6. New primitives. XSPCL should be extensible. It should be easy to add new primitives to XSPCL, for example parallelization hints from a performance predictor, or new parallelization strategies.

4.2 XSPCL language specification

Using the above-mentioned requirements, we have created an XML-based coordination language for streaming applications called XSPCL. As we derived XSPCL from SPC-XML[56], many XSPCL tags resemble SPC-XML tags. However, XSPCL is novel because it supports domain-specific features such as streaming communication, reconfiguration and user event handling. Unlike SPC-XML, XSPCL has some primitives that do not adhere to the Series-Parallel paradigm. Using these primitives, an application can avoid unnecessary synchronization, which improves performance.

We chose basing XSPCL on SPC-XML and hence on XML because XML processing tools are widely available, XML specifications are human-readable, and an XML-based language can easily be extended with new primitives.

XSPCL supports a number of compile-time flags that indicate that some parts of the application are enabled, and other parts are disabled. XSPCL has an architecture variable, that identifies the architecture, a debug flag, for enabling debugging, and a demonstration flag, for compiling a demonstration version. By enclosing parts of the component tree in special tags, these parts are enabled or disabled, depending on the compile-time flags. In case an application developer desires other flags, XSPCL can easily be extended with support for these new flags.

Using <for> and <if> tags, the application developer can express repeated and conditional pieces of XSPCL code, respectively. The for loop has an iteration counter, which the XSPCL code inside the loop can use. Besides the current iteration, the application developer can also use the next or previous iteration counter. This feature is useful for connecting components in different for loop expansions.

Below we will first describe the top-level structure of an XSPCL specification in Section 4.2.1. Then we will describe the structures that build the component tree in a bottom-up fashion in Sections 4.2.2 to 4.2.7.

4.2.1 Header

Listing 4.1 shows the top-level structure of an XSPCL specification. An <xspcl> tag surrounds the whole document. Its version attribute indicates the XSPCL version used. Within the <xspcl> tag there is one header, defined by <head>, and
4.2. XSPCL LANGUAGE SPECIFICATION

one or more procedures, defined by `<procedure>` tags. The `<procedure>` tags contain the component tree of the application, whose structures are described below. The `<head>` tag contains global definitions of the application. The application developer can omit most of these global definitions. The XSPCL compiler then uses default values. XSPCL contains the following header elements:

- `<description>` This tag contains a short description of the application. All applications require this tag.

- `<iterations>` This tag specifies the number of iterations the application executes. If it is `-1`, the application runs indefinitely or until it meets another termination condition. When this tag is omitted, the default value is `-1`.

- `<cai>` This tag specifies the number of concurrently active iterations in the application, which determines the level of parallelism. Section 4.2.4 explains in detail how XSPCL applications exploit parallelism. By default, the number of concurrently active iterations is four, which is the default value of the underlying Hinch run time system.

- `<threads>` This tag specifies the number of parallel threads used by the application. The Hinch run time system automatically balances the load over these threads using a job queue in shared memory, as explained in Section 3.3.3.

- `<spus>` When the application runs on the Cell architecture, this tag specifies how many SPEs the application uses. Section 5.1 describes this architecture in detail. XSPCL fully supports applications that use this advanced heterogeneous architecture.
CHAPTER 4. THE XSPCL COORDINATION LANGUAGE

- **<globals>** This tag contains definitions of global variables that are specific to the application. They are specified in the language generated by the XSPCL compiler, which is C in the current implementation. The XSPCL compiler copies these definitions into the generated source file.

- **<init>** This tag contains application-specific initialization code, which for example initializes the global variables defined in the `<globals>` tag. Like the global variables, the language for this code is C. An XSPCL application executes this code before it creates and initializes the application graph, including all components. An application developer can therefore use the global variables in the application graph.

- **<deinit>** This tag contains application-specific de-initialization code in the C language. The application developer uses it for freeing any resources that were allocated in the `<init>` tag. An XSPCL application executes this code at the end of the application.

- **<include>** This tag specifies a C header file that the generated application must include. For each component in the application, an application developer has to specify the header file that contains the prototype of the new function of the component. If the application uses external libraries in the `<init>` or `<deinit>` sections, the application developer uses an `<include>` tag for including the appropriate header files. The application can include multiple header files using multiple `<include>` tags.

The XSPCL compiler adds common command-line parsing routines to the application. The end-user can use command-line arguments for overriding the values in the `<iterations>`, `<cai>`, `<threads>`, and/or `<spus>` tags.

XSPCL supports multiple `<globals>`, `<init>`, and `<deinit>` tags in a single specification. The application developer can easily group the definition, initialization and de-initialization of a single variable together this way.

The `<globals>`, `<init>`, and `<deinit>` tags provide a simple means for including small pieces of C code in the resulting application. We have not intended them for large pieces of code. Mixing two languages in a single file is generally not a good idea as it leads to incomprehensible code and it violates the principle of separation of concerns. The application developer can include large pieces of code by putting this code in a separate source file. The application developer can then call the routines in the separate file from the XSPCL specification.

### 4.2.2 Components

A component is the most basic structure in an XSPCL specification. Components implement the basic functionality of the application. An application developer can use all components from the Hinch component library in XSPCL. Listing 4.2 gives an example of a component that implements a spatial down scaler. This component
4.2. XSPCL LANGUAGE SPECIFICATION

Listing 4.2: Example XSPCL component specification

reduces the size of each frame at its input stream by a given downscale factor, and writes the resulting pixels to its output stream.

The class attribute specifies which component is used. When new components become available in the Hinch component library, an application developer can immediately use them in XSPCL by simply specifying the appropriate class attributes. Following the requirements for components of Section 4.1, an application developer can use multiple copies of a component in different parts of the application, by specifying equal class attributes for all copies.

Each component has a unique set of parameters that specify component-specific properties and the communication streams that are connected to the component. The parameters satisfy both the requirement for component parameters and the communication requirements, as mentioned in Section 4.1. For each component instance, the application developer can supply different parameters. A <parameters> section contains the parameters for a component. Each parameter has a class attribute which specifies the parameter class. A component has five kinds of parameters:

1. Input stream parameters specify the streams that the component reads. The component receives the type information for these streams when the application creates it. The component checks if the input stream combination is acceptable. For example, a component can require that all input streams have the same element type and/or element size. In Listing 4.2, the component has one input stream named "big".

XSPCL uses the name attribute for specifying streams between components. When two or more components access a stream with the same name, the XSPCL application automatically creates a stream between these components.

The input streams have to be specified in the correct order. The XSPCL application connects the first input stream to the first communication port of the component, the second stream to the second port, etc.

The XSPCL compiler does not check whether the ports of two components are compatible, as the current prototype implementation does not have type information available for the communication ports. Future implementations
could include this information by annotating each component class with relevant type information. In the meantime, the underlying Hinch run time system checks the compatibility of two ports.

2. Output stream parameters specify the streams the component writes. Like the input streams, they are automatically connected based on their name attribute, they have to be specified in the correct order, and port compatibility is only checked at run time. In Listing 4.2 the component has one output stream named "small".

3. Input event parameters specify the event communication channels the component reads. Unlike streams, all event channels have the same type, which means that all channels can handle all types of events. Like stream parameters, event parameters have a name attribute, which XSPCL uses for creating event channels between different components. The class attribute for event parameters is eventq. An example input event parameter specification is: `<in name="eq" class="eventq"/>

4. Output event parameters are similar to input event parameters, except that the component writes to the event channel instead of reading it. For example: `<out name="eq" class="eventq"/>

5. Initialization parameters configure the component. For example, in Listing 4.2 the factor attribute specifies the down scale factor of the down scaler.

An XSPCL application passes the value attribute to the new function of the component when the application creates the component. Because the XSPCL compiler generates C code, the value may hold any valid C expression. The value can thus be a constant but also a complex expression with global variables from the header of the specification, as explained in Section 4.2.1.

An application developer can use initialization parameters for supplying a reference to a global data structure to multiple components. All components then share the global data structure. However, we discourage using the initialization parameters this way since XSPCL does not guarantee mutual exclusive access to this data. Moreover, when two components synchronize using shared data, it might cause deadlock as the component synchronization might interfere with the synchronization within the Hinch run time system, which all XSPCL applications use.

As explained above, a compiled XSPCL application supplies a component with the properties of the input streams it wants to connect to the component. The component then automatically adapts to the environment in which it is used, by using this information for setting its input and output port types. In case the component only has output ports, it determines the output port types by different means. For example, when it reads from a file, which is specified using an initialization parameter, it can derive the output port type from the contents of the file.
4.2. XSPCL LANGUAGE SPECIFICATION

After creating a component, the XSPCL application in turn derives the types of the output streams of the component from the types of the corresponding output ports. As explained above, the XSPCL application forwards this type information to the components that read this stream. The type information thus automatically flows from the beginning of the component graph to the end. XSPCL supports feedback loops, in which information flows in the reverse direction. However, feedback loops often require some extra scheduling dependencies, as we will explain in Section 4.2.6.

Besides these parameters, an application developer can use a <config> tag for giving the component a reconfiguration request upon creation. For example, in Listing 4.2, the component has a configuration setting with type fast, whose value is set to 1. All components have a reconfiguration interface at which they listen for reconfiguration requests (see Section 3.3.1). The tags inside the <config> differ for each reconfiguration request type, which is specified by the type attribute of the <config> tag. XSPCL supports all reconfiguration requests that are available in the Hinch run time system. Some reconfiguration requests are specific to a component, for example, a picture-in-picture blender can support changing the position of the blended picture. Other reconfiguration requests are applicable to many components. For example, the reconfiguration interface is used for telling a component which part of the input it has to process when it runs in data parallel mode. Data parallelism will be explained in Section 4.2.4.

4.2.3 Procedures

A <procedure> section groups multiple components together. A <call> tag inserts the content of another <procedure> section into the current <procedure> section. Together, these tags implement the procedural abstraction requirement mentioned in Section 4.1. All procedures have a unique name, which their name attribute specifies. The procedure named "main" is special. It defines the top-most procedure (entry point) in the application.

In XSPCL, a procedure acts as a normal component. The parameter specification in a <call> is the same as in a <component>, as explained above. It specifies the actual parameters given to the procedure. The parameter specification in a procedure specifies the formal parameters of the procedure. Of course, the actual parameters in the <call> match the formal parameters of the procedure. Using these parameters, the application developer can tune the procedure to the environment of the procedure call.

A procedure passes its actual parameters to the components inside the procedure. When a component uses one of the streaming parameters of the procedure that contains the component, the XSPCL application connects a stream that originates somewhere outside the procedure to the component. The generated XSPCL application automatically forwards the type information from the streaming parameters in and out of the procedure, as if the contents of the procedure were specified at the location of the <call>. A component uses the initialization parameters of its
surrounding procedure by simply using their names. XSPCL maps these parameters to variables which can be used in the initialization parameters of components.

The <body> part of a procedure describes the contained component tree, which contains components, procedure calls, or one of the other structures described below. XSPCL supports nesting procedures, including recursion.

Listing 4.3 shows an example of a procedure and its caller. Components B and C communicate using the "internal" stream that is only visible inside the procedure. Component C writes to the "external" stream that is exported using the parameters of the routine. At the level of the <call>, this stream is called routine_out, and it is connected to component A. The net effect is that C is connected to A. XSPCL supports input stream parameters and initialization parameters of a procedure similarly.
4.2. Parallelism

By default, when an XSPCL specification contains two successive components, the XSPCL application schedules them sequentially. Still, an XSPCL application can exploit parallelism between two sequential components by running multiple iterations of the application concurrently. Running multiple iterations of a single component concurrently is also possible, as explained in Section 3.3.3. Even an application that consists of a single component can thus benefit from a parallel architecture.

A `<parallel>` tag specifies parallelism within an iteration, as opposed to parallelism between iterations. Its `shape` attribute determines the type of parallelism. Depending on the shape, it also has other attributes. Within a parallel construct, `<parblock>` sections specify the application parts that can be run in parallel.

The XSPCL compiler maps all types of parallelism to a Hinch grouping component, which are explained in Section 3.3.3. When new grouping components become available, which support new parallelism types, XSPCL has to be extended with a new shape that maps to the new grouping component. We will now explain the types of parallelism XSPCL currently supports.

Pipeline Parallelism

An XSPCL application exploits pipeline parallelism by running components in different iterations concurrently. For example, Figure 4.2 shows two successive components. Within a single iteration, the XSPCL application can not exploit parallelism, since component B only runs when A has finished. However, while component B is still executing its first iteration, the second iteration of component A can run concurrently, thereby exploiting parallelism.

The application developer specifies the number of concurrently active iterations of an application in the header of its XSPCL specification using a `<cai>` tag, as explained in Section 4.2.1. If this number is low, an XSPCL application might not run enough iterations in parallel to use all available resources, which results in reduced performance. When it is high, the application will consume many resources, because many iterations are active. Therefore many communication streams hold temporary data, which can result in performance loss when the system runs out of memory and starts swapping memory pages to disk.

The optimal number of concurrently active iterations thus depends on the application and the architecture. However, since most applications already exploit...
parallelism within an iteration, the default value of four usually provides a good balance between the amount of parallelism in the application and resource usage. In future implementations, performance prediction tools could provide hints towards the optimal number of concurrently active iterations.

**Task Parallelism (shape=task)**

The components in each `<parblock>` section can run in parallel with the components in the other `<parblock>` sections. When all parallel sections have finished, the XSPCL application runs the successors of the parallel block in the data flow graph containing the task parallel group. For example, in Figure 4.3, C and D can run in parallel. The XSPCL compiler maps task parallelism onto the task parallel grouping component.

**Data Parallelism (shape=slice)**

XSPCL allows only one `<parblock>` section with this type of parallelism. The `<parallel>` tag has two extra attributes. The `n` attribute specifies the level of parallelism. The `dim` attribute specifies the dimension of parallelism. For example, with two-dimensional data, it specifies either a row-based or a column-based distribution.

Data parallelism is only possible when all components in the `<parblock>` section support it. The XSPCL compiler replicates the components in the `<parblock>` section `n` times, and configures each replica with its position within the parallel group and the dimension. All replicas can then run in parallel while each replica only performs part of the computation. In case of images these parts typically correspond to horizontal slices of the image, hence the name 'slice' for this shape. Figure 4.4 shows an example of expressing data parallelism, in which two replicas of component E run in parallel.

An application developer can also express data parallelism using the task parallel shape. However, this approach is cumbersome and error-prone. The application developer has to manually replicate the `<parblock>` section `n` times, and configure each component for processing part of the data. For example, Listing 4.4 shows how
4.2. XSPCL LANGUAGE SPECIFICATION

Listing 4.4: Expressing data parallelism using the task parallel shape

to express the data parallel block of Figure 4.4 using task parallelism. The listing does not show that all other attributes of the component, including all parameters, have to be copied for each replica of component E. With the slice shape, expressing data parallelism is much easier. the application developer only has to specify a single <parblock> section and a replication counter (n).

In fact, the XSPCL compiler internally performs exactly this transformation and
CHAPTER 4. THE XSPCL COORDINATION LANGUAGE

Figure 4.5: Data parallelism in XSPCL (Cross dependencies)

thereby relieves the application developer from this task. The XSPCL compiler then
maps data parallelism onto the task parallel grouping component. The XSPCL com-
piler only performs this transformation when the level of data parallelism is constant.
When \( n \) contains a global variable from the header, or a procedure parameter, the
XSPCL compiler can not determine the level of parallelism and falls back to using
the data parallel grouping component.

Resolving data parallelism at the XSPCL level has two small advantages over us-
ing the data parallel grouping component. First, it is slightly more efficient: With the
data parallel grouping component, all intermediate grouping components propagate
the data parallel configuration to their children. With the XSPCL transformation,
only the leaf-level components are configured. Second, it works with components
that do not have a \texttt{clone} function. As explained in Section 3.3.3, the data parallel
grouping component uses this function for creating replicas of the component and
thus requires the component to have this function.

The drawback of resolving data parallelism at the XSPCL level is that it results
in larger executable files, because the XSPCL compiler internally replicates the data
parallel components. Above a certain level of parallelism, using the data parallel
grouping component is probably preferable. We have not investigated this balance
because it is only a minor optimization.

Cross Dependencies (\texttt{shape=crossdep})

Cross dependencies express a special kind of data parallelism in which there are
neighbor dependencies between successive data parallel components, as shown in
Figure 4.5. Similarly to the slice shape, the crossdep shape has the \( n \) and \texttt{dim}
attributes that specify the level and dimension of parallelism, respectively.

Multiple \texttt{<parblock>} sections are allowed within the crossdep shape. Similarly
to the slice shape, the XSPCL application replicates the components in each
\texttt{<parblock>} section \( n \) times and configures the resulting components for performing
part of the computation. At the start of the parallel region, the XSPCL application
schedules all replicas of the first \texttt{<parblock>} section. The whole \texttt{<parallel>}
section is complete when all replicas of the last <parblock> have finished. When a cross dependencies section contains a single <parblock>, the parallelism type equals normal data parallelism using the 'slice' shape.

The XSPCL application generates neighbor dependencies between the replicated components in two successive <parblock> tags. A component replica with index i is only scheduled when the component replicas in the previous <parblock> with indices i-1, i, and i+1 have finished. Cross dependencies are very helpful for parallelizing a sequence of image filters. These filters need the data in the next and/or previous slice when computing boundary pixels.

The dependency structure within cross dependencies is fixed. When the application has a different dependency structure, a new grouping component must be created and XSPCL must be extended with support for this new structure.

**Nested Parallelism**

All forms of parallelism can be freely nested with one exception: When data parallelism is nested, each nesting level has to use a different dimension. Our system currently has this limitation because using the same dimension makes little sense. An application developer can easily circumvent this limitation by using a single data parallel grouping component for each dimension, for example by using a six-way column-based data parallel group instead of a three-way column-based parallel group within a two-way column-based parallel group. Figure 4.6 illustrates this example.

Figure 4.7 shows an example of nesting two sequential components (H and I) and two task-parallel components (J and K) within a cross-dependencies data-parallel structure. Listing 4.5 shows the corresponding XSPCL code. The XSPCL application replicates both the sequential group (lines 3–4) and the task-parallel group (lines 7–14) three times inside the cross dependencies group.
4.2.5 Alternating components

Some applications execute different actions in each iteration. For example, an MPEG-2 video encoder uses three different frame types for encoding the images in a video stream\cite{52}. I-frames contain a single frame and have no dependencies to other frames. P-frames and B-frames have motion vectors that refer to previously encoded I-frames and P-frames, which improves the compression ratio.

XSPCL accommodates these applications using an \texttt{<alternate>} group, as shown in Listing 4.6. In the first iteration, the application runs the sub graph in the first \texttt{<altblock>} section, which is the iframe component in this example. In the second

---

**Figure 4.7:** Example of nested parallelism

**Listing 4.5:** Example XSPCL code for nested parallelism

```xml
<parallel shape="crossdep" n="3" dim="2">
  <parblock>
    <component class="H"> </component>
    <component class="I"> </component>
  </parblock>
  <parblock>
    <parallel shape="task">
      <parblock>
        <component class="J"> </component>
      </parblock>
      <parblock>
        <component class="K"> </component>
      </parblock>
    </parallel>
  </parblock>
</parallel>
```

---

1. \texttt{<parallel shape="crossdep" n="3" dim="2">}
2. \texttt{<parblock>}
3. \texttt{<component class="H"> </component>}
4. \texttt{<component class="I"> </component>}
5. \texttt{</parblock>}
6. \texttt{<parblock>}
7. \texttt{<parallel shape="task">}
8. \texttt{<parblock>}
9. \texttt{<component class="J"> </component>}
10. \texttt{</parblock>}
11. \texttt{<parblock>}
12. \texttt{<component class="K"> </component>}
13. \texttt{</parblock>}
14. \texttt{</parallel>}
15. \texttt{</parblock>}
16. \texttt{</parallel>}

4.2. XSPCL LANGUAGE SPECIFICATION

iteration, it runs the second <altblock> section with the pframe component. Then it loops and runs the iframe component again in the third iteration, followed by the pframe component in the fourth iteration, etc. Although Listing 4.6 only shows <altblock> sections with a single component, an application developer can include any component tree in an <altblock> section. <alternate> groups can also be nested this way.

When components in different iterations access the same communication stream, the XSPCL compiler automatically creates dependencies between these components. For each component, the XSPCL compiler determines the set of components in the previous iteration that access a communication stream that is also accessed by the component. A component only runs when all components in this set have finished. In this case, finished means the components have completed their synchronous stage, in which it starts accessing its streams, as explained in Section 3.3.1. Therefore, exploiting parallelism across components in different iterations that access the same communication stream remains possible.

Without these extra dependencies, race conditions can occur when multiple concurrent iterations are active. For example, it could happen that the pframe component runs before the iframe component has started its first iteration. The pframe component then accesses the input and output streams before the iframe component, and the frame ordering becomes incorrect.

Listing 4.6: Example alternate group in XSPCL
4.2.6 Feedback loop

Some streaming applications contain a feedback loop in which a component at the end of the iteration sends data to a component at the beginning of the iteration. For example, an MPEG-2 encoder uses previously encoded image frames for encoding the current image frame.[52]

The application developer has to explicitly specify the feedback loops in the application and their parameters, because the XSPCL compiler must generate extra dependencies between the components in the feedback loop. For example, Figure 4.8 shows two components that communicate using a normal forward stream and a feedback stream. Listing 4.7 shows the corresponding XSPCL code. In the first iteration, component L does not read the feedback stream because M has not run yet. In the following iterations, L does read the feedback stream, which means it can only run after M has finished.

Therefore, the XSPCL compiler generates a feedback dependency from M to L. The XSPCL application initially fulfills this dependency, allowing L to run its first iteration. In the following iterations, the dependency is fulfilled when M is complete.
4.2. XSPCL LANGUAGE SPECIFICATION

Figure 4.8b shows the dependency and grouping structure of L and M, along with the XSPCL specification for this feedback loop.

The delay attribute of the <feedback> tag specifies the feedback delay, which is the number of iterations of the feedback loop dependency. In the above example, L reads the feedback stream one iteration after M has written to it. When there are multiple iterations between writing and reading the feedback stream, the feedback loop can be entered multiple times, without waiting for the feedback dependency. Multiple iterations of the feedback loop can then run in parallel, which improves efficiency. The XSPCL compiler maps a feedback loop to a special grouping component, which initially schedules the sub graph in the feedback loop delay times. Then it only schedules a new iteration when the oldest iteration has finished.

As explained in Section 4.2.2, an XSPCL application automatically forwards the communication stream types from the producing components to the consuming components. However, this type forwarding only works for streams that flow in the direction of the application graph, as an XSPCL application initializes the components in the order of the application graph. Because feedback streams flow in the opposite direction, type forwarding is not possible for these streams. An XSPCL application therefore informs the component that the type of a feedback stream is unknown, instead of supplying an actual type. The component has to derive the type information for these streams by other means, for example, using the type information of the other streams it reads.

4.2.7 Reconfigurability

An application can declare (groups of) components optional by encapsulating them inside an <option> or <choice> section, as shown in Listings 4.8 and 4.9 respectively. Both tags have a name attribute which identifies the option or choice.

Listing 4.8: XSPCL manager with an option

```xml
<manager eventq="eq">
  <managed>
    <component class="A"> </component>
    <option name="opt">
      <component class="B"> </component>
    </option>
  </managed>
  <event type="key" value="e">
    <enable name="opt"/>
  </event>
  <event type="key" value="d">
    <disable name="opt"/>
  </event>
</manager>
```

Figure 4.8b shows the dependency and grouping structure of L and M, along with the XSPCL specification for this feedback loop.

The delay attribute of the <feedback> tag specifies the feedback delay, which is the number of iterations of the feedback loop dependency. In the above example, L reads the feedback stream one iteration after M has written to it. When there are multiple iterations between writing and reading the feedback stream, the feedback loop can be entered multiple times, without waiting for the feedback dependency. Multiple iterations of the feedback loop can then run in parallel, which improves efficiency. The XSPCL compiler maps a feedback loop to a special grouping component, which initially schedules the sub graph in the feedback loop delay times. Then it only schedules a new iteration when the oldest iteration has finished.

As explained in Section 4.2.2, an XSPCL application automatically forwards the communication stream types from the producing components to the consuming components. However, this type forwarding only works for streams that flow in the direction of the application graph, as an XSPCL application initializes the components in the order of the application graph. Because feedback streams flow in the opposite direction, type forwarding is not possible for these streams. An XSPCL application therefore informs the component that the type of a feedback stream is unknown, instead of supplying an actual type. The component has to derive the type information for these streams by other means, for example, using the type information of the other streams it reads.

4.2.7 Reconfigurability

An application can declare (groups of) components optional by encapsulating them inside an <option> or <choice> section, as shown in Listings 4.8 and 4.9 respectively. Both tags have a name attribute which identifies the option or choice.
CHAPTER 4. THE XSPCL COORDINATION LANGUAGE

An application can enable or disable and <option> at run time, which means the application adds or deletes the contained sub graph, respectively. An option may have an enabled attribute. If its value contains 'yes', the XSPCL application enables the option at the start of the application. By default, an option is disabled.

A <choice> section contains several <alternative> sections that contain alternative sub graphs. Exactly one alternative is always enabled, while the others are disabled. Each alternative has an id attribute which identifies the alternative. The default attribute of the <choice> tag specifies the alternative that is enabled at the start of the application.

As explained in Section 3.2.4, Hinch only modifies a sub graph when it is idle because the components inside the sub graph need to synchronize. The application developer must therefore enclose all optional sub graphs inside a special <manager> section. The XSPCL compiler creates a manager grouping component for each <manager> section. A manager can temporarily suspend execution execution of its sub graph. A manager is responsible for the consistency of its contained sub graph and implements the container mentioned in Section 4.1.

Using its eq attribute, each manager is associated with an event queue. When the manager runs, it polls its event queue for new events. A <manager> section contains multiple <event> tags that specify the events the manager recognizes. When an event from the event queue matches one of the listed events, the manager performs the actions inside the <event> tag. The application developer can define one or more of the following actions for each event:

Listing 4.9: XSPCL manager with a choice

```xml
1  <manager eventq="eq">
2   <managed>
3       <choice name="c_or_d" default="1">
4           <alternative id="1">
5               <component class="C" />
6           </alternative>
7           <alternative id="2">
8               <component class="D" />
9           </alternative>
10        </choice>
11     </managed>
12     <event type="key" value="c">
13        <choose name="c_or_d" id="1"/>
14     </event>
15     <event type="key" value="d">
16        <choose name="c_or_d" id="2"/>
17     </event>
18  </manager>
```

4.2. XSPCL LANGUAGE SPECIFICATION

- `<enable name="..."/>`
  Enable the `<option>` with the given name.

- `<disable name="..."/>`
  Disable the `<option>` with the given name.

- `<toggle name="..."/>`
  Toggle the `<option>` with the given name. If it was enabled, it is disabled, and vice versa.

- `<choose name="..." id="..."/>`
  Within the `<choice>` with the given name, disable the current alternative and enable the `<alternative>` with the given id.

- `<forward eq="..."/>`
  Forward the event to another event queue. This action does not require reconfiguration. An XSPCL application immediately forwards events as soon as it detects them.

- `<config type="...">...</config>`
  Send a reconfiguration request to all components in the managed sub graph using the component reconfiguration interface. The syntax for this action is equal to a `<config>` section within the definition of a component, as explained in Section 4.2.2. An XSPCL application sends the request to the top-level component in the managed sub graph, which recursively forwards it to its children.

  A reconfiguration request can affect multiple components. These components can be active in different iterations. Therefore, an XSPCL application synchronizes all components before sending reconfiguration requests, by waiting until the sub graph is idle. For example, an application might use separate image blending components for each color component, which blend a small image into a large background image. Using their reconfiguration interface, the application instructs the components to move the small image relative to the background. When an XSPCL application does not synchronize these components, some components will perform the movement immediately in the current iteration. Other components, that process different colors, might be active in later iterations, and delay the movement until their next iteration. The resulting output images, in which all color components are combined, are incorrect until all components have performed the movement.

- `<exit/>` Terminate the application. The XSPCL application notifies the manager that it should stop scheduling new sub graph iterations. When all pending iterations are complete, the application automatically terminates.
4.3 Implementation

We implemented the XSPCL compiler using XSLT transformations. XSLT is a powerful declarative language for transforming XML documents, which is based on XML[40]. The XSPCL compiler first performs a number of XSPCL-to-XSPCL transformations that simplify the original specification. The intermediate specifications remain in XSPCL format. The XSPCL compiler performs the following XSPCL-to-XSPCL transformations:

1. Resolve compile-time application flags. The XSPCL compiler removes the parts of the application that are disabled. As explained in Section 4.2, an application developer can enclose part of an application in special sections. Depending on compile-time flags, the compiler enables or disables the enclosed parts.

2. Transform data parallelism into task parallelism. The XSPCL compiler converts data parallelism to task parallelism whenever possible, using the transformation we mentioned in Section 4.2.4.

3. For loop expansion. The XSPCL compiler replaces all for loops in the XSPCL specification by their expansions.

4. Alias conversion. An application can use aliases for often occurring components. The XSPCL compiler converts these aliases into normal component specifications. An application developer with knowledge of XSLT can easily extend the XSPCL compiler with aliases for often occurring code.

5. ID assignment. For internal use, the XSPCL compiler assigns a unique identifier to each manager.

After these XSPCL-to-XSPCL transformations, the XSPCL compiler transforms the simplified XSPCL code into a source file in C that uses the Hinch API for building the application. The XSPCL compiler comes with a small C source file with common helper routines for XSPCL applications. The application binary is built by linking the generated source code to the helper routines, the various components used by the application and the Hinch run time system.

4.3.1 Reconfiguration

As mentioned in Section 4.2.7, a <manager> section surrounds each reconfigurable sub graph in an XSPCL application. The XSPCL compiler maps a <manager> section to a manager grouping component, which Section 3.3.6 describes in detail. The XSPCL compiler automatically creates detect and idle functions for each manager, including all necessary structures and variables. The manager calls the detect function for each event it receives, as soon as it detects the event. The detect function informs the manager if its contained sub graph requires reconfiguration. If
it does, the manager halts the sub graph. When the sub graph is idle, the manager calls the \texttt{idle} function, which performs the actual reconfiguration.

In the detect function, an XSPCL application checks if the event matches one of the \texttt{<event>} tags. If it does not match any \texttt{<event>} tag, the application ignores the event. If it matches, the application checks if reconfiguration is required by checking the state of the options and choices affected by the event. When all affected options and choices are already in the required state, as specified by the event, reconfiguration is not required. For example, if an event specifies that an option must be enabled, and the option is already enabled, an XSPCL application performs no action and informs the manager that reconfiguration is not required. Reconfiguration is never required for forwarding events to another event queue. When an XSPCL application needs to send reconfiguration requests to components, it always informs the manager that reconfiguration is required, because all components that receive the request must be synchronized to each other, as explained Section 4.2.7.

An XSPCL application creates and initializes the components that need to be added in response to an event as soon as it detects the event in the \texttt{detect} function, even though the contained sub graph is still active. This way, reconfiguration time is reduced, as these components are not created and initialized during the actual reconfiguration, when the manager calls the \texttt{idle} function. In the \texttt{idle} function, an XSPCL application performs two simple actions. First, it adds the created components to the sub graph. Second, it synchronizes the new components to the sub graph. Besides reconfiguring the sub graph, an XSPCL application also sends reconfiguration requests in the \texttt{idle} function.

Building \texttt{detect} and \texttt{idle} functions by hand is a difficult task, because the structures involved are complex. It involves handling different kinds of events, which require different kinds of reconfiguration actions, which operate on complex application graphs. By using simple primitives for declaring optional sub graphs and events that trigger reconfigurability, the XSPCL compiler relieves the application developer from these difficult tasks and allows the application developer to quickly build reconfigurable applications.

\section{4.4 Experiments}

We investigate the overhead of XSPCL by comparing applications that directly use the Hinch API, to similar XSPCL applications, for which the XSPCL compiler generates the calls to the Hinch API. We will use the term \textit{Hinch application} for referring to an application that directly use Hinch. Both versions use the same components. Their performance can be different, however, for several reasons:

- Since XSPCL is a generic language, it must handle all border cases, even though they are not applicable to the application. In a hand-written Hinch application, the non-applicable border cases are omitted.
- XSPCL does not support all application-specific optimizations. We have to make a trade-off between providing an elegant abstraction level and supporting
optimizations. Although many optimizations are still possible using XSPCL, there are some optimizations that can only be expressed in hand-written Hinch applications.

- For administrative purposes, the XSPCL compiler creates some extra internal data structures, which incur some management overhead.
- The memory layout of the hand-written and the XSPCL versions differ, as XSPCL applications use some extra data structures. Moreover, both versions initialize various structures, including components and streams, in a different order. The difference in memory layout may impact cache performance.

The applications we use are a synthetic benchmark application, a convolution filter application, and a picture-in-picture application. These are the same applications we used in Section 3.4 for evaluating the Hinch run time system. We refer to Section 3.4.1 for the details of these applications.

The XSPCL applications have the same component structure as the corresponding Hinch applications, with one exception for the Filter and PiP applications: All XSPCL versions of these applications contain a manager component, even when the application does not require a manager. This modification simplifies switching between the various versions of these applications. The non-reconfigurable versions of the Hinch applications do not contain this manager. The extra manager does not incur much overhead, as it does not perform any significant computation. When the application does not send events to the manager, it acts as a dummy component and does not affect the scheduling of the other components in the application.

Similarly to Section 3.4, we run the applications on a DAS-3 node, a Cell blade, and a SPARC Enterprise Niagara server. These architectures support 4, 4, and 64 concurrent threads, respectively. As we explained in Section 3.4, we repeat each measurements eleven times on DAS-3 and five times on the Cell blade and the Niagara. Then we take the average of the set of repeated measurements. Section 1.2 gives more details about the architectures.

4.4.1 XSPCL Overhead

Figure 4.9 shows the overhead of the XSPCL applications relative to the Hinch applications. For example, an overhead of 10% means the XSPCL application is 10% slower than the corresponding Hinch application. On DAS-3 and the Cell blade, we measure the overhead using one thread and using four threads, which is the maximum number of threads supported by the architecture. On the Niagara, we show the overhead using 1, 4, 16, and 64 threads, since the architecture supports 64 threads. The measurements of the Hinch applications are equal to those in Section 3.4.

In the measurements we present, the standard deviation between repeated executions is less than 6% of the average measurement. However, there are a few exceptions to this rule, in which both the XSPCL application and the Hinch application exhibit high standard deviations. The PiP-1 application suffers from load
imbalance on DAS-3. With 4 threads, the standard deviation in the eleven repeated measurements is 9.5% for the XSPCL PiP-1 application, and 9.4% for the Hinch PiP-1 application.

In the Benchmark-Slicing application on the Niagara, the threads are often idle, waiting for jobs. Because the application uses slicing, groups of 80 components become runnable at a time. This behavior results in contention between the idle threads, as they fetch jobs simultaneously from the central job queue. Since the contention is unpredictable, the execution time varies considerably between different executions. On the Niagara, the Hinch and XSPCL Benchmark-Slicing applications
have standard deviations of 26.0% and 40.4%, respectively.

We will now describe the results for each architecture, and give some concluding remarks afterwards.

**DAS-3**

On DAS-3, the overhead of XSPCL is always less than 5%. Since the PiP applications have a relatively low compute intensity, the overhead is most visible in these applications.

**Cell blade**

On the Cell blade, we use the two PPE cores for our measurements, which support two hardware threads each. We do not use the SPE coprocessors. The XSPCL applications present some interesting results on this architecture. When using four threads, the overhead of the Benchmark applications ranges from 6.7% to 7.7%, which is significant. On the other side, the XSPCL Filter applications perform 10.8% to 19.4% better than the Hinch Filter applications. The performance of the XPSCL PiP application is only a little better than its Hinch equivalent, on average.

We have further analyzed these applications, and concluded that these big differences result from faster execution of the core computations in the applications. Since the processor speed and the executed instructions in the core computations are equal, the reason must lie in the changed memory layout, which affects cache performance.

When using the simultaneous multithreading (SMT) capabilities of the Cell processor, the impact of a changed memory layout may be severe. The Cell processor supports two SMT threads that share the L1 and L2 caches of the processor. Changing the memory layout of the application therefore not only affects the cache performance of a single thread, but indirectly also the cache performance of the other thread that uses the same core. Hily et al. confirm that the L2 cache is a bottleneck for SMT systems. The L1 cache has a large cache line size of 128 bytes, which also causes much cache interference between both threads.

Other measurements, using two and three threads, confirm that the SMT behavior is responsible for the big differences. With two threads, each thread has its own core and the differences remain small. With three threads, the differences are significantly larger, although they are still smaller than with four threads. Figure 4.9b confirms that big differences do not occur when running a single thread.

**Niagara**

On the Niagara, the XSPCL Benchmark-Slicing application performs 28% better than its Hinch equivalent. However, as mentioned before, this application exhibits a high standard deviation. We can therefore not attach conclusions to the result with this specific application. The other XSPCL Benchmark applications and the XSPCL Filter applications show no significant overhead.
4.5 SUMMARY

In the XPSCL PiP application, the overhead increases with the number of threads on the Niagara. The XSPCL application contains some small management structures, which enlarge the part of the application in which it does not exploit all possible parallelism. The XSPCL application therefore exploits less parallelism than the Hinch application. The impact of having less parallelism increases with the number of threads, which causes the increasing overhead. Detailed measurements confirm that the threads are idle for a significantly longer period of time in the XSPCL application than in the Hinch application. The total time spent in computations is not significantly different in both applications. This overhead increase occurs only with the PiP application because PiP is very sensitive to overhead, as shown in Section 5.4.

Conclusion

We can conclude that the overhead of using XSPCL is less than a few percent when the compute intensity of the application is high enough. The Benchmark and Filter applications exhibit this behavior, even though they are still relatively simple applications. In more advanced applications, with more intense computations, the XSPCL overhead will also be low. When the compute intensity of the application is low, any form of overhead impacts the performance of the application, including XSPCL overhead. The PiP application, which already suffers from the minimal overhead of the Hinch run time system, therefore also suffers from XSPCL overhead.

4.5 Summary

We have presented XSPCL, a coordination language for streaming applications. In the SP@CE programming environment, XSPCL is an intermediate language between a graphical front-end and the Hinch run time system. An XSPCL specification can also be fed to and annotated by performance prediction tools.

An application developer can easily express advanced structures in XSPCL. Besides streaming communication, these structures include task- and data parallelism, procedural abstraction, dynamic reconfiguration and asynchronous user event handling. We have implemented a prototype XSPCL compiler that converts an XSPCL specification to an application that uses the Hinch run time system. When the application has a decent compute intensity, the overhead of using XSPCL is at most a few percent.

Although XSPCL applications are built on top of the Hinch run time system, we believe its primitives are also applicable to other run time systems for streaming applications. Future research on XSPCL could focus on building streaming applications in the area of High Performance Computing (HPC), using shared memory, distributed memory and/or hybrid architectures, e.g., a cluster of Cell processors. We believe XSPCL can easily be extended with support for this kind of applications, as long as the application can be expressed as a streaming application.
Currently, XSPCL does not have primitives for expressing deadlines in real-time systems. When these primitives are added in a future implementation, the XSPCL compiler can verify that the application meets its deadlines by deriving its worst case execution time (WCET). First, it annotates the individual components with their WCET, for example, based on performance prediction results. Based on the various grouping constructs in the component graph and the available resources, the XSPCL compiler can recursively calculate the WCET of the whole application.
Chapter 5

Gordon: A run time library for the Cell processor

The Cell processor is one of the most powerful processors currently available. However, this power comes at a price. The heterogenous multi-processor system-on-chip (MPSoC) architecture of the Cell makes exploiting all resources difficult. A Cell software developer has to deal with complex issues like heterogeneity, synchronization between cores, different communication mechanisms, limited amounts of memory per core, manual data transfer, communication latency hiding, load balancing, multiple sources of overhead, and many others. In essence, the main challenge for the Cell is offering system software that unlocks the full potential of the processor without burdening the software developer.

In this chapter, we present and evaluate Gordon, a run time library that accomplishes this task by hiding the low level details of using high-performance coprocessors behind a simple API for submitting jobs to the coprocessor. Gordon is also known as the Cell run time library (Cell-RTL) [97]. Gordon supports both task- and data parallelism and has low overhead. While we have implemented Gordon on the Cell processor, we believe that a similar API and run time library can be used on other heterogenous MPSoC architectures.

After solving the problem of efficiently using the coprocessors, building hybrid streaming applications that use both the main processors and the coprocessors remains a difficult task. Besides the description of Gordon itself, we will show how we integrated support for Gordon into the SP@CE framework. A SP@CE component can execute computations asynchronously on one of the available coprocessors using Gordon. Section 1.5 describes the SP@CE framework for streaming applications.

Achieving optimal performance in hybrid applications is one step further: The

---

application needs to balance the load between heterogeneous resources and migrate tasks between different resource types. We solve this problem in Section 5.4 using dynamic reconfigurability, which is fully supported by SP@CE. At run time, a SP@CE application can dynamically replace a component by another version that uses different resources.

In Gordon, the coprocessors send short notification messages to the main processor for synchronization purposes. The Cell processor offers two special mechanisms for sending these messages, besides the default communication mechanism. For achieving optimal performance, we evaluate these mechanisms in the context of Gordon and choose the mechanism that performs best. In our experience, the two special mechanisms are redundant and do not have any added value over the default communication mechanism.

Gordon also employs an optimization called job chaining, which is also applicable to other heterogeneous MPSoC architectures besides the Cell. With job chaining, the application combines multiple jobs and Gordon schedules them as one entity, which reduces overhead. Moreover, this technique allows sharing data between successive jobs, which further reduces overhead.

This chapter first presents a description of the Cell processor and its difficulties in Section 5.1. A short survey of other programming environments for the Cell processor and their solutions to these difficulties follows in Section 5.2. Then, Section 5.3 describes Gordon in detail, followed by a description of creating hybrid SP@CE applications that use both the main processors and the coprocessors in Section 5.4. Section 5.5 presents a detailed performance analysis of Gordon. Finally, Section 5.6 summarizes this chapter.

5.1 The Cell Processor

The Cell processor [72] is one of the most powerful processors currently available. It can be found in machines ranging from game consoles to supercomputers [128]. The official name of this processor is Cell Broadband Engine or Cell B.E. Figure 5.1 shows a schematic overview of the first generation Cell processor. The processor contains nine processing cores that are connected by the Element Interconnect Bus (EIB), which is also connected to main memory. One of the cores is the Power Processing Element (PPE), which has a general purpose PowerPC core with two hardware threads. It can access all memory directly. The PPE coordinates the other eight cores, which are Synergistic Processing Elements (SPEs).

Figure 5.2 shows a schematic overview of an SPE. These cores have a different instruction set than the PPE, optimized for SIMD vector processing [60]. A large register file of 128 128-bit wide registers allows fast computation kernels, as the kernel can keep many data in registers instead of memory. An SPE can not directly access main memory and therefore does not contain any cache logic. Instead, it has 256 kilobytes of local on-chip memory which it accesses directly. Using an integrated DMA engine, an SPE asynchronously transfers data between main memory and local memory.
The SPEs have a relatively simple design, yet they have a high peak performance of 25.6 GFlop/second. The PPE and the SPEs are clocked at 3.2 GHz and can execute four single precision floating point multiply-add operations in each cycle. The combined peak performance of all eight SPEs and the PPE is therefore 230.4 GFlop/second.

### 5.1.1 Problems

Because of the heterogeneous architecture and the simple design of the SPEs, a developer needs to overcome many problems for achieving optimal performance. Some problems are related to a specific application. For example, each application requires different SIMD code for achieving optimal performance. Other problems can be solved generically for all applications and are best handled in a generic programming environment for the Cell architecture. Table [5.1] summarizes all problems and the tasks needed to address them. The tasks are split into application-specific tasks, performed by the developer, and generic tasks for the programming environment. We have identified the following problems regarding the Cell architecture:

- **Initialization.** An application needs to initialize the SPEs before it can use them. This task consists of low-level routines that are similar for all applications. The programming environment therefore performs this task.

- **Limited memory.** An SPE can only execute routines whose input and output data fit in its local memory. When the data does not fit in the local memory, the application has to split it into multiple parts. The routines at the SPE have to be modified accordingly. When the data has a regular shape, this splitting can be as easy as invoking the routine multiple times on different independent parts of the data. However, splitting is a tedious task when the data structures are complex, or when there are dependencies between the split routines.

A programming environment can only automatically split routines if this environment is specialized towards a limited set of applications with fixed regular data structures. With a generic programming environment that supports all applications, the developer has to split routines, as each application has different data structures. The programming environment in turn should be flexible regarding the data structures that are transferred between main memory and SPE local memory.
**Table 5.1:** Problems of programming the Cell architecture

<table>
<thead>
<tr>
<th>Problem</th>
<th>Application-specific tasks (Developer)</th>
<th>Generic tasks (Programming environment)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-level initialization</td>
<td></td>
<td>Initialize SPEs</td>
</tr>
<tr>
<td>Limited memory</td>
<td>Task splitting</td>
<td>Flexible structures; Task ordering; Transfer data between tasks</td>
</tr>
<tr>
<td>Code optimizations</td>
<td>SIMD; loop unrolling; ...</td>
<td>Allow optimizations</td>
</tr>
<tr>
<td>Memory management and communication</td>
<td>Specify transferable data</td>
<td>Memory allocation; Data transfer using DMA; Multi-buffering; Caching</td>
</tr>
<tr>
<td>Load balancing</td>
<td>Multiple task implementations</td>
<td>Monitor load; Migrate tasks, also between PPE and SPE</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Isolate routines</td>
<td>Synchronize PPE and SPE</td>
</tr>
</tbody>
</table>

When there are dependencies between routines, the programming environment should allow specifying a certain ordering of the routines. Ideally, it should also allow data transfer from one routine to another at low cost.

- **Code optimizations.** For optimal performance, a developer has to use SIMD code and other optimizations, such as loop unrolling. A generic programming environment can not perform this task because compilers are currently unable to apply these optimizations\[81, 104\]. However, progress is being made towards generic SIMD support\[64\].

Using SIMD optimizations, performance can quadruple when performing four 32-bit floating point or integer operations per cycle rather than one. Section 5.5.1 will show that the performance gain of using all low-level code optimizations is more than one order of magnitude. Because of these huge performance gains, a programming environment for the Cell should always allow a developer to perform these optimizations.

- **Memory management / Communication.** The local memory of the SPE can be seen as a cache of main memory that is fully managed by the application, instead of cache logic. An application thus has to manually transfer data between main memory as needed, which adds to the complexity of writing programs.

The DMA engine in the SPE supports both synchronous and asynchronous data transfers. For optimal performance the application should asynchronously...
5.1. THE CELL PROCESSOR

transfer data, which hides the latency of the transfer. While performing calculations, it should transfer results of previous calculations, and retrieve input for future calculations. The application thus has to perform multi-buffering, which is complex.

The programming environment should support multiple data layouts. The most basic and most efficient layout is a continuous block of memory. For data that is not in a continuous block, the DMA engine supports DMA lists. Each list element specifies a different block in main memory. In SPE local memory, all blocks are concatenated and form a single memory buffer. With DMA lists, multiple small transfers are thus grouped into one large transfer, which improves performance. The programming environment should therefore use DMA lists when possible. For data with unpredictable access patterns, the programming environment should supply a generic software cache mechanism.

Since memory management applies to all applications, the programming environment should perform these tasks and relieve the developer from these difficulties. Ideally, the developer only specifies which data has to be transferred. The programming environment then performs the actual transfers, using multi-buffering and DMA lists. This approach has proven successful in the ENSEMBLE communication library[33].

- Load balancing. The load needs to be properly balanced between the PPE and the SPEs. Optimal load balance is achieved when all cores execute routines that are equally compute-intensive. Although this approach could work for simple applications that execute a single routine on all cores, achieving optimal load balance is challenging for more complex applications that run a variety of different routines.

A solution for these complex applications is using dynamic load balancing. The application needs to keep track of the load of all cores and migrate tasks if the load is unbalanced. Migrating tasks between two SPEs is relatively easy, because all SPEs are equal. Migrating a task between the PPE and an SPE is more difficult, because they have different architectures. The task thus needs to be implemented twice, which might require a different optimization strategy for each implementation.

Like memory management, load balancing applies to all applications that use multiple cores and is best handled by a generic programming environment. The developer still needs to supply multiple implementations of a single program to this environment, which can be cumbersome. The environment needs to manage multiple implementations of a single task.

- Synchronization. All multi-threaded applications have to avoid race conditions by properly synchronizing all cores. The developer should not be bothered by this problem as a generic programming environment can perform this task. Still, the developer has to be aware of the parallel architecture and make sure that all concurrent routines are isolated with respect to the data they access.
In many ways, these problems resemble those of other heterogeneous MPSoC architectures, such as network processors[51]. The fact that these processors never lived up to the initial hype is commonly attributed to programming complexity. To avoid a similar fate for the Cell, programmability is key[39].

5.2 Related programming environments

Several other projects address the complexity of programming the Cell processor. Buttari et al. give a good overview of the various programming environments and the projects implementing them[31]. A challenge faced by most of these systems is that the particularities of Cell-like architectures (heterogeneous cores with local memories) make it hard to apply solutions for scheduling and communication that are based on shared memory[59, 101].

The Linux kernel handles many low-level details of programming the Cell processor. The kernel has special system calls for using the SPEs. At the PPE, the libspe2 library provides a layer on top of the kernel with functions for initializing SPEs and communicating with SPEs[68]. The SPE compiler provides a small communication library to the code that runs on the SPEs. This environment is still very basic: It allows using the SPEs but it does not solve problems like load balancing, synchronization between PPE and SPE code, and memory management. These problems have to be solved by adding another layer to this environment.

The Offload API, used by Charm++[79] has the closest resemblance to Gordon. Both systems provide an API for executing asynchronous tasks on the SPEs. The main difference is that Gordon can execute chains of tasks as one entity, which enables various optimizations. Furthermore, there is currently no performance evaluation available for the Offload API, whereas we will present a detailed performance evaluation of Gordon.

The Accelerated Library Framework (ALF)[66] is a generic library for offloading computations to accelerators. In the case of Cell, the SPEs are the accelerators. ALF comes with a complex interface as ALF is very flexible, whereas Gordon has a simple interface that hides most complexity from the user. Another important difference is that ALF only supports data parallelism, as it only allows one function on the SPEs. In contrast, Gordon has an SPE function library with multiple functions and allows both task and data parallelism.

Several other projects include a run time library that is similar to Gordon. These include the MultiCore Framework[23], Cell SuperScalar[21], and the Octopiler compiler[58]. The MultiCore Framework handles multidimensional data at the coprocessors. The communication between the main processor and the coprocessors resembles streaming. However, full streaming applications with multiple interacting components are not supported by this framework. Contrary to our approach, The MultiCore Framework uses the coprocessors synchronously, which results in low resource utilization. When the coprocessors are busy, the main processor is idle, waiting for the coprocessors. When the main processor is busy, the coprocessors are idle.
5.2. RELATED PROGRAMMING ENVIRONMENTS

Cell SuperScalar and the Octopiler use compiler technology for generating applications that use the SPEs. The user has to specify the parallel parts of the application, which are then automatically offloaded. Although this approach is viable for simple applications, we believe it will fail for streaming applications with complex communication patterns and corresponding data dependencies. In contrast, SP@CE fully supports these complex patterns in applications that use both the PPE and the SPEs. Moreover, SP@CE allows migrating tasks between the PPE and the SPEs and supports dynamic load balancing.

Graphics processors are increasingly used as coprocessors, for example by CUDA and Brook for GPUs[29, 98]. Although CUDA is often used for programming GPUs, it does not explicitly support streaming applications. Besides (and because of) the different coprocessor types, the computation kernel granularity in Brook for GPUs differs from the Gordon kernels. A Brook kernel typically computes a single element, say one image pixel. A Brook application exploits parallelism by computing multiple pixels at once. A Gordon kernel computes multiple elements, typically part of an image. A Gordon application exploits parallelism by computing multiple images or multiple image parts at once.

The Multicore Streaming Layer (MSL) executes application kernels on the SPEs using static or dynamic load balancing. The compiler for the StreamIt language adds support for the Cell architecture using this layer[131]. Contrary to SP@CE, the main processor only runs control code and is not used for computations in this system. Also, dynamic reconfiguration and event communication are not supported.

When an SPE in the Cell processor has finished a job, it notifies the PPE. There are various approaches for this notification, which resemble interrupt mitigation techniques for network packet processing processors[36, 82, 106]. The main difference is that network packets originate from external sources, whereas the SPE jobs and their notifications originate internally, which provides room for some optimization as the maximum number of notifications that occur is equal to the number of scheduled jobs.

Support for streaming applications is commonly found in network packet processing frameworks[22, 74]. The systems used in these frameworks tend to be static, with little or no support for scheduling or load balancing by pushing jobs to idle resources. Also, the communication model for NPUs is different from that of the Cell, as the local memory on NPU cores is in the order of a few kilobytes, and support for DMA transfers to and from local memory is lacking. Similarly, the idea of bypassing the OS as much as possible when performing intensive I/O is explored in [125].

TCP Offload Engines (TOEs) are another example of streaming on heterogeneous architectures within network processing[43]. The processing of a TCP/IP network stack is offloaded to a Network Interface Card (NIC) that contains a TOE, to relieve the main processor of this task. However, there are many performance and deployment problems of using TOEs[50], which confirms that developing streaming applications on heterogeneous architectures is difficult.
5.3 Gordon

Gordon is a run time library that facilitates programming the Cell processor by providing a simple interface for using the SPEs. Although Gordon has been developed specifically for the Cell, its simple interface and many optimizations may be applied to similar MPSoC architectures as well. We have identified the following requirements for Gordon:

1. Gordon has to perform as many generic tasks as possible, as identified in Section 5.1. Any task that is performed by Gordon does not have to be implemented by the developer anymore.

2. Gordon has to supply a simple interface towards the higher layers and towards the developer. Without a simple interface, developing applications for the Cell processor remains difficult, which is undesirable.

3. Gordon should have low overhead. Without low overhead, the benefits of using Gordon do not outweigh its cost and Gordon will be useless.

In our design, Gordon fully manages the SPEs and performs all but two of the generic tasks we identified. Because Gordon only focuses on using the SPEs, it does not migrate tasks between the PPE and the SPEs. Consequently it also does not perform load balancing between them. As we will show in Section 5.4 a layer on top of Gordon performs these tasks.

The interface to Gordon is designed to be as simple as possible, which satisfies the second requirement. It mainly consists of a function that asynchronously executes a function on an SPE, and a function that checks for completion of a previously executed SPE function. Section 5.3.1 will provides a more detailed description of the Gordon API.

We achieved low overhead by using a bottom up approach and evaluating the various synchronization and communication alternatives offered by the Cell. We then selected the ones that were most efficient. In addition, we analyzed the bottlenecks in the model and added primitives to Gordon for optimizing inter-core communication performance even further. For instance, the communication between the PPE and the SPEs is a bottleneck for short jobs. Gordon attacks this bottleneck by supporting batching multiple short jobs into a job chain, so that the overhead is incurred once per chain, rather than for each job.

5.3.1 Application Programming Interface

The API of Gordon is based on offloading jobs to the SPEs. A job is a self-contained application part that performs some computation on input data and produces output data. Gordon executes jobs asynchronously on the SPEs. When a job has finished, Gordon notifies the application using a call-back function.

A developer uses the Gordon push job function for sending jobs to the SPEs. A developer can check if a job has finished using the Gordon poll function, and/or wait
until the job has finished using the Gordon \texttt{wait} function. The \texttt{pushjob} function has a job description and an optional call-back function as arguments. Gordon calls the call-back function when the job has finished.

The job description contains a function identifier, allocation buffers and I/O buffers. The function identifier indicates which function the SPE executes. Each SPE has a library with the available functions, as Section 5.3.3 will explain.

The allocation buffers specify that a memory buffer has to be allocated on the SPE. Gordon performs all memory management on the SPEs. The SPE functions may not allocate memory themselves because it can interfere with the multi-buffering within Gordon. Allocation buffers provide a means around this limitation. Using these buffers, a developer can specify memory requirements before the execution of an SPE function.

Conforming to the task of flexibility regarding data structures, mentioned in Section 5.1.1, Gordon supports three kinds of I/O buffers: Input, output and in/out buffers, which correspond to the read, write, and read/write buffers in the Charm++ Offload API\cite{79}. Gordon transfers data in input and in/out buffers from main memory to the SPE before execution of a job. It transfers data in in/out and output buffers from the SPE to main memory after the execution of a job. Gordon provides special functionality for transferring data that is scattered in main memory to/from a single I/O buffer at the SPE, which adds to the flexibility of Gordon. Gordon internally uses DMA list structures for these transfers. The only limitation on the I/O buffers is that they have to fit in the local memory of an SPE.

For performing the task of isolating concurrent routines, mentioned in Section 5.1.1, Gordon enforces some restrictions on the application that uses Gordon. The application at the PPE should not access the memory buffers assigned to an SPE job while the SPE job has not finished. As an SPE job executes concurrently with the running application, the SPE accesses the buffers while the application is running at the PPE. The buffers are available to the application again when Gordon signals the job is done, for example when it calls the corresponding call-back function. Another restriction on the data buffers is that the application should properly align them in memory. The Cell processor requires that all DMA transfers are aligned at 128 bytes. To aid the user, Gordon provides functions that allocate data buffers with the proper alignment.

5.3.2 Buffer management

Gordon fully performs the task of multi-buffering mentioned in Section 5.1.1. It buffers both input and output data, thereby hiding the overhead of transferring this data. Gordon performs multi-buffering by allowing multiple pending jobs at a single SPE. Gordon can execute jobs out-of-order, as it executes them as soon as their input data is available. Before executing a given job, an SPE fetches the input data for the other pending jobs using asynchronous DMA transfers. Output data is also sent using asynchronous DMA. While executing a job, Gordon thus transfers both the input data for subsequent jobs and the output data from previous jobs.
Multi-buffering works best if the computation time of a job is larger than the communication time for transferring the input and output data. For avoiding saturating the memory bus, the ratio of computation time versus communication time on the SPEs should be larger than the number of SPEs. This ratio can be increased by grouping successive jobs together. An application can avoid communication with main memory by storing the data that is transferred from one sub-job to the next in allocation buffers in local memory.

On the SPE, Gordon allocates memory per buffer type, instead of allocating all necessary memory for a job at once. The transfer of the input data can therefore begin as soon as memory is available for these buffers. Even if memory is not available for the other buffers, Gordon transfers the input data and performs multi-buffering. When Gordon would allocate all memory at once, this optimization is not possible because these transfers are delayed until all required memory is available.

Similarly, Gordon deallocates the input and allocation buffers as soon as the SPE function is complete. This way, it makes memory available for other jobs even if the SPE is still transferring output data. When Gordon would allocate all memory at once, it could only deallocate memory after transferring all output data. With this optimization, however, other jobs will sooner be able to allocate memory and transfer input data.

Although Gordon supports a variety of data transfer types, including scatter/gather primitives, occasionally an SPE function requires extra data transfers. For example, when variable sized data is used, the application does not know in advance how much data the SPE function consumes. However, the Gordon API requires that the application specifies all data sizes in advance in the job description, which is impossible in this case.

The solution is allowing an SPE function to manually schedule extra data transfers and perform its own buffer management. Because Gordon performs all memory management, the job description should specify that the function needs extra memory using an allocation buffer. The function can then use this buffer for its own transfers. The application can also ask Gordon to fill the buffer with initial data, by declaring the buffer as an input buffer instead of an allocation buffer.

This solution shows that Gordon is very flexible regarding data transfers. All common data transfers can be performed using Gordon. When more advanced structures are involved, the SPE function can take over and perform its own data transfer strategy. Of course, an application will no longer benefit from the automatic buffer management provided by Gordon when it takes over control of data transfers. However, an application can still use all other features of Gordon with this approach, including automatic buffer management for the other buffers.

A common case in which the function requires extra data transfers, is when it irregularly accesses main memory. In this case, Gordon can provide the function with a software cache, which caches a certain region in main memory. Different caches can be used for different memory regions. The software cache handles all communication with main memory. It has a simple API for requesting and prefetching data.

However, using a software cache is discouraged, because it a very inefficient
method of accessing main memory. Besides the extra DMA transfers, the cache incurs some management overhead. When the cache hit ratio is low, the function can better avoid cache management overhead by not using the cache. The function then always transfers all data items, whether they are already present in local memory or not.

5.3.3 SPE function library

Each SPE has a function library with the functions that are executed on the SPEs. The developer supplies these functions, as they differ for each application. However, functions may be reused across different applications. Within these functions, some actions that might interfere with Gordon are disallowed, such as memory allocation.

All SPE functions implement the same interface, which allows Gordon to handle all functions equally. Gordon supplies an SPE function with the locations in SPE memory of the buffers the function has to use. The function then reads the input from local memory, performs some operation, and writes the result to local memory. The SPE function developer thus does not have to handle difficulties like synchronization with the PPE and asynchronous DMA because Gordon performs these tasks. As explained in Section 5.3.2, an SPE function can take over control of the data transfers from Gordon if it really wants to.

In our current prototype implementation, the entire SPE function library is continuously present on all SPEs. A developer can only add or remove functions from the library at compile time. Although the SPE local memory, which includes the instruction store, has a limited size of 256 kilobyte, we have not encountered problems due to this limit, since the total SPE code, including all libraries, uses 50 kilobyte. Future implementations of Gordon could remove these restrictions by allowing the application to modify the function library at run time. We expect that the added complexity in Gordon can be fully hidden behind its API. Since the total SPE function library uses 23 kilobyte, and an SPE function uses 3 kilobyte on average, the application could save 20 kilobyte of SPE local memory this way.

Optimizations

Conforming to the task of allowing optimizations, Gordon allows using all SIMD and other code optimizations that are local to the function. An SPE function can benefit from several optimizations, which have a huge impact on performance, as we will show in Section 5.5.1. The various possible optimizations are:

- SIMD optimizations. Using platform-specific compiler intrinsics, the developer instructs the compiler to use specific machine instructions. These intrinsics are used like normal C functions. The arguments to the function specify the input values for the instruction and the output value is written to the return value in the function call. The compiler still performs register allocation and instruction reordering. Using intrinsics is therefore more convenient than writing assembly code.
Figure 5.3: Example SIMD intrinsic: \( c = \text{spu\_mul}(a, b); \)

The SPE has 128-bit registers, which can contain two double precision floating point values, four single precision floating point values, four 32-bit integer values, eight 16-bit integer values, or 16 8-bit integer values. SIMD machine instructions perform a single operation on all values in the register. For example, Figure 5.3 shows the \( \text{spu\_mul} \) intrinsic, which multiplies two vectors with single precision floating point values. Each element of vector \( a \) is multiplied by the corresponding element of vector \( b \), resulting in vector \( c \). The corresponding C code for this intrinsic is \( c = \text{spu\_mul}(a, b); \). \( a \), \( b \), and \( c \) are special vector variables. This intrinsic is mapped onto an \( \text{fm} \) machine instruction that performs the 4 multiplications at once.

- Data access. The SPE can only move 128-bit values (a full register) between local memory and registers. For writing a 32-bit value from a register to local memory, the SPE first loads 128 bits from memory to a register. Then it moves the 32-bit value to the correct position within the register and writes the register back to memory. Using 32-bit values therefore incurs some overhead. For optimal performance, memory accesses have to use 128-bit values where possible. For example, when accessing arrays of 32-bit values, the function should load four elements at once, which saves three memory accesses. When writing to memory, using 128-bit values is even more important, because it avoids the memory load before the write.

- Data alignment. The SPE only accesses its local memory at 16 byte boundaries. When a memory access is not aligned, the SPE reschedules the data into the correct position when reading and writing data. For data that crosses a 16 byte boundary, it performs an extra memory access. The function can avoid this overhead by aligning all data in memory at 16 bytes. If alignment is not possible, the function can improve performance using aligned 128-bit memory accesses and manual data shuffling. Without aligned memory access, the compiler generates less efficient shuffling code as it is unaware of the data semantics.

- Loop unrolling. Non-optimized functions typically perform a single operation in each loop iteration. In each iteration, the function typically loads input from memory, performs some operation, and writes the result back to memory.
By combining multiple loop iterations into a single long iteration, performance may improve considerably for the following reasons:

– The function combines memory accesses, which reduces the total number of memory accesses.
– The remaining memory accesses can often be aligned to 16 bytes.
– The function can use SIMD intrinsics for performing the operations of multiple loop iterations in one go.
– The number of branches is reduced, as the loop performs less iterations.
– The processor pipeline usage improves. When unrolling many iterations into a single iteration, there will be many independent operations in the resulting iteration. The compiler detects the independent operations, as they use different variables, and schedules the machine instructions in a way that the pipeline is optimally used. For optimizing pipeline usage, the developer can use a special tool that annotates SPE assembly code with the pipeline usage of each instruction. Using software pipeling, a developer can further improve performance of loops[10].

5.3.4 Job chaining

Because an SPE has limited local memory, the developer has to split operations into multiple jobs. Gordon facilitates scheduling a series of jobs, where for example each job processes part of the input data, by allowing combining several jobs into a job chain. A job chain is a list of jobs that is scheduled as one entity. An application schedules a chain by simply asking Gordon to execute the first job of the chain. Gordon will automatically execute all subsequent jobs in the chain on the same SPE.

Similarly to executing a single job, the application can specify a call-back function when executing a job chain. When executing a single job, Gordon calls the call-back function when the job has finished. When executing a job chain, Gordon calls the call-back function when the last job of the chain has finished.

Using job chains has various advantages:

• Job ordering. A job chain implicitly specifies an ordering between jobs, as Gordon executes all jobs within a chain sequentially on one SPE. This property is important when there are dependencies between jobs. Without chaining, an application an only send a job that depends on another job to Gordon when the other job has finished. With chaining, Gordon can execute both jobs in one go.

• Memory re-use. When all jobs in a chain use similar input and output buffer sizes, the application can re-use the memory buffers in SPE local memory for these jobs. Memory allocation overhead is reduced this way, because there is no deallocation and allocation between two jobs in a chain.
• Transfer data between jobs. When re-using memory buffers in local SPE memory, the data in the allocation and output buffers remains on the SPE between the execution of two successive jobs. The data is thus persistent and shared between two jobs in a chain. Using persistent data reduces overhead as it avoids two data transfers. Without persistent data, an SPE has to transfer the data to and from main memory between two jobs.

For saving memory traffic, Gordon also allows persistent input and in/out data buffers. For example, when all jobs in a chain have the same arguments, the application can put these arguments into an input buffer that is only transferred with the first job in the chain.

• Less overhead. Besides the above-mentioned possible reduction in memory allocation and data transfer overhead, overhead is reduced because fewer interactions with Gordon occur. Gordon calls the call-back function once for every chain instead of once for every job. Similarly, Gordon has to synchronize the SPEs less often with the PPE. An SPE processes a chain autonomously without synchronizing with the PPE between two jobs. An SPE only sends a notification of completion to the PPE when the last job of a chain has finished.

The overhead reduction is especially important for short jobs, as the overhead of running these jobs is relatively high. Because of the limited SPE memory, jobs process few data and many jobs will only run for a short time.

There are two disadvantages to using job chains. First, all jobs in a chain are executed sequentially on one SPE instead of in parallel on multiple SPEs. When an application uses few chains, load imbalance is likely. Second, because of the memory re-use and persistent data optimizations, multi-buffering is not possible for jobs in a single chain. Gordon does however still perform multi-buffering between the jobs in different chains. Both disadvantages are overcome by executing a sufficient amount of chains. Applications with a small number of jobs need to seek a balance between having few long chains and many short chains. In Section 5.5.4 we will investigate this balance.

Gordon implements job chains using a linked list of job description structures. A job description contains the address of the next list item. Gordon allocates and initializes these lists for the developer. An SPE autonomously processes a job chain using these addresses, as we will show in Section 5.3.6. This approach is efficient as processing a job chain does not require transferring any extra structures.

Gordon implements memory-reuse efficiently using a flag in the job description structure. Because the addresses of job description structures in memory have to be aligned to 16 bytes, the four lower bits of the address of the next list item are always zero. Gordon uses the lower bit as memory-reuse flag. If it is set, the next job reuses the memory of the current job.
5.3. GORDON

5.3.5 Notification approaches

Gordon submits a job to an SPE by sending the 64-bit memory address of a job description structure to the SPE using two 32-bit mailbox messages. In the Cell architecture, each SPE has an inbound mailbox with four entries. An SPE periodically polls this mailbox for new messages. The mailbox is an efficient means of sending messages from the PPE to an SPE, as it only requires two bus transactions. After receiving the address, the SPE transfers all other data using asynchronous DMA.

When a job is complete, the SPE sends a single 32-bit notification message to the PPE. The Cell processor has two special-purpose mechanisms that are intended for transferring these messages, namely interrupts and outbound mailboxes. Gordon can also use the default DMA communication mechanism. Figure 5.4 and Table 5.2 show the three approaches, which we describe below.

In Section 5.5.2 we will show that, although the messages are small, the notification mechanism has a serious impact on performance. Gordon therefore uses DMA by default as it outperforms both special-purpose mechanisms, rendering them useless. For evaluation purposes, we have created modified versions of Gordon that use interrupts and outbound mailboxes instead of DMA.

**Interrupts**

Using a special interrupt outbound mailbox, an SPE can trigger an interrupt at the PPE. The PPE then reads the 32 bit mailbox content. For servicing interrupts, Gordon runs a separate thread at the PPE, which wakes up at every SPE interrupt. This mechanism works similarly to softirqs in the Linux kernel[24], as this thread does not run in strict interrupt mode. Although this approach is relatively straightforward, it has two main disadvantages:

1. Interrupts are costly because the operating system handles them[125]. When an interrupt arrives, the hardware invokes an interrupt handler in the operating system. The operating system needs to make a context switch to the interrupt handling thread, and a context switch back to the original thread.
CHAPTER 5. GORDON: A CELL RUN TIME LIBRARY

<table>
<thead>
<tr>
<th>Approach</th>
<th>PPE task</th>
<th>SPE task</th>
<th>Slots</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupts</td>
<td>Wait for interrupts in separate thread</td>
<td>Use interrupt mailbox</td>
<td>1 slot for all SPEs</td>
<td>Interrupts are costly</td>
</tr>
<tr>
<td>Mailbox</td>
<td>Poll SPE outbound mailbox</td>
<td>Use normal mailbox</td>
<td>1 slot per SPE</td>
<td>Polling requires bus transaction</td>
</tr>
<tr>
<td>DMA</td>
<td>Poll by reading memory</td>
<td>Use DMA</td>
<td>Unlimited</td>
<td>none</td>
</tr>
</tbody>
</table>

**Table 5.2:** Summary of the three notification approaches

2. Interrupt communication does not scale because there is only one interrupt mailbox for the whole Cell processor, which is shared among all SPEs. When one SPE has sent an interrupt message, the other SPEs have to wait until the PPE has processed the interrupt before they can send another interrupt message.

**Mailbox**

Besides the interrupt outbound mailbox, there is one normal outbound mailbox per SPE, which also holds a 32 bit message. When the SPE sends a message to the PPE using this mailbox, the Cell does not generate interrupts. Therefore the PPE needs to poll the mailbox periodically to see if a new message is available. Compared to the interrupt outbound mailbox, the normal outbound mailbox has two different disadvantages:

1. Polling the outbound mailbox for new messages incurs some overhead because it requires a system call to the operating system. Moreover, the mailbox resides on the SPE, and not on the PPE. For each poll, the PPE makes a transaction over the internal element interconnect bus (EIB) for accessing the mailbox at that SPE.

2. There is only one mailbox slot for each SPE. When an SPE wants to acknowledge multiple jobs, it has to wait until the PPE has read the previous message.

The PPE automatically polls all SPEs whenever the application submits a job. Gordon therefore automatically adjusts the polling rate to the job submission rate. The application can also call a function in Gordon that waits for one or more jobs to complete. This function continuously polls the SPEs until the requested number of jobs have completed. The application can also instruct this function to wait until all pending jobs have completed.
DMA

An SPE can use DMA for sending 32 bit messages to the PPE. With this notification approach, Gordon assigns a special memory area to each SPE. An SPE writes job acknowledgment messages to its memory area using DMA. Each memory area contains a fixed number of message slots, which Gordon uses in cyclic order.

With this approach, the PPE polls by simply reading memory. Similarly to NAPI, the PPE can easily poll multiple slots at once, draining all pending notifications in one go. When a poll is unsuccessful, the PPE will usually only access its internal cache. Unsuccessful polls thus usually do not require accessing the internal interconnect bus. By enforcing a maximum number of outstanding jobs per SPE, the PPE ensures that the acknowledgment slot for a certain job is always available. The statically allocated circular buffer is therefore an efficient structure as it provides a lock-free producer-consumer channel and incurs no run-time allocation. Various other systems also use this structure.

Unlike the other approaches, the SPE does not have to wait for the availability of a mailbox slot when acknowledging a job. It only has to schedule an asynchronous DMA transfer. Only when its DMA engine is fully occupied, the SPE has to wait. However, this condition is unlikely to occur, as an SPE DMA engine can queue up to 16 DMA transfers. When the queue has an available slot, the SPE hands the DMA request over to the DMA engine and the SPE immediately continues processing other jobs.

Similarly to the mailbox approach, Gordon automatically adjusts the polling rate to the job submission rate by polling all notification buffers whenever the application submits a job. Again, the application can use the waiting function to wait for completion of jobs and to make Gordon poll continuously.

5.3.6 Implementation

The PPE code within Gordon performs the tasks of initializing the SPEs, load balancing the SPEs and synchronization with the SPEs. It also maintains the administration of the call-back functions. The code that runs on the SPEs consists of the SPE function library, which we explained in Section 5.3.3 and the Gordon SPE run time.

Gordon performs load balancing by storing the same binary code on all SPEs. Gordon can then schedule each job on any available SPE, which allows dynamic load balancing of the SPEs. When all SPEs are busy, Gordon stores new jobs in an internal queue. Gordon immediately sends a job from this queue to an SPE as soon as the SPE notifies a job is complete.

The SPE run time performs various tasks on its SPE, including initialization, synchronization with the PPE, fetching job descriptions, allocating memory, fetching input and in/out buffers, starting functions from the SPE function library, and sending in/out and output buffers. The SPE run time supports multiple jobs at a single SPE.
The SPE run time includes an automaton that processes jobs using multiple stages, as shown in Figure 5.5. When the run time cannot advance the automaton for a job, for example when no memory is available or when DMA transfers are not finished, it tries advancing the automaton of another pending job. Gordon uses the SPEs very effectively this way: An SPE is only waiting when it cannot make progress on all pending jobs, for example when all jobs are waiting for DMA transfers. In all other cases, the SPE will make progress on a job and advance the corresponding automaton. When an SPE is waiting, it continuously checks all automata. When one of the automata can be advanced, for example when a DMA transfer has finished, an SPE quickly notices it and the SPE will almost immediately make progress on the corresponding job.

The automaton has a reuse memory flag that indicates if the previous or next job in a job chain uses the same local memory buffers at the SPE. An SPE sets it using the reuse memory flag in the job description, which we explained in Section 5.3.4. Setting this flag in the automaton happens after a job has run, as will be explained below near the 'Run' state. When this flag is set, the automaton processes the job faster because many states are skipped. By default, the flag is clear. The various states in the automaton are:

- **Idle.** This state is the initial state of the automaton. It means that there is no active job in this slot. In this state, an SPE polls the inbound mailbox for new mail. When it detects new mail, it will store the message, which is a pointer to a job description structure in main memory, and advance the automaton to the "Fetch job" state. When the poll is not successful, the SPE switches to the automaton of another job.

- **Fetch job.** In this state the SPE transfers the job description from main memory to its local memory using DMA. When the job description has arrived, it will advance the automaton to the next state, which depends on the reuse memory flag.

- **Allocate allocation, input, in/out, or output data buffers.** In these states an SPE tries to allocate memory for the specified data buffer type. Only one job may allocate memory at a time, which prevents deadlock. Another job can only allocate memory once the 'allocating' job has allocated all necessary
memory. This other job then automatically becomes the 'allocating' job. When enough memory is available and the job is the 'allocating' job, Gordon allocates memory for the job and advances the automaton to the next state.

- Fetch input or in/out. The SPE checks if its DMA engine has an available slot for a DMA transfer. When there is an available slot, the SPE starts asynchronous DMA transfers of the input or in/out buffers from main memory to its local memory. It does not wait until the transfer is complete and advances the automaton to the next state, which depends on the reuse memory flag.

- Wait for input. The SPE asks its DMA engine if it has transferred all input data, and advances the automaton to the Run state if it has.

- Run. The SPE has allocated all memory in this state. It has also received all input data, which means the job is ready to run. The SPE invokes a function from the SPE function library and supplies pointers to the various buffers as parameters. If the job is part of a job chain, and the next job in the chain reuses the memory of this job, it sets the reuse flag to skip several states in the next iteration of the automaton. Finally, the automaton advances to the next state, which depends on the reuse memory flag.

- Free input and allocation. The SPE frees the allocated allocation and input buffers for the job, as they are no longer needed, and advances to the "Send in/out and output" state.

- Send in/out and output. The SPE checks if its DMA engine has an available slot for a DMA transfer and transfers the in/out and output buffers from local memory to main memory using asynchronous DMA. It does not wait until the transfer is complete. If the reuse memory flag is set, it implies that there is a next job in the job chain. The SPE extracts a pointer to the next job from the job description, and continues fetching the new job. The SPE can safely skip the "Wait for output" state: When the SPE waits for the new job description in the "Fetch job" state, it automatically also waits for the transfer of the output.

- Wait for output. The SPE asks its DMA engine if it has transferred all output data, and advances the automaton to the next state if it has.

- Free in/out and output. The SPE frees the allocated in/out and output buffers for the job, as they are no longer needed. If a job is part of a job chain and it is not the last job, the SPE extracts the address of the next job description structure in main memory from the current job description and the automaton advances to the "Fetch job" state. Otherwise, the automaton advances to the "Send ACK" state.

- Send ACK. The SPE sends an asynchronous message to the PPE indicating that it has completely processed the job or job chain. The SPE does not have to wait for an acknowledgement from the PPE and always advances the automaton to the Idle state.
CHAPTER 5. GORDON: A CELL RUN TIME LIBRARY

5.4 SP@CE applications on the Cell processor

In Chapters 3 and 4 we described how the SP@CE framework supports streaming applications on homogeneous platforms using the Hinch run time system and the XSPCL coordination language. Hinch and XSPCL abstract the user from the difficulties of developing these applications by allowing the developer to specify a streaming application as a data flow graph of interacting components. Both task- and data-parallelism are easily expressed and advanced features such as dynamic reconfiguration are fully supported. In this section we will show that SP@CE is also suitable for heterogeneous MPSoC platforms. Mapping applications on these platforms is difficult for several reasons:

- The developer needs to decide which resource each application task uses. As this decision is not always obvious, the developer needs to implement multiple versions of tasks and evaluate which combination performs best.
- Performing this evaluation requires that the different task implementations are easily interchangeable. Each implementation must therefore adhere to a standard interface, which is difficult because the implementations use different resources.
- For optimal performance, a task should allow other tasks to use its resources when it is idle. The application thus has to share resources between multiple tasks.
- Different communication mechanisms exist between the various resource types. For achieving optimal performance, communication needs to be implemented differently for every combination of resource types.

In this section, we will show how we support streaming applications for heterogeneous MPSoCs, by integrating the Gordon run time library into the SP@CE framework. A SP@CE component developer can choose between using the main processor or one or more SPEs for the computations in the component. Gordon automatically assigns SPEs to the component as soon as they are available. When the component is idle, the SPEs are available for other components.

5.4.1 Gordon components

Special Gordon components form the glue between SP@CE and Gordon. Towards SP@CE, they act like normal components. At the PPE, they adhere to the standard component interface, as described in Section 3.3.1 and use the primitives provided by the Hinch run time system for streaming and event communication. Internally, they use Gordon for offloading computations to the SPEs. Figure 5.6 illustrates how a Gordon component interacts with both SP@CE and Gordon, by showing the steps that occur when a Gordon component runs:
5.4. SP@CE APPLICATIONS ON THE CELL PROCESSOR

1. The Hinch data flow scheduler schedules the Gordon component by calling its
run function, as we described in Section 3.3.2.

2. The Gordon component reads the data from its input streams using the Hinch
streaming communication interface.

3. Using Gordon, the Gordon component schedules jobs on the SPEs that process
this data.

4. Gordon invokes a call-back function, specified by the Gordon component, when
the jobs are complete.

5. The Gordon component writes the output data from the SPEs to its output
streams.

6. The Gordon component notifies Hinch it has finished executing and thereby
allows Hinch to schedule the successors of the Gordon component in the data
flow graph that Hinch manages.

A Gordon component transfers the data from the Hinch streams to Gordon using
a zero-copy protocol, as shown in Figure 5.7. When a component reads or writes a
stream, Hinch supplies the component with the address in memory of a data buffer in
the stream. The component in turn supplies Gordon with this address by entering it
in the job description structure, as explained in Section 5.3.1. Gordon then submits
the addresses to the SPEs. At the PPE, the component and Gordon do not perform
any data copying. They only copy addresses. The SPE that executes the job still
transfers the data to its local memory using DMA as it can not directly access main
memory.

For achieving optimal performance, running a Gordon component may differ
from running a normal component that only uses the PPE. A normal component
always returns control to Hinch after it has finished running. This approach is
not always optimal for Gordon components. Table 5.3 summarizes the alternatives,
the necessary run time system actions, and the drawbacks of various approaches of
running Gordon components. The alternatives are:
CHAPTER 5. GORDON: A CELL RUN TIME LIBRARY

<table>
<thead>
<tr>
<th>Approach</th>
<th>RTS actions</th>
<th>Drawbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait for SPE jobs</td>
<td>none</td>
<td>Low resource utilization</td>
</tr>
<tr>
<td>Use many threads</td>
<td>Start more threads</td>
<td>Overhead; Thread count depends on application</td>
</tr>
<tr>
<td>Asynchronous components</td>
<td>Special support</td>
<td>More complex RTS</td>
</tr>
</tbody>
</table>

Table 5.3: Alternatives for running a Gordon component

- A Gordon component waits until the jobs at the SPEs are complete, and then returns control to Hinch. This approach is similar to running normal components and does not require any special treatment by the Hinch run time system that schedules all components. However, this approach wastes many resources, for three reasons:
  - PPE resources are wasted as the PPE could do useful work instead of waiting for the SPEs.
  - The application only uses the SPEs when a Gordon component is running, and not when other components are running.
  - Since only one Gordon component can be active, it is unlikely that the SPEs will be optimally used. Load imbalance is very likely as a Gordon component schedules a limited number of jobs on the SPEs.

- Use more threads than the architecture supports on the PPE. While a Gordon component is waiting, other threads run other components on the same processor. These other components can be other Gordon components. The SPE usage will thus improve as multiple Gordon components can be active simultaneously. Implementing this alternative is easy: The Hinch run time system simply has to start more threads. However, this approach has two major problems:
  - Using more threads than the architecture supports causes context switching and thread management overhead.
  - Determining the optimal number of PPE threads is difficult. Applications that are dominated by Gordon components should have a high number of threads. With few threads, it is likely that all threads are executing Gordon components that wait for completion of SPE jobs. The PPE will then still be idle. Applications with relatively few Gordon components should have fewer threads to avoid context switching and management overhead.

- Run Gordon components asynchronously. When a Gordon component has submitted its jobs to Gordon, it does not wait until Gordon has finished running these jobs. Instead, it returns control to Hinch, without informing Hinch that
it has finished executing. Hinch can then run other components, including other Gordon components. Gordon notifies the component when the jobs are complete, using a call-back function supplied by the component. In this call-back function, the component informs Hinch that the component has finished executing. Hinch then schedules the successors of the Gordon component in the data flow graph.

The SPE jobs submitted by the Gordon component thus run asynchronously to Hinch without requiring an extra thread of control at the PPE. The call-back by Gordon also does not require an extra thread as Gordon only polls for completion of jobs when a new job is submitted, or when the main processor is idle.

The drawback of using asynchronous components is that it requires special support in Hinch, which entails the following tasks:

- Hinch must detect that a Gordon component has not finished running when it returns control to Hinch.
- When Gordon calls the call back function, the Gordon component needs to inform Hinch that an iteration of the component is complete. Hinch must supply functionality to the component to perform this operation.
- Hinch must handle multiple finishing modes of components, which adds to the complexity of Hinch. Normal components finish their iteration at the end of their run function, while Gordon components finish their iteration when they inform Hinch.
- When the PPE is idle, this condition normally indicates that all work has been done and Hinch should terminate the application. With asynchronous Gordon components, a Gordon component could still be active. Hinch must detect this situation, and call the Gordon waiting function that polls the SPEs in this situation.

Although using asynchronous components adds to the complexity of the Hinch run time system, we have chosen it instead of the other two approaches because it provides important advantages:

- There is no context switching or thread management overhead as this approach requires only a single thread.
- The number of active Gordon components is unlimited. Unlike the multi-threading approach, it is not limited by the number of threads, nor is it limited by Gordon. Gordon will always accept new jobs and maintains an internal queue of pending jobs. Gordon puts jobs at this queue when all SPEs are busy.
- The PPE and the SPEs are optimally used. The PPE is always available for normal components as Gordon components do not wait. The SPEs are optimally used as Gordon receives all jobs as soon as Hinch schedules the corresponding Gordon component.
5.5 Experiments

We evaluate Gordon by performing experiments on a Cell blade with two first-generation Cell processors. Each Cell processor has eight SPEs available for user applications. For development and demonstration purposes, Gordon also runs on a PlayStation 3 console. This console contains a single Cell processor that has six SPEs available for user applications.

Both the PPE and the SPEs run at 3.2 GHz. All measurements use the built-in decremter register, which ticks at a fixed frequency. The experiments are repeated 11 times. Then we took the average of these 11 measurements. Unless specified otherwise, the standard deviation within a set of 11 repeated measurements is always less than 8.1% of the average measurement. In the measurements with a single SPE, the standard deviation is always less than 2.1%.

We measure the overhead of using Gordon using a synthetic benchmark function in the SPE function library, which executes a fixed number of cycles on the SPE when Gordon invokes it. The total number of cycles spent in all invocations of this function on all SPEs is divided by the number of SPEs, which results in the average effective execution time (AEET).

On the PPE, we measure the total execution time (TET) on the application. The measurement starts just before the application submits the first job to Gordon and ends when all jobs have finished. It does not include initialization, e.g., set up cost, and finalization, e.g., shutdown cost and printing the result of the measurement.

The relative and absolute overhead of using Gordon are derived from the AEET and the TET. The absolute overhead is the TET minus the AEET, which is zero in the optimal case. The relative overhead is the absolute overhead divided by the AEET, and is given as a percentage. These overhead figures thus include the cost of DMA transfers and the overhead of Gordon. Gordon has overhead both on the PPE side in the interaction with the application and the SPEs, and on the SPE side in the automaton that runs on the SPEs.

5.5.1 SPE functions

We determine typical computation to communication ratios for SPE jobs and the impact of SIMD and other optimizations by analyzing several SPE functions that are used in multiple applications. We run these benchmarks by creating a stand-alone SPE program that runs the specified function and measures the number of cycles the function uses.

All functions in the SPE function library except one perform image processing. The input and output pixels for these functions have one byte per color component. We optimized all functions using the optimizations mentioned in Section 5.3.3. We will now explain the functions and their optimizations.

- Two down scaler functions reduce the size of an image by calculating the average of blocks of pixels. The optimized 4x4 version exploits the apu_sumb
5.5. EXPERIMENTS

...intrinsic, which performs eight sums of four bytes at once. In each loop iteration it processes 256 input bytes, which results in 16 output bytes. The optimized 3x3 version uses spu.sumb for performing eight sums of three bytes in one instruction. It processes 144 input bytes in one loop iteration, which also results in 16 output bytes.

- Yuv2rgb converts color pixel data in YUV format to RGB format. It converts 16 pixels at once.

- IDCT performs an Inverse Discrete Cosine Transform. Its input consists of 8x8 blocks of 32-bit integers. It processes two of these blocks at once, which results in 128 pixels of one byte.

- A convolution function applies a 5x1 or non-separated 3x3 Gaussian blurring kernel to the input image. The input image for the 5x1 kernel contains a border of 2 pixels on the left and right sides. With the 3x3 kernel, the input image has a border of 1 pixel at all four sides. These borders are not present in the output image. The function applies the convolution matrix to 64 pixels in one iteration.

- A geometric transformer applies an affine transformation to the input image. Its input is a block of pixels from the input image. Its output is the corresponding block of pixels in the output image. Its arguments include the transformation matrix, and the position of both the input and output block in the full images.

- A correlator function correlates data from radio telescopes. This task is one of the main signal analysis tasks in the LOFAR radio telescope[123].

Code optimizations

We investigate the impact of SIMD and other code optimizations mentioned in Section 5.3.3 by comparing a slow, non-optimized version of several functions to a fast, optimized version. Both versions have the same signature and functionality, but differ in speed and size.

Table 5.4 shows the complexity in terms of lines of C code and the performance of both versions. We measure the lines of C code by copying the relevant functions to separate files, and using the SLOCCount utility[126] on these files. The lines of C code factor is the ratio between the lines of C code in the both versions. It indicates the relative complexity of optimizing the function. The two convolution functions have the same complexity, because we use the same function with different arguments. In the average lines of code factor, the convolution function therefore only counts once.

The lines of code difference shows that the fast versions are considerably more complex than the slow version. While developing a fast version, a developer has to match the SIMD register structure with the data layout in memory. This matching...
increases complexity as it often requires explicit data-reordering statements, which are absent in the slow version.

The performance improvement of the fast versions is considerable. The minimum speedup is already one order of magnitude, while the maximum speedup reaches two orders of magnitude. With these improvements, the bottleneck in the application could shift from computation to communication. We conclude that the cost of added complexity is well worth the performance benefit.

**Computation to Communication ratio**

For evaluating the computation to communication ratio of SPE functions, we use the functions described above. We set the total input and output data size to about one quarter of the total SPE local memory. We cannot use all local memory for input and output data since the function code, stack, and global data also use local memory. When SPE functions use one quarter of the local memory, multiple jobs on a single SPE can allocate memory and Gordon performs multi-buffering between these jobs. When a single job uses all available memory, multi-buffering between different jobs is not possible.

Table 5.5 lists the total number of bytes transferred for each function along with a measured decremeneter tick count. From these figures, we compute the number of cycles per transferred byte. This value differs by two orders of magnitude, which is very significant. We can distinguish two groups of functions:

- One group performs hardly any computation for each transferred byte. This group consists of the Downscale, yuv2rgb, Maximum, IDCT 8x8 and Correlator functions. The number of cycles per transferred byte is 1.51 or less. In an application, the performance of these functions depends mainly on communication performance, and not on the amount of computation. Hiding communication latency behind computation cycles using multi-buffering is difficult
5.5. EXPERIMENTS

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Bytes in/out</th>
<th>Ticks</th>
<th>Cycles/byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downscale 3x3</td>
<td>64808</td>
<td>112</td>
<td>0.39</td>
</tr>
<tr>
<td>Downscale 4x4</td>
<td>65288</td>
<td>45</td>
<td>0.15</td>
</tr>
<tr>
<td>yuv2rgb</td>
<td>61444</td>
<td>207</td>
<td>0.75</td>
</tr>
<tr>
<td>Maximum</td>
<td>61952</td>
<td>78</td>
<td>0.28</td>
</tr>
<tr>
<td>IDCT 8x8</td>
<td>51208</td>
<td>345</td>
<td>1.51</td>
</tr>
<tr>
<td>5x1 convolution</td>
<td>66736</td>
<td>2165</td>
<td>7.25</td>
</tr>
<tr>
<td>3x3 convolution</td>
<td>64234</td>
<td>2448</td>
<td>8.52</td>
</tr>
<tr>
<td>Geom. transform</td>
<td>51248</td>
<td>4532</td>
<td>19.76</td>
</tr>
<tr>
<td>Correlator</td>
<td>73264</td>
<td>212</td>
<td>0.65</td>
</tr>
</tbody>
</table>

Table 5.5: Computation to communication ratios for various kernels

with these functions, since there are hardly any computation cycles. In our following experiments, we will represent this group using measurements with 1 cycle per transferred byte.

- One group performs a moderate amount of computation for each transferred byte. This group consists of the Convolution and Geometric transform functions. The number of cycles per transferred byte varies between 7 and 20, which allows hiding communication latency behind computations. The bottleneck of the performance of an application thus mainly lies in the computations themselves, and not in communication. However, when using many SPEs, communication may still become a bottleneck. In our following experiments, we will represent this group using measurements with 10 cycles per transferred byte.

5.5.2 Notification approaches

We evaluate the different notification approaches modes we described in Section 5.3.5 using a synthetic benchmark application. We run the benchmark for all three approaches using 16384 separate SPE jobs with 32720 input bytes and 32720 output bytes. Due to memory constraints, the application uses only 1024 input and output buffers, which the application cyclically distributes over the jobs. The synthetic benchmark application uses an SPE function that waits until a certain number of cycles has passed. We use 'small' jobs with 65540 cycles and 'large' jobs with 655400 cycles, which leads to 1 or 10 cycles per transferred byte, respectively. These ratios comply with the results from the previous section.

Figure 5.8 shows the results of these benchmarks using 1 to 16 SPEs. Note the logarithmic scale of the y axis. The standard deviation in the 11 repeated experiments is high with the interrupt mode using large jobs. It varies between 0.2 % and 30.6 % of the average measurement, which shows that interrupts do not provide stable performance. Figure 5.8 shows three other important results:
1. There is a big overhead difference for jobs with different cycles per byte. An application can significantly reduce overhead by increasing the number of cycles per byte, for example, by combining multiple operations that would otherwise be separate jobs.

2. The overhead increases with the number of SPEs, because there is more contention within the processor. Also, load imbalance is more likely with more SPEs.

3. There are considerable differences between the various acknowledgment modes. The DMA mode always outperforms the mailbox mode, which in turn outperforms the interrupt mode. We conclude that the special interrupt and mailbox communication primitives in the Cell do not provide any added value over the default DMA communication primitive with this benchmark. We have therefore chosen DMA as the notification mechanism within Gordon. In our next experiments, we will only use DMA.

5.5.3 DMA overhead and multi-buffering

The overhead of running a job on an SPE using Gordon consists of the SPE management performed by Gordon, which includes synchronization between the PPE and the SPEs, and the cost of data transfers to and from the SPEs using DMA. We distinguish both sources of overhead by running SPE jobs with and without DMA transfers. We use a single SPE for these experiments, at which we vary the amount
5.5. EXPERIMENTS

Figure 5.9: Gordon overhead for varying DMA transfer sizes, using one SPE and 1024 jobs of input and output data for each job. We also evaluate the effect of multi-buffering by varying the maximum amount of pending jobs per SPE.

Note that besides the input and output data, Gordon also transfers the SPE job description to the SPE using DMA. The notification mechanism also uses DMA. Since these DMA transfers are necessary for each job, we consider these DMA transfers as part of the SPE management overhead, instead of the DMA overhead.

Figure 5.9 shows the total overhead of Gordon, including the cost of all DMA transfers. We use small and large SPE jobs, which run 65536 and 655360 cycles, respectively. The application sends 1024 jobs to Gordon in one go, which allows Gordon to exploit as many DMA latency hiding techniques as possible.

We vary the DMA transfer size from 0 bytes to 65536 bytes. These bytes are divided evenly over one input and one output buffer. We increase the transfer size in steps of 4096. All DMA transfers thus transfer multiples of 128 bytes. We also align all buffers at 128 bytes. These transfer and alignment sizes yield optimal performance on the Cell architecture. Figure 5.9 shows four important results:

1. For small jobs, the overhead varies between 7.5 % and 123 % of the total amount of compute cycles in the SPE jobs. For large jobs, the overhead varies between 0.75 % and 12.5 %. Gordon is thus able to run SPE jobs at a relatively low overhead, when the jobs are big enough.

2. Figures 5.9a and 5.9b are similar. We conclude that the absolute overhead does not depend on the complexity of the SPE jobs, in terms of compute cycles.

3. When the number of jobs per SPE is one, Gordon cannot hide DMA latency.
behind the execution of another job, since there is no other job. The overhead is therefore relative high. With two or more jobs per SPE, Gordon exploits multi-buffering and the overhead drops significantly.

4. The overhead of Gordon increases with the DMA transfer size. When Gordon could hide all DMA transfer latency behind the computations of the SPE jobs, the overhead of Gordon would be constant. However, even with large jobs, behind which DMA transfers could theoretically be fully hidden, the overhead increases with the DMA transfer size. Therefore we conclude that Gordon does not hide all DMA latencies.

5.5.4 Job chaining

For evaluating the effect of chaining multiple jobs together we keep the total number of jobs constant and we vary the number of chains and jobs per chain. This way, we investigate the balance between having few long chains and many short chains, as mentioned in Section 5.3.4. We run the experiment using a varying number of SPEs. The total number of jobs in all job chains is 1024 times the number of SPEs.

For each job, we transfer 32768 input and 32768 output bytes. We run both small and large jobs that run 1 and 10 cycles per transferred byte, respectively. These settings comply with the results of Section 5.5.1.

Figure 5.10 shows the results. With small jobs, using 512 chains of 2 jobs instead of 1024 chains with 1 job reduces overhead by 22 to 60 percent. At 1, 2, and 4 SPEs, the overhead keeps dropping as we increase the number of jobs per chain. At 8, 12, and 16 SPEs, the overhead remains constant. With large jobs, the overhead drops significantly as we increase the number of jobs per chain, regardless of the number of SPEs. We conclude that job chaining effectively reduces the overhead of Gordon.
5.5. EXPERIMENTS

5.5.5 Persistent data

As described in Section 5.3.4, an application can reduce overhead using persistent data. We evaluate this effect by performing experiments in which we store part of the data in a persistent buffer. The application only transfers this data to the SPE with the first job in a chain and not with successive jobs. This experiment resembles the common situation that all jobs in a chain need the same arguments.

The total data size stays constant at 64 kilobytes, which is the same as in our earlier experiments. When adding a persistent input buffer, we divide the remaining bytes evenly over an input and output buffer. For example, when the persistent input buffer holds 16 kilobytes, the normal input and output buffers hold 24 kilobytes each.

Figure 5.11 shows the results of putting 12.5 % to 50 % of the data in a persistent buffer. The application runs using 1 to 16 SPEs with 128 chains for each SPE in the experiment. Each chain contains 8 jobs. Similarly to our earlier experiments, we use both small and large jobs that run 65536 and 655360 cycles, respectively.

Using persistent data effectively reduces overhead. In all cases, the overhead reduction drops linearly as the amount of persistent data increases. With small jobs, the overhead increases with the number of SPEs, because the contention on the bus increases. Large jobs have relatively less contention, since the application transfers less bytes per time unit. The overhead using multiple SPEs therefore remains equal.

With large jobs, there is however an increase in overhead between 8 and 12 SPEs, which is caused by the usage of the second Cell processor. The PPE part application runs on one of the Cell processors. Beyond 8 SPEs, the application uses SPEs in the other Cell processor, since each Cell processor has 8 SPEs. Accessing these off-chip SPEs has some overhead over using on-chip SPEs.
5.6 Summary

This chapter presents Gordon, a run time library for using the Cell processor. Although this processor is very powerful, its heterogeneous MPSoC architecture makes exploiting this power difficult. We identified the various difficulties of programming the Cell and derived the set of tasks required to address these difficulties. We split these tasks into application specific developer tasks and tasks that are generically applicable across many applications.

Gordon performs most of the generic tasks by providing an API that is based on offloading jobs to the coprocessors (SPEs) in the Cell processor. These tasks include dynamic load balancing of the SPEs, multi-buffering, memory management, communication, and synchronization, amongst others. Gordon processes multiple pending jobs at a single SPE, using a finite state automaton for each job. Experiments on the first generation Cell processor show that Gordon has low overhead.

Within Gordon, the SPEs sends short messages to the main processor (PPE) when a job is complete. The Cell processor offers two special mechanisms for sending these notifications, besides the default DMA communication mechanism. We have analyzed the three techniques and concluded that DMA always performs best. Both special-purpose mechanisms thus do not have any added value over the default mechanism when using Gordon.

A developer may combine several SPE jobs in a job chain, which Gordon schedules as one entity. Job chaining provides various advantages such as job ordering, memory reuse, and data transfer between tasks (persistent data). Experiments showed that job chaining also effectively reduces overhead, as it requires fewer synchronizations and data transfers.

Support for Gordon is integrated into the SP@CE framework using Gordon components, that translate the streaming semantics of SP@CE to jobs for Gordon. The Hinch run time system in SP@CE asynchronously runs Gordon components, which increases resource usage at the cost of added complexity.
Chapter 6

Applications and Evaluation

This chapter shows the effectiveness of the SP@CE framework using several challenging streaming applications. The applications perform several complex operations on video streams or other streaming data. We will first describe the applications and their structure in Section 6.1. Then we evaluate their performance in Section 6.2. For heterogeneous applications, we also evaluate implementation alternatives. We have described the SP@CE framework in Section 1.5.

6.1 Application descriptions

We have developed a range of challenging streaming applications which use various different components. All applications are written in XSPCL. All components and SPE library functions are written in C. The components conform to the Hinch component interface and interact with Hinch using its API, as described in Chapter 3. The applications vary considerably, which shows that we support a wide range of applications. For some applications, we have implemented multiple versions, for example a version that uses Gordon components, which only runs on the Cell architecture, and a generic version, which runs on any architecture.

Many applications use the advanced support for dynamic reconfigurability in Hinch and XSPCL. These applications can switch between different modes of operation while they are running. Multiple application versions are often implemented using reconfigurability, by wrapping the application in a manager structure and creating an optional part for each version, as explained in Section 4.2.7. The user can then switch between different versions while the application is running.

We use both data flow graphs and streaming graphs for visualizing the structure of our applications. Figure 6.1 gives an example of both. Data flow graphs show the data flow network in the application using the grouping structure and the dependencies within each group. A streaming graph shows the communication streams between the components. In both graph types, circles and ellipses indicate the components that perform the computations in the application. Normal compo-
components, which use the main processor, have a white background. Gordon components, which use the SPEs of the Cell processor, have a grey background.

In a data flow graph, the arrows indicate data flow dependencies between components, whereas they indicate communication streams in a stream graph. Although the dependencies are derived from the communication streams, there is typically no one-to-one correspondence between both. For example, Figure 6.1(a) shows a communication stream between component A and C, whereas there is no direct dependency between A and C in Figure 6.1a. The dependency is not needed because C already indirectly depends on A via B. For avoiding confusion, we use different kinds of arrows for dependencies and communication streams.

Besides basic components, a data flow graph contains grouping components, which we display using rounded boxes. Grouping components contain other components, as described in Section 3.2.1 We specify the type of the grouping component in its upper left corner. The open and closed circles at the border of a grouping component indicate the entry and exit point of the group, respectively. When all dependencies towards the entry point are satisfied, the group satisfies the dependencies originating at the entry point. Similarly, it satisfies the dependencies at its exit point when all dependencies towards the exit point are satisfied. For clarity purposes, we will often omit the top-level sequential grouping component. The dependency arrows between the top-level components already clearly indicate sequential dependencies.

Table 6.1 lists the applications and their differences. The component count does not include grouping components and the components XSPCL adds for internal use. We count data-parallel components as a single component. The task parallelism column shows if the application exploits task parallelism within a single iteration. All applications exploit task parallelism across iterations, by running multiple concurrent iterations.

We will now describe our applications in turn. In Section 6.2 we will describe common settings for running the applications, and evaluate their performance.
### 6.1. Application Descriptions

<table>
<thead>
<tr>
<th>Application</th>
<th>Architecture</th>
<th>Number of components</th>
<th>Parallelism</th>
<th>Reconfigurable</th>
</tr>
</thead>
<tbody>
<tr>
<td>PiP</td>
<td>All</td>
<td>12</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Filter</td>
<td>All</td>
<td>4</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Rotator</td>
<td>Cell</td>
<td>74</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Correlator</td>
<td>Cell</td>
<td>513</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Edge-Rot</td>
<td>Cell</td>
<td>870</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Edge-2D</td>
<td>Cell</td>
<td>654</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>JMD-Generic</td>
<td>All</td>
<td>177</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>JMD-Cell-IDCT</td>
<td>Cell</td>
<td>177</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>JMD-Cell-JPEG</td>
<td>Cell</td>
<td>129</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>JMD-Cell-Combine</td>
<td>Cell</td>
<td>81</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Application list

#### 6.1.1 Picture-in-Picture

The picture-in-picture (PiP) application combines multiple video input streams into a single video output stream. Figure 6.2 shows the structure of this application, with two pictures-in-picture. The main input component produces background images, which the application displays unmodified. The overlay input components produce picture-in-picture streams. The application scales the images in these streams down by a factor of three in both x and y directions. Then it blends the down scaled images into the background image, resulting in a picture-in-picture display.

The input components read uncompressed video files in YUV format with 4:2:2 color subsampling. The resolution of the images is 720x576. The files contain luminance (Y) and packed chrominance (UV) data. The Y and combined UV data are equally sized due to the color subsampling.

For each picture-in-picture stream, the application contains a down scale component and a blender component, which both support data parallelism. The downscale component splits the input image into blocks of 3x3 pixels. It takes the average of the pixel values in each block and writes the resulting pixel value to its output stream.

The blender component reads the stream from the down scale component and the background image stream and merges them using memory copies. For reducing memory usage and the number of memory copies, the blender does not create a new output stream. Instead, it modifies the images inside the background image stream directly. When it is finished, the next blender component accesses the same background image stream and blends the next picture-in-picture onto the modified background, and so on.

PiP exploits task parallelism by processing components in a pipeline. It also processes multiple pictures-in-picture, and the color fields in the images concurrently. The application exploits data parallelism using slicing for the down scaler.
CHAPTER 6. APPLICATIONS AND EVALUATION

Figure 6.2: Structure of the PiP application, with two pictures-in-picture

and blender components. By default, the number of slices for these components is 16. When we run the application on the Niagara architecture, which supports 64 threads, we increase the number of slices for the down scale components to 60, since these components are the most compute intensive part of the application.

6.1.2 Filter

The Filter application blurs a stream of input images using a two dimensional zero-derivative Gaussian convolution kernel, which is separated into a horizontal and vertical phase. This application is the XSPCL version of the Filter-ID application, which we used in Sections 3.4 and 4.4.

Figure 6.3 shows the structure of the application. The input component is the same as in the PiP application: It converts a 720x576 YUV input file to a Y stream
6.1. APPLICATION DESCRIPTIONS

[Diagram of Filter application]

(a) Data flow graph
(b) Communication streams

Figure 6.3: Structure of the Filter application

[Diagram of Rotator application]

(a) Data flow graph (full application)
(b) Communication streams (one iteration)

Figure 6.4: Structure of the Rotator application

and an UV stream of equal size. The horizontal and vertical convolution kernels run in parallel using cross dependencies, as explained in Section 4.2.4. The level of parallelism in the cross dependencies group is 16 by default, which means both the horizontal and the vertical component have 16 replicas. Similarly to the PiP application, we increase the level of parallelism on the Niagara architecture. The level of parallelism is 64 on this architecture.

6.1.3 Rotator

The Rotator application rotates input images over different angles and writes the rotated images to the output. Figure 6.4 shows the structure of the application. The core of the application is an alternator grouping component that runs a different rotation component in each iteration. Since the data flow graph of the Rotator application does not contain parallelism, the application only exploits parallelism by running multiple iterations concurrently.

Unlike other grouping components, the alternator grouping component actively manages the communication streams to and from its children. It internally has streaming connections to each child. In each iteration, the alternator forwards a
data buffer from its input stream towards the child that runs in that iteration. When the child has finished, it forwards a data buffer from the child’s output stream to its own output stream.

The rotation component is a generic geometric transformation component, which performs these transformations on the SPEs using Gordon. It accepts any transformation matrix.

The rotation component determines the size of its output images from the size of its input images and the transformation matrix, in a way that the output contains the full input image. The component uses a fixed value for the output pixels that are ‘outside’ the input image. The output images for the various rotation angles have different sizes, since the rotation component does not scale the image. We therefore use a resize component after each rotation component, that resizes the rotated images to a fixed size. This way, all iterations within the alternator group output images of equal size and the application combines the output of all iterations into a single stream.

Since full images do not fit into the local memory of an SPE, the rotation component divides the image into blocks. For each block of output pixels, it determines the corresponding input image block using the transformation matrix. The component creates a single job chain of SPE jobs, in which each job processes one image block.

### 6.1.4 Correlator

The correlator application correlates data from different radio telescopes of the LOFAR project\[123\]. The input data and output data consist of complex floating point values. The application simulates the computations for 256 independent channels.

Figure 6.5 shows the structure of the application. The streaming connections follow the data flow dependencies in this application. Because the computations for each channel are independent, the application is trivially parallel. The correlator performs its computations at the SPEs in a Cell processor using a Gordon component. The application only runs on this architecture.

We simulate a continuous input stream using an input component that fills the stream with equal non-zero values. We reuse the same simulated input stream for all channel processing components. In reality, the LOFAR radio telescopes generate one big stream, containing all channels. This stream is then split up into different streams for each channel.
6.1. APPLICATION DESCRIPTIONS

The correlator component splits the data for each channel into different SPE jobs, which it connects into a job chain. The component uses a persistent input buffer for samples that are used by multiple consecutive jobs. It uses an in/out buffer for the correlation results, since each SPE job updates existing correlation results. Before running the SPE jobs, the PPE part of the component initializes the correlation output buffer in main memory with zeroes.

6.1.5 Edge Detector

The edge detector application detects the edges in an image using the luminance color component. It is based on line detection applications found in [107], which use second order Gaussian derivative convolution kernels. In our application, we use first order Gaussian derivative convolution kernels. Because these kernels only detect the edges at a single angle, the application tries multiple angles for detecting all edges [35].

Figure 6.6 shows the top-level data flow graph of the application. The input component feeds 720x576 grey-scale images to 36 component groups that determine

![Data flow graph](image)

![Communication streams](image)

Figure 6.6: Top level structure of the Edge Detector application
the edges for a given angle. Then the application takes the maximum of the results of the angles and sends it to the output component.

Since each angle is independent of the other angles, the application processes all angles in parallel. However, this approach requires much memory for storing intermediate results. We have reduced the memory requirements by splitting the application into four parts. In each part, the application processes nine angles. The first Maximum component takes the maximum over angles one to nine. The following Maximum components have ten inputs and take the maximum of the output of the previous Maximum component and the results of the previous nine angles.

This split into four parts was entirely done in the XSPCL specification of the application and did not involve modification of the components themselves. It shows that XSPCL suits its needs: Using XSPCL, the application developer can explore different implementation strategies, based on global application behavior.

The edge detector application has two modes for processing the individual angles, which differ in quality and speed. The application is reconfigurable and can dynamically switch between both modes at run time. When the application runs in a fixed mode, we will refer to it using the name of the mode. We will now explain the details of each mode.

The Edge-Rot application

Edge-Rot performs edge detection using rotations and one dimensional convolution kernels, as shown in Figure 6.7. For each angle, the application rotates the image over that angle. Then it performs the actual edge detection using four different horizontal first order Gaussian derivative convolution kernels. After each horizontal kernel, it performs smoothing using four different vertical zero order Gaussian derivative convolution kernels. The convolutions yield 16 intermediate results. The application takes the maximum of these results and rotates this 'maximum' image back, which yields the result for one angle.
The communication streams in the Edge-Rot application follow the dependencies shown in Figure 6.7. All vertical convolution kernels are directly connected to the Maximum component.

The application uses three different Gordon components that use optimized SIMD code in their SPE functions. A generic geometric transformation Gordon component performs image rotations. This component is the same component as in the rotator application. A convolution Gordon component performs the horizontal and vertical convolutions at the SPEs. Finally, the maximum operations also run on the SPE using a function that takes the byte-wise maximum of a number of inputs.

At the PPE, the application uses a border component that generates a border around the input image. The convolution operations require this border.

**The Edge-2D application**

Edge-2D performs the edge detection for a given angle using rotated two dimensional convolution kernels. In the direction of the angle, the kernel is a first order Gaussian derivative kernel, which performs edge detection. In the perpendicular direction, the kernel is a zero order Gaussian derivative kernel, which performs smoothing. Figure 6.8 shows the dependency graph for processing one angle. The communication streams follow the dependencies.

Similarly to Edge-Rot, the application combines four different edge detector kernels with four different smoothing kernels. The parameters for these kernels are equal in both modes. The application applies a border before running the convolutions, and aggregates the 16 intermediate results by taking their maximum. Edge-2D uses the same components as Edge-Rot, except that it does not use the geometric transformation Gordon component.

**6.1.6 JPEG Matrix Display**

This application reads 16 compressed motion JPEG image streams and displays these images in a 4x4 matrix display. A motion JPEG stream consists of concatenated JPEG images. Figure 6.9 shows an example output image of the JMD application. All input streams have a resolution of 1280x720. After decompressing the JPEG images, the application scales their width and height down by a factor of four. It combines the 16 resulting 320x180 images into a 4x4 matrix display. The resolution of the output stream is therefore also 1280x720.
Figure 6.9: Output image of the JMD application

Figure 6.10: Global data flow graph of the JMD application

Figure 6.10 shows the data flow graph of the JMD application. It processes all 16 input streams in parallel. The resulting uncompressed images are in YUV format, with one stream for each color component. Because the input files use 4:2:0 color subsampling, the Y stream holds four times more data than both the U and the V stream.

When all 16 images are complete, the application feeds them to blender components that copy the images to their correct positions in the output image. All blenders for one color component execute sequentially; however, the application can process the three color components in parallel. The blender components are the same as in the PiP application; however, there is no background image, as the 4x4 tiled display covers the full image area. The JMD application therefore supplies the first blender for each color component with a dummy background image, which is then augmented with a down scaled input image in each step.

For evaluating different implementation strategies, we have implemented four
6.1. APPLICATION DESCRIPTIONS

Figure 6.11: Data flow graphs for input, decompression, and down scaling in the various versions of the JMD application

versions of the JMD application that use different components for decompressing and down scaling the input images. The MJPEG input components, the blenders and the output components are equal in all versions. Figure 6.11 shows the components that process one of the 16 input streams in each version.

Each version has an MJPEG input component for each input stream which generates a stream of compressed JPEG images from a motion JPEG input file. At initialization time, it reads the images from this file into memory and it creates a table with the address of each individual image. The input stream writes these addresses to its output stream, instead of the actual data, to avoid memory copying. After processing the last image from the file, an MJPEG input component loops to the beginning of the file, which simulates an endless input stream.

The JPEG decoding and down scaling components in Figure 6.11 differ for each version. We will now describe each version and the components it uses. Table 6.2 summarizes the differences between the versions by showing which resources in the Cell processor each version uses for each task.
### JMD-Generic

This version runs on any architecture supported by SP@CE as it uses only generic components without architecture-specific optimizations. JMD-Generic feeds the JPEG images from the MJPEG input component to a bitstream decoder component, which interprets the JPEG bitstream and performs variable length decoding. The bitstream decoding outputs one stream for each color component with Discrete Cosine Transformed (DCT) data. The application feeds this DCT data to Inverse DCT (IDCT) components, which results in full-size uncompressed images. The down scalers then reduce the size of the image by taking the average of each block of four by four pixels.

JMD-Generic exploits task parallelism by processing the components in a pipeline, and by processing the various color components in the images concurrently. It exploits data parallelism by running the IDCT and down scale components using four slices.

### JMD-Cell-IDCT

This version corresponds to JMD-Generic, however, it uses Gordon components for performing IDCT and down scaling. Figure 6.11 displays the Gordon components using a grey background. These Gordon components create a job chain where each job processes part of the image. JMD-Cell-IDCT uses optimized versions of both the IDCT and the down scaling SPE functions, which are up to two orders of magnitude faster than naïve implementations, as explained in Section 5.5.1.

In each iteration, JMD-Cell-IDCT transfers 8,380,800 bytes between main memory and SPE local memory while processing one of the 16 images. The intermediate IDCT data, which is the input for the IDCT function, is 5,529,600 bytes as it uses four bytes per pixel. The full-size decompressed images, which occupy 1,382,400 bytes, are transferred twice, as they are both the output of the IDCT function and the input for the down scaler. Finally, the output of the down scaler is 16 times smaller and thus occupies 86,400 bytes.

### JMD-Cell-JPEG

This version corresponds to JMD-Generic, however, it uses Gordon components for performing IDCT and down scaling. Figure 6.11 displays the Gordon components using a grey background. These Gordon components create a job chain where each job processes part of the image. JMD-Cell-IDCT uses optimized versions of both the IDCT and the down scaling SPE functions, which are up to two orders of magnitude faster than naïve implementations, as explained in Section 5.5.1.

In each iteration, JMD-Cell-IDCT transfers 8,380,800 bytes between main memory and SPE local memory while processing one of the 16 images. The intermediate IDCT data, which is the input for the IDCT function, is 5,529,600 bytes as it uses four bytes per pixel. The full-size decompressed images, which occupy 1,382,400 bytes, are transferred twice, as they are both the output of the IDCT function and the input for the down scaler. Finally, the output of the down scaler is 16 times smaller and thus occupies 86,400 bytes.
6.1. APPLICATION DESCRIPTIONS

JMD-Cell-JPEG

Instead of running only the IDCT decoding part of the JPEG decoder at the SPEs, this version runs a full JPEG decoder on the SPEs. The SPE function decodes the JPEG bitstream, which is compressed using variable-length coding. It stores the result into temporary buffers in local memory, and calls the IDCT SPE function for performing IDCT.

The Gordon component builds a chain of JPEG decoding functions, in which each job produces a fixed amount of output data. Because JPEG uses variable length coding, it is not known in advance how many input bytes each job in the chain requires. The JPEG decoding function at the SPE therefore employs its own double buffering scheme for the input data. The Gordon component stores the input data in a persistent data buffer which holds input data for multiple jobs. It completely fills the input buffer with JPEG data before executing the first job of the chain. Because the buffer is persistent, Gordon does not transfer data to these buffers for the next jobs in the chain.

The jobs use double buffering by splitting the input buffer in two halves. One half always contains enough input data for at least one job, assuming a worst case compression ratio. When previous jobs have consumed either half, the current job manually loads new data into that half using asynchronous DMA transfers. The execution of this job, which only uses data in the other half, then overlaps with the data transfer. When the next job starts, the new data is available immediately, and the process repeats itself.

The job chain uses another persistent buffer for the state of the double-buffered input buffer and the state of the bitstream decoder. The persistent data functionality of Gordon is therefore very useful for this function. It is a natural way of transferring state from one job to the next and allows the jobs to implement double buffering of variable sized data. Without persistent data, it would have been very hard, if not impossible, to develop JPEG decoding at the SPE.

For the output data, manual multi-buffering is not necessary since the size of the output data is known in advance. The Gordon component uses regular output buffers for transferring the decoded images. Gordon automatically performs multi-buffering for these buffers.

JMD-Cell-Combine

This version is similar to JMD-Cell-JPEG, however, besides JPEG bitstream decoding and IDCT, the SPE function in JMD-Cell-Combine also performs down scaling. The SPE function internally uses the same JPEG bitstream decoding routines as JMD-Cell-JPEG. It also uses the optimized IDCT and down scaling functions from the JMD-Cell-IDCT version.

Combining these steps into a single JPEG decoding function reduces overhead, mainly because it requires fewer memory transfers between the SPE and main memory. The application only transfers compressed JPEG data to the SPE, and it only transfers down scaled images from the SPE to main memory. It stores the interme-
diate data in SPE local memory. With an average JPEG file size of 150 kilobytes, JMD-Combine only transfers 240.000 bytes for each image, which is 34.92 times less than JMD-Cell-IDCT.

The drawback of combining several steps is that it reduces parallelism. JMD-Combine no longer processes the different color components concurrently, like the other versions. However, the application still processes all images in parallel and uses multiple concurrent iterations. While the blenders at the PPE composes the output image, the SPEs process the images of the next iteration. The current implementation of the Gordon component in JMD-Cell-Combine is not reentrant, however, which also limits the amount of parallelism.

6.2 Experiments

We analyze the performance of our applications on various parallel architectures. Similarly to Chapters 3 and 4, we run the applications on three architectures: a DAS-3 node, a Niagara server, and a Cell blade. A DAS-3 node contains two dual-core x86 CPUs. It therefore supports four threads. The Niagara server contains a single Sun Niagara T2 processor, which has eight cores. Each core supports eight threads, which yields a total of 64 hardware threads. A Cell blade contains two Cell processors. Each processor has a PPE core, which supports two hardware threads, and eight SPE cores. The Cell blade thus supports four threads and has sixteen SPEs. Section 1.2 gives more details about these systems.

We use multiple similar DAS-3 nodes and Cell blades for our experiments. On these architectures, we repeat each measurement eleven times, and take the average of these 11 repetitions. Since we had access to a single SPARC Enterprise T5120 server, we repeat each Niagara measurement 5 times instead of 11 times. The standard deviation in a set of 5 or 11 repeated measurements is always less than 5.5% of the average measurement.

As shown above, all applications include output components. These components normally write the output to a file, send it over a network connection for further processing, or present it to the user using displayed images or other methods. For avoiding influence of these output devices, we replace the output components by components that discard the result of the application. For example, if the application normally writes its result to a file on a hard disk, its performance may be limited by hard disk speed, which is typically less than the maximum speed of the application itself.

The application only uses the normal output components for demonstration purposes. For debugging the application, the application developer can use output components that write the result to a file, which allows a detailed output analysis.

In the input components we also avoid using external resources while the application is running. All input components that read their input from a hard disk file read the whole file into memory before the application starts. When an input component runs, it only copies the address of the image in memory into its output stream, instead of copying the data. When the component reaches the end of the
file, it restarts inserting images from the beginning of the file into its output streams. When an input file does not completely fit in memory, the component only reads part of the file and it loops to the beginning when it reaches the end of this part.

6.2.1 Parallel performance

We evaluate the efficiency of the applications by analyzing their speedup. On a DAS-3 node, a Cell blade, and on the Niagara, we run PiP, Filter, and JMD-Generic using varying numbers of threads. The Cell applications that use SPEs run using varying numbers of SPEs. Figure 6.12 shows the speedup relative to the performance of the application using a single thread or a single SPE. For the measurements of Figure 6.12d, the application uses 4 PPE threads, which is the number of hardware threads supported by the architecture.

Figure 6.12: Parallel performance on different architectures
CHAPTER 6. APPLICATIONS AND EVALUATION

For DAS-3 and the Cell blade, Figures 6.12a and 6.12c show the speedup using maximally 4 threads. On the Cell blade, the applications do not use the SPEs. For the Niagara, Figure 6.12b shows the speedup with up to 64 threads. On DAS-3, all applications run efficiently. The least efficient application (PiP) achieves a speedup of 3.16 with 4 threads. Filter performs close to ideal, with a speedup of 3.97 with 4 threads. On the Cell blade, speedup is ideal with two threads. With more than two threads, the efficiency drops because two SMT threads then share a single PPE core. On the Niagara, this effect occurs beyond 16 threads, even though the Niagara has 8 cores. PiP performs bad on the Niagara, with a speedup of 9.00 at 64 cores. As we explained in Section 3.4.2, PiP has high overhead. Therefore it does not scale well. JMD-Generic and Filter exploit the multithreading capabilities of the Niagara more efficiently, with speedups of 28.1 and 31.6 using 64 threads, respectively.

Figure 6.12d shows that only the Correlator and Edge-2D efficiently use 16 SPEs. The Correlator trivially exploits parallelism, which yields high efficiency. The two dimensional convolution kernels in Edge-2D have a high computation to communication ratio, which explains its high efficiency. Edge-2D therefore uses less bandwidth than Edge-Rot, which achieves a speedup of 6.4 on 16 SPEs. Because Edge-Rot is less computationally intensive than Edge-2D, the performance of Edge-Rot is still 2.7 times better than Edge-Rot, which is not visible in Figure 6.12d.

The Rotator application achieves a maximum speedup of 7.8. At 8 to 16 SPEs, the standard deviation is relatively high as it varies between 3.0 and 11.0 percent of the average measurement. We noticed that the resize components cause both the limited speedup and the high standard deviation. When we remove the resize components, the application achieves a speedup of 15.0 at 16 SPEs, and the standard deviation drops to at most 1.0 percent of the average measurement. Without resize components, the application immediately discards the output of the rotator components. Normally, the application discards the output of the resize components.

The resize components, which use the PPE only, thus clearly interfere with the rotation components which use the SPEs. Since the resize components mainly perform memory copies, we conclude that these components cause a saturation of the memory bus. This saturation also affects the performance of the SPEs, since they transfer input and output data from and to memory. The high standard deviation is also explained by the saturated memory bus, as performance is less deterministic with a saturated bus.

JMD-Combined suffers from load imbalance, as it only executes 16 big SPE jobs in every iteration. These jobs have varying run times, as they decode JPEG images with varying compression ratios. The total performance is therefore limited by the slowest job. By running multiple concurrent iterations of the application, Hinch increases performance: While waiting for the slowest jobs, the application runs jobs from the next iteration at the SPEs. However, because the Gordon JPEG decoding component is not reentrant, the amount of parallelism remains limited and the load imbalance remains.

The load imbalance also explains the sudden performance increase between 14 and 15 SPEs. Since an SPE may have multiple pending jobs, Gordon accidentally
schedules two slow jobs at a single SPE, which becomes a bottleneck. By adding an extra SPE, the schedule changes and performance improves. Below we will show that this effect does not occur when we limit the number of pending jobs per SPE to one.

We conclude that most of our applications efficiently exploit the resources of parallel architectures. Hinch exploits the parallel resources of homogeneous architectures using multi-threading. The Gordon run time library additionally exploits the heterogeneous resources of the Cell processor. The efficiency depends mainly on the structure of the application, and not on the underlying run time systems. Even challenging applications with multiple different fine-grained coprocessor kernels achieve a decent performance.

6.2.2 Heterogeneity

When developing an application for a heterogeneous architecture, an application developer chooses which resources each component uses. On the Cell, an application developer can build normal components, which use the PPEs, or Gordon components, which use the SPEs for their computations. In this section we evaluate different implementations of the Edge Detector and JPEG Matrix Display applications.

JPEG Matrix Display

In Section 6.1.6 we described different strategies for implementing JPEG decoding and down scaling. Figure 6.13 shows the speedup of all variants for the Cell processor using 1 to 16 SPEs. We compute the speedup relative to the fastest implementation with one SPE, which is JMD-Cell-IDCT. The speedup of JMD-Cell-Combined is lower than in Figure 6.12c, because we use different single-SPE reference performance figures. In Figure 6.12c, the speedup is relative to the single-SPE performance of JMD-Cell-Combined, instead of JMD-Cell-IDCT. Similarly to our previous experiments, we run 4 threads on the PPEs.

As explained in Section 6.2.1, JMD-Cell-Combined suffers from load imbalance. We therefore include measurements in which we limit the number of pending jobs per SPE to one. We have omitted measurements of JMD-Cell-IDCT with one pending job per SPE, since this application does not perform well for other reasons, as we will explain below. With JMD-Cell-JPEG and JMD-Cell-Combined, the performance improves significantly on 7 to 15 SPEs using one pending job per SPE. However, the performance does not increase significantly beyond 10 SPEs. At 16 SPEs, the performance is equal to the original version with multiple pending jobs per SPE.

On one SPE, JMD-Cell-IDCT performs better than JMD-Cell-JPEG and JMD-Cell-Combined. The single SPE is a bottleneck in this case. Since JMD-Cell-IDCT performs JPEG decoding on the PPE, and the other two applications use the SPE for this task, the SPE is less heavily loaded in JMD-Cell-IDCT, which therefore outperforms the other applications. However, performing JPEG decoding on the PPEs has a serious drawback. At three SPEs or more, the PPEs become the bottleneck.
Figure 6.13: Comparison of JMD implementations

in JMD-Cell-IDCT. Whereas the performance of JMD-Cell-JPEG and JMD-Cell-Combined still improves, the maximum speedup of JMD-Cell-IDCT is 1.35.

When we take a closer look at the results of JMD-Cell-JPEG and JMD-Cell-Combined with one pending job per SPE, we see an interesting result. Up to 9 SPEs, JMD-Cell-Combined performs better, as integrating the down scale job into the JPEG decoding job reduces overhead. From 11 SPEs, JMD-Cell-JPEG performs better. As explained before, JMD-Cell-Combined suffers from load imbalance, since it maximally runs 16 SPE jobs in parallel. Because the JPEG decoding job and the down scale jobs are separate in JMD-Cell-JPEG, it can maximally run 64 SPE jobs in parallel. The dynamic load balancer in Gordon automatically improves load balancing and the performance of JMD-Cell-JPEG becomes better than that of JMD-Cell-Combined.

**Edge detector: PPE/SPE Maximum component**

Because taking the maximum is a relatively simple operation, the overhead of performing this operation on an SPE coprocessor using Gordon is high. The Gordon component needs to initialize the jobs it submits to Gordon. Gordon needs to synchronize with the SPE. Furthermore, the SPE needs to transfer the input and output data from and to main memory. However, using the main processor might also incur overhead as it can easily become a bottleneck. Moreover, when it is too busy executing other tasks, the main processor can not keep the SPEs busy, resulting in even more performance loss.

For evaluating if using the coprocessor is worthwhile for performing a simple operation like taking the maximum, we implemented a component that performs this operation on a PPE instead of an SPE. Like the Gordon component, it uses optimized SIMD code that processes 16 pixels at once, which allows a fair comparison. Both components have the same interface and can therefore be interchanged.

We have performed measurements of Edge-Rot and Edge-2D in which we replace all Gordon Maximum components, that use the SPEs, by Maximum components
that use the PPE. Figure 6.14 shows the speedup of both versions, relative to the fastest version using one SPE.

In Edge-2D, computing the maxima at the PPEs hardly affects performance. With 7 SPEs, the PPE maximum version performs 3.1% better than the SPE maximum version. With 16 SPEs, the SPE maximum version is 0.9% faster again.

Edge-Rot becomes significantly slower when computing the maxima at the PPEs. The PPEs clearly become a bottleneck, as the speedup is limited to 2.6x in the PPE maximum version. When offloading the computation to the SPEs, the speedup reaches 6.4x at 16 SPEs, which is 2.5 times better than the PPE maximum version.

6.3 Summary

This chapter describes six applications built using XSPCL, Hinch, and Gordon. The applications include the convolution filter and picture-in-picture applications we used in Chapters 3 and 4, an image rotator, an application that correlates data from different radio telescopes, an edge detector, and a matrix display application that combines 16 motion JPEG input streams into a single output stream. The Edge detector and JPEG matrix display applications have multiple different modes, in which they use different components.

Experiments on three parallel architectures show that the applications can efficiently exploit parallelism. The resulting efficiency depends mainly on the application, and not on the underlying run time systems.

We also evaluated different implementation strategies for heterogeneous applications on the Cell processor. Integrating several SPE operations into one big operation can improve efficiency significantly. However, with only few big components that use the SPEs, load imbalance is likely, especially when the SPE jobs have different sizes. We conclude that using the PPEs for computations is typically not worthwile. For a simple operation such as taking the maximum, there can be a small performance benefit. However, the performance loss when using the PPEs instead of the SPEs can be severe.
Chapter 7

Conclusion

This chapter summarizes this thesis in Section 7.1 by giving an overview of the various problems of developing parallel streaming applications, and the various systems we built that address these problems. Section 7.2 describes future work directions for the systems we developed.

7.1 Summary

This thesis presents a framework for developing streaming applications for heterogeneous MPSoC architectures. In a streaming application new data continuously enters the application. The application performs similar operations on each data item and sends the result to one or more output devices. A streaming application consists of components that perform different operations. Components are connected by streams, which provide communication channels between components. Components may also communicate using events, which are short notification messages. Section 1.1 described streaming applications in more detail.

In this section, we will repeat the research questions we posed in Section 1.4. We will summarize the problems we encountered answering these questions. Finally, we will summarize the systems we developed. The design, implementation and evaluation of our systems form the contributions of this thesis.

7.1.1 Research questions

At the beginning of this thesis we posed three research questions. Answering these questions requires solving many problems. The complexity of these problems ranges from trivial to very complex. In this section, we will list our research questions and the most complex problems we encountered while answering them.

Addressing the various individual problems separately does not answer the research questions. One of the most complex problems we encountered was integrating the solutions to various problems into a single system. As this problem applies to all
questions, we will not specify it with each research question. We will now list the three research questions this thesis answers.

1. **How can one abstract the various challenges of building parallel streaming applications behind a simple interface, with low overhead?**

Our answer to this question is building a software system that provides this abstraction. We have identified the following problems when building parallel streaming applications:

- A streaming application performs various operations that may be implemented using different library functions, with different interfaces, or even using different programming languages. A developer has to integrate all operations in a single application, by creating a component for each operation. An operation may occur multiple times in different parts of an application. A developer has to ensure that the application can use the components that implement these operations multiple times, without conflicts.

- A streaming application runs as a series of iterations, in which each component performs its operations on a new data item. A developer must structure the application in a way that resembles this behavior. In each iteration, the components in a streaming application have to run in a specific order, since there are dependencies between components. Components that produce data that is used by other components, have to run first. The application thus has to schedule the execution of components properly.

- Components communicate using streams and events. A stream or event channel needs to temporarily store data. A single stream or event channel may be accessed by more than two components. Some components access data from previous iterations from a stream, besides the data from the current iteration. A developer has to implement these communication streams and event channels, and handle various border cases.

- When running on a parallel architecture, a streaming application has to exploit parallelism. A developer has to split the application into multiple concurrent parts, and avoid race conditions by synchronizing these parts. Moreover, a developer has to perform load balancing of these parts.

- Streaming applications may be reconfigurable. A component may have parameters, which the application adjusts at run time in response to events. An application could also reconfigure itself, which means it adds or removes components and streams while the application is running. The internal structure of a streaming application thus has to be dynamic, which affects the entire application.
7.1. SUMMARY

- A system that provides abstractions for building parallel streaming applications should have a simple interface towards the developer. Designing such an interface is difficult, as it has to support all functionality described above. For avoiding a steep learning curve, all advanced functionality should be optional. Using a set of basic functions, a developer can then quickly build applications. Once a developer knows these basic functions, the developer can advance to more advanced functions, for further optimizing performance, for example.

- For achieving low overhead, which is part of the research question, a streaming application should perform its operations as efficiently as possible. This requirement applies to the entire application, including the functionality described above.

2. How does one design and implement a high-level coordination language for parallel streaming applications?

A generic coordination language for parallel streaming applications should support specifying all aspects of streaming applications. A compiler for such a language should generate the necessary run time structures for the application. The most complex problems include:

- The language should have primitives for expressing basic elements in a streaming application, including components, streams and events. The language should have grouping constructs for creating the application out of multiple components. These grouping constructs also express parallelism in the application.

- Using the language should be easy, which means its syntax should be clear and simple. The language should have features for reusing pieces of code that are used multiple times within an application, or even across different applications.

- The compiler has to generate application-specific structures. For generic structures that are common to all parallel streaming application, the compiler can use a generic run time system that provides these structures.

- The language and the compiler should ensure that an application written in the coordination language has low overhead compared to hand-written applications. The language and the compiler should particularly avoid overhead in the critical path of the application.

3. How can one abstract the various difficulties of using heterogeneous resources with distributed memory semantics behind a simple interface, with low overhead?

Similarly to our first research question, our answer is building a software system that provides this abstraction. We have identified the following problems when building such a system:
• A system that supports using heterogeneous resources with distributed memory semantics performs many tasks. The system has to allocate resources, schedule application tasks, and transfer data, amongst others. When the architecture provides multiple similar resources, the system has to perform load balancing. Ideally, the system hides these tasks from its user.

• Achieving optimal performance on resources with distributed memory semantics requires performing complex optimizations, such as multi-buffering. Ideally, the system performs these optimizations automatically.

• Since the system does not target a specific application type, its interface should be suitable for all applications, which may have different requirements. The system interface should for example support both polling and interrupt semantics. For aiding the developer, the interface should be as simple as possible.

7.1.2 Contributions

The contributions of this thesis consist of three systems that address the research questions posed above. The Hinch run time system, the XSPCL coordination language and the Gordon run time library solve the problems related to the first, second and third research question, respectively. These systems are part of the SP@CE programming environment, which we described in Section 1.5. They are the building blocks for a valuable framework for developing parallel streaming applications. We will summarize these systems in the remainder of this section.

The Hinch run time system

Hinch is a generic run time system for streaming applications, which abstracts a developer from low-level communication and synchronization primitives. A streaming application that uses Hinch contains a data flow graph of components. Towards an application developer, the Hinch API has functions for building the data flow graph and connecting communication channels between components. Towards the component developer, the Hinch API has functions for accessing the data in the communication channels that are connected to the component.

Hinch was designed for parallel architectures. It supports both task- and data parallelism and performs automatic load balancing. Hinch focuses on exploiting parallelism between components, instead of in components. A component developer only has to write sequential code, which simplifies the effort for the developer. The current implementation supports parallel architectures based on shared memory, however, its design does not exclude a distributed memory implementation.

Hinch supports both streaming and event-based communication channels between components. Hinch handles all synchronization and memory management for these communication methods. Multicast streams and many-to-one event communication channels are fully supported.

The overhead of using applications that use Hinch instead of hand-written applications is typically less than five percent. This overhead depends heavily on the
7.1. SUMMARY

 architecture and the application. Sometimes, Hinch applications are even faster than hand-written equivalents.

 All Hinch structures are designed for reconfigurable applications. The application can create, modify and destroy structures, such as components and communication channels, while the application is running. By comparing reconfigurable applications against non-reconfigurable applications we have shown the overhead of dynamic reconfiguration. Even in a worst case scenario in which reconfiguration occurs very often, the overhead is typically less than seven percent.

 One of the lessons we learned from building Hinch is that parallel streaming applications require a modular run time system, where each module has different responsibilities. Furthermore, modeling streaming applications as data flow process networks has many advantages, such as automatic load balancing. We did not encounter any significant downsides of this approach. Chapter 3 described Hinch in detail.

 The XSPCL coordination language

 The XSPCL coordination language is a declarative language for specifying the relations between the components in a streaming application. Using XSPCL, an application developer specifies the data flow graph of the application by recursively specifying component groups. XSPCL supports various grouping constructs, such as sequential, task parallel, and data parallel. It also has methods for specifying component groups that occur multiple times.

 In XSPCL, communication channels between components are specified independently from the data flow graph. XSPCL automatically creates and connects streams or event channels between components that use the same stream or event channel, respectively.

 XSPCL supports reconfigurable applications. An application developer can declare parts of the data flow graph as optional, and enable or disable these optional parts when an event occurs. The XSPCL compiler automatically creates the necessary application-specific structures and routines for reconfigurable applications. This abstraction is very valuable, as manually performing reconfiguration is difficult.

 We have implemented an XSPCL compiler that compiles an XSPCL application to a C source file that uses the Hinch API. A standard C compiler compiles this source file and links it to the Hinch run time system, which is also written in C. Since an XSPCL application effectively uses the Hinch API, the overhead of using XSPCL compared to directly using the Hinch API is typically less than a few percent. Chapter 4 described XSPCL in detail.

 The Gordon run time library

 The Gordon run time library abstracts a developer from the problems of using the SPE coprocessors in the Cell architecture. These problems include synchronization, communication, and load balancing. A developer who uses Gordon only has to specify what computations the application has to execute on the SPEs, and on
which data. Gordon fully handles how the SPEs execute these computations and transfer the data.

Gordon includes many optimizations, including fast notifications, automatic multi-buffering, job chaining, and persistent data. Besides these optimizations, a developer can always use application-specific optimizations such as optimized SIMD code or manual DMA transfers. Gordon does not restrict these application-specific optimizations in any way.

Gordon requires a developer to split the computation into small jobs, that fit into the local memory of an SPE, which has a limited size. This restriction is not specific to Gordon, as the limited SPE local memory always forces a developer to split the computation into small parts.

We have evaluated the different notification mechanisms in the Cell processor using Gordon, and concluded that the default DMA communication mechanism performs best. The special-purpose mailbox and interrupt notification mechanisms in the architecture are useless for our generic Cell run time library.

Using experiments we show that the overhead of Gordon mainly consists of DMA transfers on behalf of a job. The overhead without DMA transfers is less than 5000 processor cycles per job.

Hinch and XSPCL applications can use Gordon using special Gordon components, which provide the glue between Hinch and Gordon. This way, an application can easily exploit heterogeneous resources. Towards Hinch, these components act like normal components, although Hinch performs some specific optimizations for these components. Internally, these components use the Gordon API for performing their operations.

Chapter 5 describes the Cell architecture and Gordon in detail. It also presents measurements of the impact of the various optimizations, which reduce the overhead of Gordon.

7.1.3 Applications

We have evaluated our framework using several applications: A picture-in-picture application, a convolution filter application, an image rotator, an application that correlates data from radio telescopes, an edge detector and a JPEG matrix display application.

Experiments on three parallel architectures show that the applications can efficiently exploit parallelism. The resulting efficiency depends mainly on the application, and not on the underlying run time systems. We have also evaluated different implementation strategies for heterogeneous applications on the Cell processor, and concluded that using the PPEs for computations is typically not worthwhile.

7.2 Future work

In previous chapters, we have identified several future work directions. This section augments these directions and gives an overview of possible future work.
7.2. FUTURE WORK

7.2.1 Distributed memory / Grid computing

Currently, our systems focus on shared memory architectures. The only exception is that we use the SPE coprocessors in the heterogeneous Cell architecture, which has distributed memory semantics. The current implementation does not support other distributed memory architectures, such as compute clusters or grid computing infrastructures.

Adding support for distributed memory architectures mainly requires changing the Hinch run time system. Hinch has to schedule components over a distributed architecture, for example using work stealing algorithms. Moreover, the existing shared memory streams and event queues have to be augmented with support for distributed memory. For optimal performance, components that communicate intensely should be scheduled together, which requires new grouping constructs that interface with the Hinch scheduler.

The existing components require no thorough modifications. A component developer mainly has to add serialization and de-serialization routines, which Hinch uses for sending the component to a different machine. The XSPCL coordination language can easily be augmented with support for new grouping constructs.

For heterogeneous grid architectures, we envision implementing a platform-independent Java version of Hinch, on top of the Ibis communication library\[96\]. Ibis has proven successful in supporting other grid applications. Ibis automatically generates serialization and de-serialization routines, which relieves the component developer from this task.

7.2.2 Load balancing heterogeneous resources

Since our systems support dynamic reconfiguration, an application that uses our systems can interchange components that execute similar operations using different resources. For example, one component could use the main processor for its computations, whereas another component could perform the same computations using a special coprocessor, such as an SPE in the Cell processor or a GPU.

For achieving optimal load balancing, the system should monitor the availability of all resource types and interchange components when needed. The implementation of such a monitoring system is subject to further research.

The main difficulties of a monitoring system are determining the conditions for performing reconfiguration, and choosing how to reconfigure the application, when there are multiple options. The system needs information about the performance of all components on all possible architectures. Since the solution space grows exponentially with the number of components, the system should use clever heuristics for avoiding overhead.

Furthermore, the system has to ensure that the application arrives at a stable state. After reconfiguration, resource availability changes. The system might therefore want to reconfigure again, which may cause and endless feedback loop. The system has to avoid these loops since they will incur significant overhead.
7.2.3 Real time systems

Our current systems do not have special support for real time applications, as we have focused on optimizing throughput. However, embedded streaming applications often require some support for real time behavior. Our systems can be augmented with this support, for example, by using a priority queue instead of a standard queue for scheduling.

XSPCL could be augmented with primitives for expressing deadlines in real time applications. The XSPCL compiler can verify that the application meets its deadlines by deriving the worst case execution time (WCET) of individual components. Based on the various grouping constructs in the component graph and the available resources, it can recursively calculate the WCET of the whole application, and compare this number to the available processing time.

A more advanced strategy is exploiting dynamic reconfiguration in real time applications. Similarly to the monitoring system for load balancing heterogeneous resources we described above in Section 7.2.2, a monitoring system could monitor the resource consumption of the application, and reconfigure the application accordingly. For example, if resource consumption is high and the application misses deadlines, the application could switch to algorithms with a lower compute intensity, at the cost of lower output quality.

In real time systems, an application often has to handle events in a timely and predictable manner. Our current system does not take the iteration a component executes into account when delivering events. The event communication mechanism could be augmented with support for teleport messages, which are events annotated with a minimum and maximum latency, in terms of iterations\[118\]. The sender of an event then controls in which iteration(s) an event may be delivered, which increases predictability.

7.2.4 Automatic component combining

When a streaming application contains many small components, scheduling each component as separate entity may incur a large overhead. We have witnessed this behavior with a Picture-in-Picture application.

An application could reduce this overhead by combining components into a component group, which is scheduled as one entity. The component group then runs the components inside it in a fixed order, without exploiting parallelism. The component group could exploit cache behavior, by running a component that reads data from a stream immediately after the component that has written data to that stream.

The communication streams between the components in the group can also be simplified, since at most one component accesses these streams in every iteration. This simplification further reduces overhead. Since the access pattern to these streams is fixed, other optimizations could be performed as well.

Ideally, the XSPCL compiler or another tool should detect when it should group components this way and automatically create these groups. Since the overhead of our systems is already low, we have not pursued research in this direction, as it would
probably only provide significant performance benefits in limited cases. However, Gordon et al. do report significant benefits of a similar optimization [57].

7.2.5 Challenging applications

Besides the various applications described in Chapter [9], future research could investigate if our systems are also applicable to other challenging applications. Example future applications include MPEG-2 and H.264 decoding, and combining audio and video streams in a single application.

One could also build a general-purpose audio/video streaming application, that automatically configures itself according to its input and output types, similarly to the MPlayer application [4]. The main advantage of using our systems is that they are designed with support for parallelism in mind. Existing systems, such as MPlayer, only exploit parallelism in limited cases.

Since Gordon is a generic run time library for the Cell architecture, applications other than streaming applications can also benefit from Gordon. For example, the Gromacs package [112], which simulates molecular dynamics, could use Gordon for computing interactions between atoms on the SPEs.

7.2.6 Graphical front-end

Although XSPCCL provides a valuable abstraction for building streaming applications, an application developer has to write XML code for specifying streaming applications. It would be more user-friendly to present a graphical interface that visualizes XSPCCL specifications to the application developer. Such an interface should graphically represent components, data flow dependencies and communication streams between components, and allow zooming in to and out of grouping components, amongst others.

Other systems have successfully used graphical interfaces for composing applications out of several building blocks, including CO2P3S [86], Soundium [92], the Common Component Architecture [9], and Fractal [26].
application developer, 14
asynchronous dependency, 49
component, 11, 27, 42, 43, 72
component developer, 14
dependency, 46
dependency list, 47
end-user, 13
event, 12, 52
Gordon component, 114
GPU, 12
grouping component, 33
head dependency, 46
head job, 46
Hinch application, 33, 89
iteration, 15, 29
job (Gordon), 102
job (Hinch), 37, 42, 46
job chain, 102
job chaining, 96
kernel, 27
manager, 53
module (Hinch), 42
port, 37
PPE, 13
SIMD, 13
slice, 35
SPE, 13
stream, 11, 29, 37, 51
synchronous stage, 49
tail dependency, 46
tail job, 46
XSPCL application, 68
Bibliography


In deze samenvatting zullen we eerst de onderzoeksvragen vermelden. We geven een samenvatting van de problemen die we tegenkwamen bij het beantwoorden van deze vragen. Daarna geven we een samenvatting van de systemen die wij hebben gebouwd. Het ontwerp, de implementatie en de evaluatie van deze systemen vormen de contributies van dit proefschrift.

**Onderzoeksvragen**

In dit proefschrift hebben wij drie onderzoeksvragen gesteld. Voor het beantwoorden van deze vragen hebben wij veel problemen op moeten lossen. De complexiteit van deze problemen varieert van triviaal tot zeer complex. In deze sectie zullen we de onderzoeksvragen één voor één behandelen. Bij elke onderzoeksvraag vermelden wij de meest complexe problemen die we tegenkwamen bij het beantwoorden van de vraag.

Het afzonderlijk aanpakken van de problemen levert geen antwoord op de onderzoeksvragen. Eén van de meest complexe problemen was het integreren van de oplossingen voor verschillende problemen in één enkel systeem. Omdat dit probleem zich voordoet bij alle onderzoeksvragen, noemen we dit niet bij de onderzoeksvragen afzonderlijk. Hieronder volgen de drie onderzoeksvragen waar dit proefschrift een antwoord op geeft.
1. Hoe kan men de verschillende moeilijkheden van het bouwen van parallelle streaming applicaties abstraheren door middel van een simpele interface, met weinig extra kosten?

Ons antwoord op deze vraag is het bouwen van een systeem dat deze abstractie levert. We hebben de volgende problemen gevonden voor het bouwen van parallelle streaming applicaties:

- Een streaming applicatie voert verschillende operaties uit die geïmplementeerd kunnen zijn in verschillende bibliotheken met verschillende interfaces, of zelfs met verschillende programmeertalen. Een applicatie-ontwikkelaar moet alle operaties in één enkele applicatie integreren, door een component te maken voor elke operatie. Eenzelfde operatie kan meerdere keren voorkomen in verschillende delen van een applicatie. De ontwikkelaar moet ervoor zorgen dat een applicatie de componenten die deze operaties implementeren meerdere keren kan gebruiken, zonder conflicten.

- Het uitvoeren van streaming applicaties gebeurt door middel van meerdere iteraties. In elke iteratie voert elke component zijn operaties uit op een nieuw data element. De ontwikkelaar moet de applicatie structureren aan de hand van dit gedrag. In elke iteratie voert een streaming applicatie de componenten in een vaste volgorde uit omdat er afhankelijkheden bestaan tussen componenten. Componenten die data produceren voor andere componenten moet de applicatie eerst uitvoeren. De applicatie moet de uitvoering van de componenten dus op de juiste manier ordenen.

- Componenten communiceren met streams en events. Een communicatiekanaal voor streams of events moet tijdelijk data opslaan. Het is mogelijk dat meer dan twee componenten hetzelfde communicatiekanaal gebruiken. Sommige componenten gebruiken data uit voorgaande iteraties, naast de data van de huidige iteratie. Een ontwikkelaar moet deze communicatiekanalen implementeren en daarbij rekening houden met de verschillende randgevallen.

- Wanneer een streaming applicatie wordt uitgevoerd op een parallellerearchitectuur, dient de applicatie dit parallelisme te benutten. De ontwikkelaar moet de applicatie opsplitsen in meerdere delen die parallel uitgevoerd kunnen worden. Daarbij moet de ontwikkelaar raceproblemen vermijden door de verschillende delen op de juiste manier te synchroniseren. Bovendien moet de ontwikkelaar de delen goed verdelen over de verschillende parallelle rekenenheden.

- Streaming applicaties kunnen herconfigurerbaar zijn. Een component kan parameters hebben die door de applicatie worden aangepast tijdens het uitvoeren in reactie op events. De applicatie kan zichzelf ook herconfigureren door componenten en communicatiekanalen toe te voegen of te verwijderen.
ONDERZOEKSVRAGEN

171
terwijl de applicatie wordt uitgevoerd. De interne structuur van streaming applicaties moet dus dynamisch zijn. Deze eis heeft betrekking op alle delen van de applicatie.

- Een systeem dat abstracties levert voor het bouwen van parallelle streaming applicaties moet een simpele interface hebben voor de ontwikkelaar. Het ontwerp van zo’n interface is moeilijk, omdat de interface alle hierboven beschreven functionaliteit moet ondersteunen. Om een steile leercurve te voorkomen moeten alle geavanceerde functies optioneel zijn. Met behulp van de basis functies kunnen ontwikkelaars dan snel applicaties bouwen. Wanneer een ontwikkelaar de basis functies onder de knie heeft, kan hij verder gaan met het gebruiken van de geavanceerde functies om bijvoorbeeld de snelheid van de applicatie te verbeteren.

- Voor het bereiken van lage extra kosten, wat onderdeel is van de onderzoeksvraag, moet een streaming applicatie zijn operaties zo efficiënt mogelijk uitvoeren. Deze eis geldt voor de gehele applicatie, inclusief de hierboven beschreven functionaliteit.

2. Hoe kan men een hoog-niveau coördinatietaal voor parallelle streaming applicaties ontwerpen en implementeren?

In een generieke coördinatietaal voor parallelle streaming applicaties moeten alle aspecten van streaming applicaties uit te drukken zijn. Een vertaler (‘compiler’) voor zo’n taal moet de structuren die nodig zijn voor het uitvoeren van de applicatie genereren. De meeste complexe problemen hierbij zijn als volgt:

- De taal moet primitieven hebben om de basiselementen van een streaming applicatie, zoals componenten, streams en events, uit te drukken. De taal moet groeperingsstructuren hebben om de applicatie op te bouwen uit meerdere componenten. Deze groeperingsstructuren representeren ook het parallelisme in de applicatie.

- De taal moet makkelijk te gebruiken zijn, wat betekent dat de syntax duidelijk en simpel moet zijn. De taal moet een manier hebben om stukken code die meerdere keren voorkomen binnen een applicatie, of zelfs binnen meerdere applicaties, te hergebruiken.

- De vertaler voor de taal moet applicatie-specifieke structuren genereren. Voor de generieke structuren die in alle parallelle streaming applicaties worden gebruikt, kan de vertaler een generiek runtime systeem gebruiken dat deze structuren levert.

- De taal en de bijbehorende vertaler moeten ervoor zorgen dat applicaties die in de coördinatietaal zijn geschreven lage extra kosten hebben ten opzichte van applicaties die met de hand zijn geschreven. De taal en de vertaler moeten vooral extra kosten in het kritieke pad van de applicatie vermijden.
3. Hoe kan men de verschillende moeilijkheden van het gebruik van heterogene rekeneenheden met gedistribueerd geheugen abstraheren door middel van een simpele interface, met weinig extra kosten?

Analooog aan onze eerste onderzoeksvraag is ons antwoord het bouwen van een software systeem dat deze abstractie levert. We hebben de volgende problemen geïdentificeerd voor het bouwen van zo’n systeem:


- Het behalen van optimale prestaties op rekeneenheden met gedistribueerd geheugen vereist complexe optimalisaties, zoals meervoudig bufferen. In het ideale geval voert het systeem deze optimalisaties automatisch uit.

- Omdat het systeem zich niet richt op een specifiek type applicatie moet de interface van het systeem geschikt zijn voor alle applicaties, die verschillende eisen kunnen hebben. Het systeem moet bijvoorbeeld zowel synchrone als asynchrone communicatie ondersteunen. Bij synchrone communicatie vraagt de applicatie aan het systeem of er nieuwe berichten zijn. Bij asynchrone communicatie onderbreekt het systeem de applicatie bij een nieuw bericht. Om de ontwikkelaar te helpen moet de interface van het systeem zo simpel mogelijk zijn.

**Contributies**

De contributies van dit proefschrift bestaan uit drie systemen die zich richten op bovenstaande onderzoeksvragen. Het Hinch runtime systeem, de XSPCL coördinatietaal en de Gordon runtime bibliotheek lossen de problemen op die gerelateerd zijn aan respectievelijk de eerste, tweede en derde onderzoeksvraag. Deze systemen zijn onderdeel van de SP@CE programmeeromgeving die we hebben beschreven in dit proefschrift. Zij zijn de bouwstenen van een waardevol raamwerk voor het ontwikkelen van parallelle streaming applicaties. In de rest van deze sectie zullen wij deze systemen samenvatten.

**Het Hinch runtime systeem**

Hinch is een generiek runtime systeem voor streaming applicaties, dat de ontwikkelbaar abstrahereert van laag niveau communicatie en synchronisatie primitieven. Een streaming applicatie die Hinch gebruikt bevat een gegevensstroom-graaf met componenten. Voor een applicationontwikkelaar heeft Hinch functies voor het bouwen van

Hinch ondersteunt zowel streaming- als event-gebaseerde communicatie tussen componenten. Hinch zorgt voor alle synchronisatie en al het geheugenbeheer in deze communicatiekanalen. Streams met meerdere ontvangers en event communicatiekanalen met meerdere zenders worden volledig ondersteund.

Het prestatieverlies van applicaties die Hinch gebruiken ten opzichte van handgeschreven applicaties is typisch minder dan vijf procent. De hoogte van dit verlies hangt zwaar af van de architectuur en de applicatie. Soms zijn Hinch applicaties zelfs sneller dan equivalente handgeschreven applicaties.

Alle structuren in Hinch zijn ontworpen voor herconfigureerbare applicaties. De applicatie kan structuren, zoals componenten en communicatiekanalen, maken, wijzigen en verwijderen terwijl de applicatie wordt uitgevoerd. Door herconfigureerbare applicaties te vergelijken met niet-herconfigureerbare applicaties hebben we de kosten van herconfiguratiebehaard bepaald. Zelfs bij een extreem slecht geval waarbij herconfiguratie zeer vaak gebeurt, is het prestatieverlies typisch minder dan zeven procent.

Een van de lessen die we geleerd hebben van het bouwen van Hinch is dat parallellle streaming applicaties een modulair runtime systeem nodig hebben, waarbij elke module andere verantwoordelijkheden heeft. Daarnaast merkten we dat het modelleren van streaming applicaties als gegevensstroomnetwerken veel voordelen heeft, zoals automatische verdeling van werk over de reken eenheden. We hebben geen onoplosbare problemen ondervonden met deze aanpak.

De XSPCL coördinatietaal

De XSPCL coördinatietaal is een declaratieve taal voor het specificeren van de relaties tussen de componenten in een streaming applicatie. Met XSPCL specificeert een applicatieontwikkelaar de gegevensstroom-graaf van de applicatie door recursief componentgroepen te specificeren. XSPCL ondersteunt verschillende groepingsstructuren, zoals sequentieel, taak-parallel en data-parallel. De taal heeft ook methoden om componentgroepen te specificeren die meerdere keren voorkomen.

In XSPCL worden communicatiekanalen onafhankelijk gespecificeerd van de gegevensstroom-graaf. XSPCL maakt automatisch communicatiekanalen voor streams en events tussen componenten die hetzelfde communicatiekanaal gebruiken.
XSPCL ondersteunt herconfigurerbare applicaties. Een applicatieontwikkelaar kan delen van de gegevensstroom-graaf optioneel maken en deze delen in- en uitschakelen wanneer er een gebeurtenis plaats vindt. De vertaler voor XSPCL genereert de nodige applicatie-specifiche structuren en routines voor herconfigurerbare applicaties automatisch. Deze abstractie is zeer waardevol, omdat handmatig herconfigureren erg moeilijk is.

We hebben een vertaler voor XSPCL gemaakt die een XSPCL applicatie vertaalt naar een bestand met broncode in C die Hinch gebruikt. Een standaard vertaler voor C vertaalt dit bestand vervolgens en koppelt het aan het aan het Hinch runtime systeem, welke ook geschreven is in C. Omdat een XSPCL applicatie dus effectief Hinch gebruikt is het prestatieverlies bij het gebruik van XSPCL, in vergelijking met rechtstreeks Hinch gebruiken, typisch minder dan een paar procent.

De Gordon runtime bibliotheek

De Gordon runtime bibliotheek abstraherst een ontwikkelaar van de problemen van het gebruik van de SPE rekeneenheden in de Cell architectuur. Deze problemen omvatten synchronisatie, communicatie en werkverdeling. Een ontwikkelaar die Gordon gebruikt hoeft alleen te specificeren welke berekeningen de applicatie moet uitvoeren op de SPEs en op welke data. Gordon zorgt er volledig voor hoe de SPEs deze berekeningen uitvoeren en de data oversturen.


Gordon vereist dat een ontwikkelaar zijn berekeningen splits in kleine taken die in het lokale geheugen van een SPE passen. Dit lokale geheugen heeft een beperkte grootte. Deze beperking hoort niet specifiek bij Gordon, aangezien de beperkte geheugengrootte een ontwikkelaar altijd dwingt om de berekening in meerdere kleine delen op te splitsen.


Met experimenten laten we zien dat de kosten van Gordon voornamelijk bestaan uit het oversturen van data met DMA voor een taak. De kosten zonder DMA overdrachten zijn minder dan 5000 rekyecycli per taak. We meten ook de impact van de verschillende optimalisaties die de kosten van Gordon reduceren.

componenten Gordon voor het uitvoeren van hun berekeningen.

**Applicaties**

We hebben ons raamwerk geëvalueerd met verschillende applicaties: Een beeld-inbeeld applicatie, een convolutie-filter applicatie, een beeld roteerder, een applicatie die data van radio telescopen correleert, een randdetector en een JPEG matrix beeld applicatie.

Experimenten op drie parallelle architecturen laten zien dat de applicaties parallelismus efficiënt kunnen benutten. De resulterende efficiëntie hangt vooral af van de applicatie en niet van de onderliggende runtime systemen. We hebben ook verschillende implementatiestrategieën voor heterogene applicaties voor de Cell architectuur geëvalueerd en geconcludeerd dat het over het algemeen beter is om de SPE reken- eenheden te gebruiken dan de PPE reken Eenheden, zelfs voor simpele taken.