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 indivual 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
CHAPTER 7. CONCLUSION

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
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.