ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Semester Thesis
at the Department of Information Technology
and Electrical Engineering

A DAL backend for the Parallella platform

AS 2014

Dominik Böhi

Advisors: Andreas Tretter
Lars Schor
Professor: Prof. Dr. Lothar Thiele

Zurich
23rd January 2015

Abstract

To satisfy the need for increased performance, embedded systems are getting more complex and more intregraded. The trend is to integrate multiple cores on the same chip. These systems offer increased performance, but are harder to program.

The Parallella platfrom. produced by Adapteva, is an example of such a system which contains a chip containing 16 cores. To offer an easier way of programming such a system, this thesis presents a new backend for the Distributed Application Layer (DAL), a software framework for the development of parallel applications. This backends allows the execution of applications developed using DAL on the Parallella.

Acknowledgements

First of all, I would like to thank Prof. Dr. Lothar Thiele for giving me the opportunity to write this thesis in his research group.

I would also like to thank my advisors, Andreas Tretter and Lars Schor, for their invaluable support during this thesis. They always had time to answer my questions, and were a big help in making in this thesis possible.

Contents

List of Figures

1 Introduction

1.1 Motivation

To satisfy the demand for increasing computing performance, processor designs are getting more complex and more integrated. In the world of embedded systems, the current trend is to increase the number of cores integrated in the same chip, and to combine different architectures. These systems offer increased computing power, but are also harder to program.

One way to offer a better experience to developers is to abstract away the low level details of the platform. An interesting approach for this is the use of process networks to specify parallel applications, where an application is designed as a set of processes connected with FIFO (first-in, first-out) channels. A supporting software framework is then responsible for compiling and running such an application on the actual hardware, and providing the necessary communication methods. The distributed application layer (DAL), developed at ETH Zurich, is an example of such a system. There are already backends for a number of systems, such as the Intel Xeon Phi, OpenCL capable devices, and the Intel SCC.

However, these systems are not very portable and usually not designed with power efficieny as a main goal. The Epiphany is a new architecture designed by Adapteva Inc. with the goal to provide a power efficient manycore architecture. To raise interest in the architecture, Adapteva raised funds using Kickstarter to produce the open-source Parallella board, which combines a 16-core Epiphany chip with the Xilinx Zynq chip to produce a credit-card

sized computing platform similar to the popular Raspberry Pi.

This thesis presents a new backend for DAL with the capability to run DAL applications on the Parallella.

1.2 Contributions

- A new backend for DAL is presented, which enables the execution of DAL applications on the Parallella.
- Using existing benchmark applications for DAL, we offer an inital evaluation of the backend.

1.3 Related Work

Most of the existing work discussing the Parallella is focussed on the perfomance characteristics or the power consumption of the platform. There is only one paper which looks at code generation for the Parallella.

In [1], a code generation framework for applications written using the CAL actor language is presented and its output compared to applications handwritten for the Epiphany. Their results show that auto-generated code is able to achieve comparable perfomance, while requiring significantly less code lines to be written. This work is similar to ours, with the biggest difference to our work lying in the specification of the original program, which is done using Kahn process networks and a finite state machine in DAL, and the added capabilites of DAL to change active applications during runtime.

A performance analysis of the 64 core Epiphany chip is done in [2], using microbenchmarks to analyse the performance characteristics of individual compomenents, and discussing the implementation of different algorithms. Especially mentioned is the problem of the limited memory size, which requires finetuning an application to make optimal use of the Epiphany.

There is also a set of papers looking at the possibility of using the Epiphany to save Energy: [3] looking at compiler options, and [4] and [5] discussing power saving opportunies for specific problems.

1.4 Outline

Chapter 2 will describe Kahn process networks, how they are used in DAL to allow efficient development of parallel applications, and the Parallella platform. In chapter 3, the main challenges for running DAL applications

on the Epiphany and the proposed solutions are presented. The results of running a sample DAL application on the Parallella are shown in chapter 4. Finally, the achieved work is summarised and possibilities for improvements are discussed in chapter 5.

2 Background

This chapter will introduce the Kahn process networks used to specify applications which are executed by DAL, and give an overview over the Parallella platform.

2.1 Kahn Process Networks

Kahn process networks (KPNs) [6] are a method to specify parallel applications. They consist of a set of processes connected using one way channels, which are the only method for processeses to communicate with each other. Figure 2.1 shows a simple KPN. The channels have infinite capacity, which means that a process can always send a message over a channel. When a process decides to read from a channel, however, it has to wait until data is available. It cannot check whether there is data available before reading. This leads to an important property of KPNs, namely that the execution is deterministic and only depends on the input itself, and not on the time of arrival of the input.

2.2 DAL Framework

The DAL [7] is a software development framework which allows the developmenent of distributed applications specified as KPNs, and the ability to map these applications onto different hardware platforms and execute them.

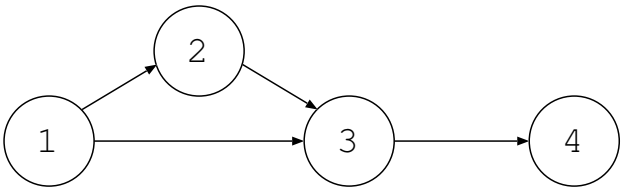

Figure 2.1: Example of a Kahn process network with 4 processes and 4 channels

A single application for DAL consists of a Kahn process network, specified as an XML-File, which describes the connections between the individual processes, and the source code of these processes. A process for a DAL application has to implement three functions, an *init* function, which is used to initialize the process on startup, a *fire* function, which is the main part of a process and is continuously executed during the run time of the process, and a *finish* function, which can be used to clean up ressources when the process is shut down.

To create executable source code for the target platform, a mapping from processes to actual hardware resources has to be specified. Using this mapping, DAL is used to create wrapper code for the processes which allows them to be compiled and executed on the target platform. This wrapper code also ensures that the correct library functions are used when the processes use the channels to communicate.

To manage the startup of the individual processes and to setup the communication channel, DAL also creates an additional controller application. This application consists of multiple processes, which are hierarchically organised. A main process is responsible for overall coordination, and several slave processes are created to manage the actual processes. In a typical architecture, one slave process is responsible for a related group of processors. Chapter 3 will discuss how the controller for the Epiphany is implemented.

2.3 Parallella

The Parallella is a new credit-card sized computing platform designed and sold by Adapteva Inc. [8]. It was developed with funds raised by a Kickstarter campaign in 2012, with the goal to provide an open-source platform to raise interest in their Epiphany chip architecture and start a software-ecosystem.

The Epiphany is an energy-efficient manycore architecture with floating point support. This makes it especially suitable for streaming applications like

radar signal processing or audio filtering. Current versions support up to 64 cores, and there are plans to create versions with up to 1024 cores [9].

The Parallella model P1602 used in this project consists of a Zynq Dual-Core ARM A9 processor with an integrated FPGA, manufactured by Xilinx, and the 16-core Epiphany chip E16G301 [10], together with 1 GB of external SDRAM, which is shared between the ARM processor and the Epiphany.

The ARM processor is the main processor of the system, and responsible for all I/O functions and for programming the Epiphany chip. It runs a Linux OS provided by Adapteva which includes the necessary driver and SDK to use the Epiphany chip.

The integrated FPGA is used for communication between the ARM core and the Epiphany chip. It allows direct write access from the ARM core to the memory of the Epiphany, and is also responsible for access to the shared DRAM from the Epiphany.

The Epiphany contains multiple RISC cores arranged in a grid and connected using a network on chip (NoC). The version used in this projects contains 16 cores, arranged in a four by four grid. Samples of an epiphany chip with 64 exist, and the architecture allows for up to 1024 cores on the same chip [9]. It is also possible to transparently connect multiple Epiphany chips, and use them in the same way as a single chip with the equivalent amount of cores.

Each individual processing core on the Epiphany consists of a eCore CPU s upporting the Epiphany Instruction set, 32 kB of local memory, a DMA engine and an interface to the network on chip.

The eCore CPU is a 32-bit RISC architecture which contains an integer arithmetic unit, a floating point unit, and a 64 word register file. It runs at a frequency of 600 MHz, has a variable length instruction pipeline and is able to issue two instructions simultaneously.

The DMA engine can be used to transfer data to other cores or to the shared DRAM. It works at the same frequency as the eCore CPU, and can transfer one 64-bit double word per second, enabling a sustained data transfer rate of 8 GB/sec.

The Epiphany is a shared memory multiprocessor. The local memory of each core is accessible using a global address, where the coordinates of the core are encoded in the upper 12 bits of the address. The upper 6 bits are used to encode the row, the lower 6 bits to encode the column. In addition, each core can access its local memory using a local address, where these upper 12 bits are set to zero.

To access memory on other cores, the memory accesses are done using the network-on-chip. The Epiphany contains three seperate network-on-chips, the rMesh for read requests, the cMesh for on-chip write requests, and the

xMesh for off-chip write requests. These architecture favoures write requests, since each read requires sending a read request and waiting for the answer, whereas writes only have to be sent tot the destination.

Transactions on the mesh are routed according to their destination. First, a packet gets routed east or west, and then, when it has arrived in the right column, it gets routed north and south, until it arrives at the destination node.

3

Parallella backend implementaion

This chapter describes the main part of this report, the design an implementation of a new backend for DAL with the capabilty to run DAL applications on the Parallella platform.

As mentioned in the background section about DAL, the DAL backend is responsible for providing the communication method between the individual processes. The next section will present the solution choosen for the Parallella backend.

3.1 Communication between processes

Communication between processes in DAL applications is done using point-to-point, one-way FIFO channels. These channels are setup in several stages. First, each channel is installed. This generally means that memory for the channel is allocated and necessary data structures are initialized. In a second round, channels are linked to the correspondending ports of the processes. The following sections will explain how this is achieved on the parallella.

3.1.1 Communication method

The FIFO channels are implemented using a simple ringbuffer, which is placed in a memory location accessible by all processes which use the channel.

3.1.2 Installation

Channels which are ending at a process running on the Epihany are placed in the local memory of the core where tise process is running. The reasoning for this is the much higher write speed of the NoC on the Epiphany [8]. These channels are installed by the wrapper generated for processes running on the Epiphany.

Communication from processes on the Epiphany to processes on the ARM is done using FIFOs allocated in the shared memory. The allocation and setup of these channels is done by the controller running on the ARM.

3.1.3 Linking of channels

After the channels are installed, they have to be linked to the corresponding ports of the processes. This can only be done at runtime because the same generated code is used for all active instances of a process, which means that the channel details are not yet known at the time of code generation.

On the Parallella, linking is done in two steps. First, each process is sent the channel ids of all its incoming and outgoing processes. This unqiue channel ids are then used, together with the application id to look up the actual location of the channel. A fixed location in shared memory is used to exchange the addresses of these channels.

3.2 Code generation

This section describes the process of generating an executable for an Epiphany core from the source code of a DAL process. Currently, there exists no operating system or library support for multihreading on a single Epiphany core, which led us to the decision to only allow mapping of a single DAL process onto one core.

The DAL Parallella backend will create a wrapper for each process which is responsible for communication with the controller based on the ARM processor and for initialisation of the incoming channels. To allow communication with the controller running on the ARM, a special FIFO is allocated. In addition, the code is linked with the libraries implementing the DAL communication functions.

The main problem during code generation is the limited amount of memory available. The existing benchmarks for the DAL framework often assume the availability of several megabytes of memory for a process, which is not possible on the Epiphany without using the shared memory for code and

data, which greatly slows down an application. The wrapper allocates as much variables as possible statically outside the main function, to help with the detection of memory issues during compilation.

3.3 Controller

This section will describe the controller architecture choosen for the Parallella backend. It is based on the controller for the backend for single processor Linux systems. On the Parallella platform, one slave controller is responsible for processes on the ARM, and one slave controller responsible for processes on the Epiphany was choosen.

The Epiphany controller is running on the ARM processor, and not on an Epiphany core. This is mainly due to the reason that a significant part of the work of a controller is the ability to start and stop the processes of an application. As the binaries for these processes have to be loaded from the filesystem of the Linux system, it makes sense to run the controller as a Linux thread, to allow it easy access to these binaries. An additional ability is the possibility to reuse code from the controller for the Linux backend.

The controller responsible for the Epiphany uses functions provided by the Epiphany SDK in version 5.13.09.10 to load the compiled code for a process into the memory of the corresponding epiphany core, and starting the processor.

It is also responsible for managing the shared memory and allocating space for channels leading from the Epiphany to the ARM, and for linking channels to the appropriate ports of processes.

After the processor has started, it is also responsible for sending control messages to the process. This is done using an extra FIFO channel installed for this purpose.

3.4 Limitations

The current implementation has several limits, some of which are due to limits of the Epiphany platform, while others could be improved with future work.

One of the main limitations of the Epiphany is the small amount of local memory available to each processor. Applications written for DAL often assume the availability of at least several megabytes of memory, which is not the case on the Epiphany. The only solution, except rewriting the application, is to place code and data in the shared memory. This is possible with

an adapted linker script, but reduces performance due to longer memory access times. A future version of the Epiphany SDK will include a software caching mechanism, which could help alleviate this problem.

The Epiphany SDK only includes a limited implementation of the C standard libraries, some functions (eg. erand48) are completely missing, while other functions are only implemented for use with an attached debugger (eg. printf). This leads to problems with DAL applications that need these functions.

The generation of DAL events is currently not implemented for processes running on the Epiphany. This could be implemented by installing an additional back channel for each process, which could be used to send an event message to the controller.

4 Evaluation

This chapter will present an evaluation of the backend implemented in this thesis. This was done using a FIR filter of order 15. The runtime of the filter is compared to the runtime of a filter running on the ARM core, using the Linux backend.

The FIR filter application contains 17 processes. Figure 4.1 shows an overview over the KPN. One process, the producer, is responsible for generation of the input data. It will send the generated tokens to the first stage of the actual filter, labeled F1. All stages of the filter are implemented using the same process source code.

Each filter process has two input and two output channels. During each execution of the fire function, the process reads one token each from all both incoming channels, and sends out one token each over all outgoing channels. The incoming token from input channel A is sent out unchanged over output channel A. It is also is multiplied with the filter coefficient, which is randomly generated during the process initialisation, added to the token from input channel B, and then sent out over output channel B.

4.1 Speed up

In theory, using the Epiphany should lead to a speedup of 15, due to 15 simultaneous calculations. The achieved speedup is far lower, as can be seen in figure 4.2. A posible explanation for this is the low amount of actual

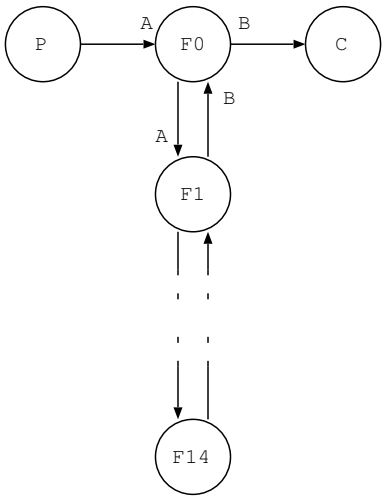

Figure 4.1: KPN for a FIR filter with order 14

calculation done in each process. Most of the time executing this application is spent transfering data between the cores. In addition to this, the transfer sizes are very small, which means that the ability of the Epiphany Network on Chip to transfer a double word in one clock cycle cannot really be exploited.

This evaluation shows that merely using additional processes does not automatically lead to a big speedup. Care has to be taken to also design the program in a way exploits parallelity.

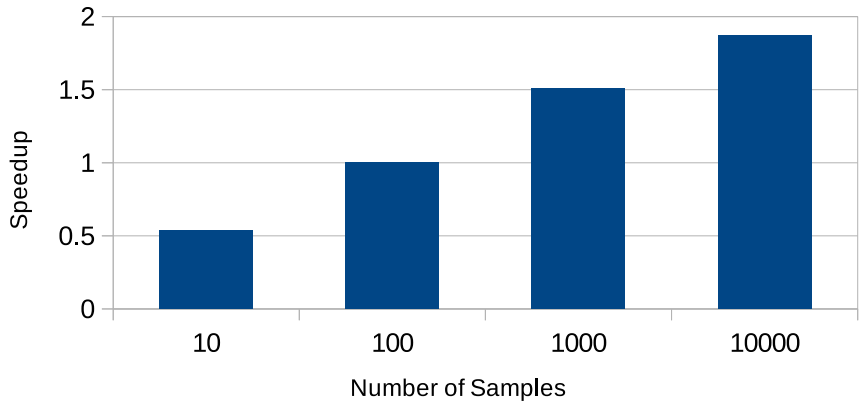

Figure 4.2: Achieved speedup for a FIR filter with order 14

5 Conclusion and Outlook

5.1 Conclusion

This semester thesis proposed to implement a DAL backend with the ability to execute DAL applications on the Epiphany chip. To achieve this, the code generation backend in DAL has been extended with the ability to create the necessary wrapper code to execute a single DAL process on an Epiphany core. Additional capabilities for the runtime controller have been developed, which allow it to controll processes on the Epiphany. The necessary libraries for communication between processes on the ARM and the Epiphany have also been added.

5.2 Outlook

There are several possibilities to improve and extend the current implementation.

- The current implementation of the FIFOs used to communicate between processes does not use the DMA engine. Depending on the size of the transfer, it could be worthwhile to use DMA to offload work from the Epiphany processor and to increase transfer speed.
- Investigate new features in the Epiphany SDK. The next version of the Epiphany SDK will provide software caching, where application code is

placed in shared memory, but often used functions are automatically cached in local memory. This could help with running applications with larger memory requirements on the Epiphany.

A

List of Acronyms

DAL Distributed Application Layer. III

KPN Kahn process network. 5

NoC network on chip. 7

B

Presentation Slides

Bibliography

[1] S. Savas, E. Gebrewahid, Z. Ul-Abdin, T. Nordstrom, and M. Yang, "An evaluation of code generation of dataflow languages on manycore architectures," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*. IEEE, 2014, pp. 1–9.

[2] A. Varghese, B. Edwards, G. Mitra, and A. P. Rendell, "Programming the adapteva epiphany 64-core network-on-chip coprocessor," *arXiv preprint arXiv:1410.8772*, 2014. [Online]. Available: http://arxiv.org/abs/1410.8772

[3] J. Pallister, S. J. Hollis, and J. Bennett, "Identifying compiler options to minimize energy consumption for embedded platforms," *The Computer Journal*, p. bxt129, 2013.

[4] K. Malvoni and J. Knezovic, "Are your passwords safe: energy-efficient bcrypt cracking with low-cost parallel hardware," in *WOOT'14 8th Usenix Workshop on Offensive Technologies Proceedings 23rd USENIX Security Symposium*, 2014.

[5] Z. Ul-Abdin, A. Ahlander, and B. Svensson, "Energy-efficient synthetic-aperture radarprocessing on a manycore architecture," in *Proceedings of the 42nd Annual International Conference on Parallel Processing (ICPP-2013)*, 2013.

[6] K. Gilles, "The semantics of a simple language for parallel programming," in *In Information Processing'74: Proceedings of the IFIP Congress*, vol. 74, 1974, pp. 471–475.

[7] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele, "Scenario-based design flow for mapping streaming applications onto on-chip many-core systems," in *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2012, pp. 71–80.

[8] Adapteva Inc., *Epiphany Architecture Reference*, March 2014, revision 14.03.11. [Online]. Available: http://adapteva.com/docs/epiphany_arch_ref.pdf

[9] A. Olofsson, "A 1024-core 70 gflop/w floating point manycore microprocessor," 2011.

[10] Adapteva Inc., *Parallella-1.x Reference Manual*, September 2014, REV 14.09.09. [Online]. Available: http://www.parallella.org/docs/parallella_manual.pdf