On the construction of operating systems for the Microgrid many-core architecture
van Tol, M.W.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
On the construction of operating systems for the Microgrid many-core architecture

Michiel W. van Tol
On the construction of operating systems for the Microgrid many-core architecture

Michiel W. van Tol
This work was carried out in the ASCI graduate school.
ASCI dissertation series number 269.

This research was supported by the University of Amsterdam as well as the European Union
under grant numbers IST-027611 (ÆTHER), FP7-215216 (Apple-CORE) and FP7-248828
(ADVANCE)

Copyright © 2012 by Michiel W. van Tol, Amsterdam, The Netherlands.

This work is licensed under the Creative Commons Attribution-Non-Commercial 3.0
Netherlands License. To view a copy of this license, visit the web page at:
http://creativecommons.org/licenses/by-nc/3.0/nl/, or send a letter to
Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Cover design by Michiel W. van Tol, inspired on the cover of Voer voor Hippopotamus, club
magazine of the Hippopotamus-Cric cricket club.

Typeset using LATEX, written using NEdit.
Printed and bound by Gildeprint Drukkerijen - Enschede, Netherlands

On the construction of operating systems for the Microgrid many-core architecture

Academisch Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus prof. dr. D.C. van den Boom
ten overstaan van een door het college voor promoties ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel
op dinsdag 5 februari 2013, te 14:00 uur

door

Michiel Willem van Tol

geboren te Amsterdam.
Promotor: Prof. dr. C.R. Jesshope

Overige Leden: Prof. dr. ir. H.J. Bos
Prof. dr. ing. S.J. Mullender
Prof. dr. A. Shafarenko
Prof. dr. ir. C.T.A.M. de Laat
Prof. dr. R.J. Meijer

Faculteit der Natuurwetenschappen, Wiskunde en Informatica
voor pappa
## Contents

### Preface

1 **Introduction**

1.1 Introduction ........................................... 3  
1.2 Context .................................................. 5  
1.3 Research Questions ................................. 7  
1.4 Contribution ........................................... 8  
1.5 Thesis structure .................................... 8

2 Operating Systems 11

2.1 Introduction ........................................... 11  
2.2 What is an operating system: A historical perspective 12  
  2.2.1 Early history ..................................... 12  
  2.2.2 1960s: Multiprogramming and time-sharing systems 12  
  2.2.3 Operating system abstractions ................... 13  
  2.2.4 Operating system design ........................... 14  
2.3 Hardware platforms: A future perspective ........ 16  
  2.3.1 Multi-core is here and now ..................... 16  
  2.3.2 Towards heterogeneous *many-cores* .......... 16  
  2.3.3 Memory and cache organization ................. 18  
  2.3.4 A case for hardware managed concurrency .... 19  
2.4 Operating systems: State of the art ................ 20  
  2.4.1 Early multiprocessor systems .................. 20  
  2.4.2 Loosely coupled multiprocessors ............... 21  
  2.4.3 Networks of workstations ...................... 23  
  2.4.4 Multi and many-core operating systems ....... 26  
  2.4.5 Commodity Operating Systems ................. 28  
2.5 Conclusion ........................................... 32

3 SVP 35

3.1 Introduction ........................................... 35  
3.2 Definition of SVP .................................... 36  
  3.2.1 Resources ....................................... 38  
  3.2.2 Communication and Synchronization .......... 38
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.3</td>
<td>Memory Model and Consistency</td>
<td>39</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Example</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>An SVP Many-core: The Microgrid</td>
<td>41</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Many simple fine-threaded cores</td>
<td>42</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Synchronizing registers</td>
<td>43</td>
</tr>
<tr>
<td>3.3.3</td>
<td>On-chip networks and memory</td>
<td>44</td>
</tr>
<tr>
<td>3.4</td>
<td>SVP on pthreads</td>
<td>46</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Implementation</td>
<td>46</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Efficiency</td>
<td>47</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Application</td>
<td>49</td>
</tr>
<tr>
<td>3.5</td>
<td>Extending SVP for Distributed Memory</td>
<td>50</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Distributed Environment in SVP</td>
<td>51</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Software Components</td>
<td>52</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Distributed Memory</td>
<td>53</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Extending μTC: Data description functions</td>
<td>54</td>
</tr>
<tr>
<td>3.5.5</td>
<td>Distributed SVP Prototype</td>
<td>55</td>
</tr>
<tr>
<td>3.6</td>
<td>Summary</td>
<td>56</td>
</tr>
<tr>
<td>4</td>
<td>Processes and Synchronization</td>
<td>57</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Processes &amp; Scheduling</td>
<td>57</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Defining a Process</td>
<td>58</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Fine grained scheduling</td>
<td>58</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Coarse grained scheduling</td>
<td>59</td>
</tr>
<tr>
<td>4.2.4</td>
<td>The problem of squeeze</td>
<td>60</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Pre-emption on the Microgrid</td>
<td>62</td>
</tr>
<tr>
<td>4.2.6</td>
<td>Space-sharing with SVP</td>
<td>64</td>
</tr>
<tr>
<td>4.2.7</td>
<td>System Services &amp; Identification</td>
<td>67</td>
</tr>
<tr>
<td>4.2.8</td>
<td>Summary</td>
<td>68</td>
</tr>
<tr>
<td>4.3</td>
<td>Synchronizations</td>
<td>68</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Exclusive places</td>
<td>69</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Spinlocks in SVP</td>
<td>71</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Suspend and resume with sear threads</td>
<td>72</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Suspend and resume with continuations</td>
<td>75</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Summary</td>
<td>76</td>
</tr>
<tr>
<td>5</td>
<td>Protection and Security</td>
<td>77</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>77</td>
</tr>
<tr>
<td>5.2</td>
<td>Protecting Computational Resources</td>
<td>78</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Applying capabilities to SVP</td>
<td>78</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Capabilities on the Microgrid</td>
<td>79</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Special case: System calls</td>
<td>80</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Configuration</td>
<td>82</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Summary</td>
<td>83</td>
</tr>
<tr>
<td>5.3</td>
<td>Memory Protection &amp; Management</td>
<td>83</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Concurrency granularity of protection</td>
<td>83</td>
</tr>
</tbody>
</table>
CONTENTS

5.3.2 Address space granularity .................................. 85
5.3.3 Managing protection .......................................... 87
5.3.4 Data communication and consistency ......................... 88
5.3.5 Memory management .......................................... 93
5.3.6 Summary .................................................... 94

6 Resource Management .............................................. 95
6.1 Introduction ..................................................... 95
6.2 Resource-oriented Resource Management ....................... 96
   6.2.1 Scalability ............................................... 97
   6.2.2 Examples ............................................... 98
   6.2.3 Evaluation ............................................ 100
   6.2.4 Summary ............................................. 110
6.3 Service-oriented Resource Management ......................... 110
   6.3.1 Protocol ............................................... 112
   6.3.2 The root SEP ........................................... 114
   6.3.3 Evaluation ............................................ 115
   6.3.4 Summary ............................................. 119
6.4 Discussion .................................................... 119

7 I/O Services .......................................................... 121
7.1 Introduction ..................................................... 121
7.2 Related Work .................................................. 122
7.3 I/O System Service ............................................. 122
   7.3.1 I/O Model (Synchronous and Asynchronous) ............. 123
   7.3.2 General Notes ......................................... 126
   7.3.3 I/O Places ............................................ 126
7.4 Microgrid I/O Implementation .................................. 126
   7.4.1 I/O Cores ............................................. 127
   7.4.2 High-Speed Bus ....................................... 128
   7.4.3 Bus Interface ......................................... 128
   7.4.4 Device Communication and Interrupt Handling .......... 129
   7.4.5 Synchronizing and Interrupts ......................... 129
   7.4.6 Memory Interface ..................................... 129
7.5 Evaluation ..................................................... 130
   7.5.1 Experimental Setup and Implementation ............... 130
   7.5.2 Overhead and DCA performance ....................... 131
   7.5.3 Throughput of I/O requests ........................... 132
   7.5.4 Contention of I/O requests ............................ 136
   7.5.5 Discussion ........................................... 139

8 Run-time system .................................................... 141
8.1 Introduction ................................................... 141
8.2 Introduction to S-Net ......................................... 142
   8.2.1 S-Net Example ........................................ 143
   8.2.2 Existing S-Net implementations ...................... 144
8.3 Resource Walker .............................................. 145
  8.3.1 Implementation ....................................... 147
8.4 Discussion .................................................. 149

9 Conclusion .................................................. 151
  9.1 Overview & Discussion ................................... 151
    9.1.1 Processes ............................................ 152
    9.1.2 Scheduling ........................................... 152
    9.1.3 Protection & Memory management .................... 153
    9.1.4 Synchronizations & IPC ............................... 153
    9.1.5 Resource Management & Run-times ................... 153
    9.1.6 I/O ................................................ 154
    9.1.7 Discussion .......................................... 154
  9.2 Conclusion ............................................... 156
  9.3 Future Work ............................................. 158

Samenvatting in het Nederlands .................................. 159

Acknowledgements .......................................... 161

General Bibliography ........................................ 165

Publications by CSA and affiliated project partners ........ 179

Publications by the author .................................. 185
Preface

Amsterdam, December 2012

I am writing this text on a Wang PC-240, a nearly 25-year old 286 machine. It was our first computer, which my father bought through a private-PC project of his employer, the Royal Dutch Airlines (KLM), where he worked as a programmer. We collected our machine on April 19th, 1988, I was 8 years old at the time, and I still have memories of sitting in the back of our car with a pile of big boxes. I am convinced that this was a turning point in my life, as without this machine, I would not have ended up where I am today; completing my PhD thesis on a topic in Computer Science. The machine sparked my enthusiasm and fascination for computers and computer hardware. I was amazed that we, humans, could build such a complex thing and make it work. As children often do with computers or other new technology, I quickly got to know the machine inside out and even started assisting my teachers in primary school when the first computers arrived there. The amount of time I have spent with this machine is unimaginable, playing games, writing school reports, making my own newspaper, and last but not least, teaching myself how to program. Writing this preface for my thesis on this machine expresses the special bond I have with it, it almost feels like meeting an old friend again, and I feel that it is a nice gesture that completes the circle.

It is amazing how much computer technology has moved forward in these 25 years. The 80286 chip in our old computer contains 134,000 transistors and has a performance of about 1.5 million instructions per second at its 10 MHz clock rate. Current state of the art computers run at 300 to 400 times this clock rate, have CPUs that consist of over 1 billion transistors and can process more than 100 billion instructions per second executing in multiple processor cores. Computers have become several orders of magnitude faster, as well as more complex. To continue this development, the current trend is to incorporate more and more processor cores on a single chip. This is where the research I was involved in at the Computer Systems Architecture group at the University of Amsterdam focused; the development of a new processor architecture, the Microgrid, composed of many fine-threaded processor cores.

I got in touch with my supervisor, Chris Jesshope in 2006, after being introduced to him by Andy Pimentel who I had been talking to about a proposal for a graduation project which involved dedicated processor cores for operating system tasks. Chris saw a good match between our interests and proposed I would investigate operating system issues for the Microgrid architecture he was working on with his group. During
my initial exploration phase where I got myself acquainted with the architecture, I discovered it had no approach for dealing with interrupts, exceptions or faults, a mechanism heavily relied on by operating systems. I completed the graduation project for my Doctoraal (MSc) with a proposed designs to implement exceptions in the architecture. However, the original question remained; how do we design an operating system for the Microgrid? I was really enjoying the type of work we were doing and liked the people I was working with in the group, and heard that there was a vacant PhD position. I decided to boldly ask Chris if I would be considered for this position, and after the work delivered for my graduation project he was happy to take me on board to continue to pursue answers to our original question.

I started my PhD project in February 2007, and it was the beginning of a long and complex journey. Our research efforts were diverted and distracted in order to meet requirements for European projects, and real operating system development was hampered by the ever changing architecture details and lack of a working compiler. Therefore, I did not reach the end result that I had hoped for at the start of my PhD, really delivering an implementation of a small initial operating system for the Microgrid. Instead, I present a blueprint of what a Microgrid operating system would have to look like, exploring the issues at a more abstract level and by using simulated or emulated environments. I am happy that I managed to draw the complete picture of an operating system, and feel this was the best I could achieve in this situation. I hope that this thesis will be an interesting read for other people that are working on many-core architectures and/or many-core operating systems. For me, it was certainly a period in which I have learned a lot about a broad collection of topics and research directions, and have met and interacted with many interesting people all over Europe and even beyond.

Michiel W. van Tol.
CHAPTER 1

Introduction

1.1 Introduction

Market demand has been pushing computer technology forward for many decades, as it has always been hungry for faster and more efficient systems. Semiconductor technology has been keeping up with this, scaling up every one or two years following the observation made by Gordon E. Moore in the mid-1960s [111], commonly known as "Moore's law". This technology scaling has allowed us to put more and more components onto a single chip, and to increase the frequencies at which they run. This increase in both complexity and switching frequency of digital circuits has lead to a steady performance improvement for each new generation of the most important part of the computer: the central processing unit (CPU), or processor, which is responsible for executing programs.

However, a point has been reached where gaining performance improvements by adding more logic components to as single processor has stagnated, and where power consumption limits further performance improvements by frequency increase. In other words, we are unable to make the processor any faster, as when we further increase the frequency it will consume too much power and therefore generate too much heat. This is a problem across the whole spectrum of computing; it affects battery life for mobile appliances, cooling requirements for desktop computers, and the cost for both cooling and power consumption for large data centers is becoming the most dominant cost factor.

As technology scaling still allows us to put more components on the chip, a move has been made towards putting multiple processors, or processor cores onto a single chip. Most multi-core processors that are currently on the consumer market today contain two or four cores, but also chips with 16 cores or even more are already available for specific application domains. The trend we see here is that the number of cores is likely to further increase and in the future we might have chip designs with hundreds or thousands of cores, which we refer to as many-core architectures.

An important observation that must be made is that having multiple cores in a chip will not magically make a single application run faster. The processor can then do
multiple sequential tasks at the same time, but each task itself will still be processed at the same maximum speed of a single core. This problem can be illustrated with an example using queues at the cashiers in a supermarket.

Imagine yourself at the supermarket, you have just gathered your groceries and proceed to queue up at the cashiers for checkout. If there are one hundred customers in front of you that also want to check out and pay for their groceries, it is clear that it will take you a long time before you are finished and will be able to go home. However, if several other cashiers arrive and open their stations, the customers will spread and your queue will be processed more quickly. This shows how multiple cores can process more applications (customers) in less time, as it increases the overall throughput.

Now imagine another situation, where you arrive as the only customer at the cashiers. Then, it does not matter if there are one, five, or a hundred cashiers available\footnote{Please note that this example assumes that all cashiers work at the same speed, which is highly unlikely to be observed when shopping at a real supermarket.} to check out your groceries, it will still take you the same amount of time to pay for your items. This illustrates the other observation about multi-cores, where a single application is not processed faster, regardless of the number of cores available.

A solution to this problem is to decompose programs into smaller activities that can execute in parallel. To complement our cashier analogy, this would mean spreading the products you buy over multiple cashiers with the help of some friends so that you’re finished quicker, assuming they are available. However, finding such parallelism is not always trivial, as certain activities will depend on the result of other activities. Also distributing the products to your friends and gathering and settling the bills afterward takes time, so you will only manage to be done quicker if this overhead takes less time than the time you gain from using multiple cashiers at once. Parallel programs suffer from exactly the same problem; the overhead of starting a parallel task should be sufficiently cheap compared to the amount of time you can gain from executing the tasks in parallel.

In an attempt to make handling such parallelism as cheap and efficient as possible which enables programs to better exploit the throughput of a many-core processor, a new many-core processor design was developed at the University of Amsterdam, that we called the Microgrid. This design incorporates special mechanisms in the hardware for handling parallelism; for example to divide small tasks over processor cores and to wait for them to complete. This is a different approach compared to traditional contemporary systems which implement this in the software running on the cores, often as a part of the operating system.

When designing such a different approach in a hardware architecture, it is important that there is a software environment that can exploit these new features. If the software is not able to do this, there is no added value from having such a different hardware design. In the case of managing parallelism, one important affected software component is the operating system, as it sits between the applications and the hardware and, as the name suggests, it operates, or in other words manages, the machine.

An operating system can be compared with a bus driver, where the bus is the computer and the passengers that travel with the bus are the applications. The bus
1.2. CONTEXT

The work covered by this thesis is based on the aforementioned Microgrid many-core architecture, that has been under investigation and development at the Computer Systems Architecture group at the University of Amsterdam since 2005. The goal is to develop it as a general-purpose many-core architecture for mainstream computing. In this section we will give some background on this research as it forms the context in which the research covered by this thesis has been performed.

The Microgrid architecture originated from the early work on Microthreading [CSA4, CSA23] which focused on offering an alternative for speculative out of order execution in superscalar processors. It consists of using fine grained multithreading with simple in order processors to increase pipeline efficiency as an alternative for the complex support for out of order execution. Dynamic scheduling of fine grained code fragments, or Microthreads, allows latency hiding when implemented efficiently in hardware. The model consists of an extension of a RISC instruction set with special instructions to start and manage threads.

It became quickly clear that these code fragments could also be distributed over multiple computing pipelines using the same mechanisms and interfaces, which allows to yield more throughput, and the first approaches for chip multiprocessors using Microthreading were suggested [CSA16, CSA5]. This research then evolved into the development of the Microgrid [CSA17, CSA1], as a many-core architecture that focuses on scalability and that uses the concepts and mechanisms of Microthreading for scheduling computations both inside and across cores.

Several important steps were made as part of the ÆETHER project\(^2\), such as further development of the Microgrid architecture [MWvT2], and its system level simulator [CSA29], but also a C-like system level language, µTC [CSA19] (an acronym for microThreaded C), was developed within ÆETHER and the NWO Microgrids projects.

\(^2\)ÆETHER was an EU funded FP-6 ACA/FET project, number IST-027611, that partially funded the research behind this thesis. See http://www.aether-ist.org
CHAPTER 1. INTRODUCTION

to provide a programming interface. Before this language existed, machine instructions had to be programmed directly using assembly, originally using extensions to the MIPS instruction set, but around the start of the Microgrid work a switch to the Alpha instruction set was made. This project marked the start of the development of a compiler [CSA2] for the $\mu$TC language, which would ease the development of software for the architecture, but which turned out to be a much more difficult development than was anticipated.

The general theme of the ÆTHER project was to design future self-organizing ubiquitous computing systems which consisted of networked self organizing elements, which the project named as the Self-Adaptive Networked Element or SANE. The Microgrid was part of this project as one of the possible implementations of a SANE, but others were developed as well, for example based on reconfigurable FPGA-based accelerators [CSA36, CSA28]. The goal was that these SANEs and a computing system combined of these could all be programmed using the same concurrency model to manage and delegate work. This model, the SANE Virtual Processor, or SVP\(^3\), was derived as an abstract model from the concurrency model of the Microthreaded processors, exhibiting the same constructs and pattern, analogous to the API that was defined in the $\mu$TC language. While SVP served as an abstraction layer for programming the ÆTHER platform, higher level languages such as the S-Net concurrent co-ordination language [CSA14, CSA12] were designed to support a modular based programming approach.

The development of the SVP model as an abstract model for the execution model of the Microgrid is important for this thesis for two reasons. First of all, it allowed us to start constructing system components based on the model while the architectural details of the Microgrid were still under development and subject to change. Secondly, it gives us an abstract model for programming many-cores which embed concurrency management at the heart of the system, and makes the work described in thesis more widely applicable. Effectively, the Microgrid becomes one of the possible (hardware) implementations based on the SVP model, but we can also implement SVP (in software) on top of other systems. One example of such other systems are the aforementioned other SANE implementations [MWvT14], clustered multi-core environments [MWvT13], but we have also demonstrated SVP on other many-core processors such as the experimental 48-core Intel SCC processor [MWvT1].

The Apple-CORE project\(^4\) followed up on the work started in ÆTHER, but fully focused on the Microgrid architecture [MWvT5]. Within this project the simulation platform was further developed, an FPGA hardware prototype was made of a SPARC core with the Microthread extensions [CSA7, CSA42]. The compiler developments started in ÆTHER continued but faced many difficulties which lead to an alternative compilation toolchain from a conceptually equivalent, but syntactically different, language named SL [CSA37], which has a more explicit syntax that corresponds with the hardware mechanisms in the Microgrid. A compilation strategy for this language based on macro expansion and post processing allowed for reuse of

\(^3\)Please note that the meaning of the SVP acronym was later changed to System Virtualization Platform, for more information visit http://svp-home.org/

\(^4\)Apple-CORE was an EU funded FP-7 STReP Project, number FP7-215216, that partially funded the research behind this thesis. See http://www.apple-core.info
existing compilers, without requiring heavy modification of the compiler itself as was the case with μTC. This provided the long-needed C-like interface to program the Microgrid simulations, and included a port of many of the C standard library functions. This was then used to implement some initial operating system components, such as a simple resource allocator, memory allocator, and the prototype of an I/O system using the I/O interface that was also proposed [MWvT3] in the project.

Other important directions in the Apple-CORE project were to develop compilers and programming systems that can exploit the massive amount of concurrency that is supported by the Microgrid architecture. This was done with work on a parallelizing C compiler [CSA39, CSA40] and on the compilation of SAC [CSA13, CSA10, CSA41, CSA15], a high level data-parallel functional programming language. Both these compilers target the SL language and its toolchain as an intermediate language, and do not compile directly down to the Microgrid architecture.

1.3 Research Questions

While the Microgrid design has been evaluated several times using single computational kernels or small benchmarks that show the advantages of the architecture in [CSA5, CSA22, CSA42] and [MWvT2, MWvT5], the question remains if it can be used efficiently as a general purpose computing platform. General purpose computing means running different (possibly interactive) applications at the same time that are not known statically at design time, and which requires an operating system to manage the machine. This in turn requires a strategy for the design of operating systems for the Microgrid, which is not straightforward. The differences in the architecture, such as its memory consistency model and the lack of atomic operations, prevents us from using off the shelf operating systems without requiring heavy modifications; if it is possible at all.

The challenge of designing an operating system for many-core systems is a general challenge at the moment and applies to a much wider audience than to people interested in the Microgrid. During the research for this thesis, several other many-core operating system projects have emerged [135, 104, 159] that observed many similar problems and proposed similar solutions to what we will discuss here. However, our challenge is still slightly different due to our exotic hardware architecture, which raises the following research questions about the Microgrid:

1. How is the design and structure of an operating system affected when we move the support for concurrency management from software to hardware, such as on the Microgrid?

2. Is it possible and feasible to have one uniform mechanism for handling concurrency at all levels in a system?

3. What are the missing critical features in the current Microgrid design to support a general purpose operating system?

4. How should a general-purpose operating system for the Microgrid be structured, and is the architecture suitable for general-purpose use?
With our work on generalizing the Microgrid concurrency model into SVP, we can also generalize our view to other platforms by applying SVP to them. In theory this means that we can re-use the approaches we define for an SVP based system for the Microgrid, but is this feasible? We investigate the use of SVP with the two following research questions:

1. Does SVP provide the right abstractions on which we can build operating and run-time systems, and can we use SVP as an abstraction to design an operating system for the Microgrid?

2. Similar to the question for the Microgrid, is SVP suitable for general-purpose computing use?

1.4 Contribution

The main contribution of this thesis is an analysis of the feasibility of constructing a general-purpose operating system for the Microgrid architecture and other platforms that are based on the SVP concurrency model. This is done by identifying the most important building blocks of such a general-purpose operating system, and then showing how these can be either constructed in SVP or on the Microgrid specifically. During this analysis, we identify which functionality is currently missing from the Microgrid design and, where possible, what we propose as solutions to this. Using the result of this analysis we then reflect on the feasibility of using the Microgrid architecture and the SVP model for general-purpose computing.

While we focus on the underlying mechanisms and do not present and evaluate a complete operating system design, we do look ahead how such an operating system is likely to be be structured, following from the design of the mechanisms. We also present a preliminary implementation and evaluation of two operating system services, namely a resource manager that manages core allocation and an I/O service for managing I/O requests to devices. The latter also includes a technical contribution in the design of an I/O interface for the Microgrid architecture.

Further technical contributions from this thesis lie in the design of a software SVP run-time system which has been used to prototype SVP based systems and several mechanisms described in this thesis. This run-time is also being used in the development of a new event driven high-level simulation of the Microgrid. We also present a further extension of this run-time to apply SVP to distributed memory architectures. Finally, we propose a novel run-time for the S-Net co-ordination language that can be constructed using SVP, and that is able to exploit both the amount of concurrency supported by the Microgrid and our proposed resource management mechanisms.

1.5 Thesis structure

This thesis is structured into nine chapters, where chapters 3 through 8 cover the contributions of this thesis. In Chapter 2 we start with a historical perspective of operating systems, why we have them and how they came into existence. This then
extends into a discussion that identifies a set of basic mechanisms required to build a general purpose operating system. The chapter then continues with a discussion on what to expect from future many-core architectures and why we think it is interesting to have concurrency management support in hardware. As future many-cores are likely to have a more distributed organization, and require similar operating system organizations for scalability, we discuss several historical and state of the art operating systems for distributed systems, parallel systems and multi/many-core systems.

In Chapter 3 we introduce the SVP model which we mainly use throughout the thesis as an abstraction of the architectures that we want to target. In this chapter we also discuss several implementations of SVP, such as the Microgrid architecture and a software based SVP run-time. We also introduce an extension to SVP to apply it to distributed memory systems, and discuss the extension to our software based SVP implementation to support this.

In Chapter 4 we start analyzing how to construct the most basic mechanisms of an operating system based on top of SVP; identifying a process, scheduling processes, synchronization between processes and accessing a system service. We show how this can be approached from SVP, but also which specific issues there are with implementing these on the Microgrid. We then continue in Chapter 5 with defining an approach for protection and memory management, and combining this with the synchronizations of Chapter 4 we show how we can implement inter process communication. One of the important protection mechanisms we present is a mechanism to safely access system services by providing fixed entry-points.

The first example of a system service that we will discuss in a bit more detail is the Resource Manager in Chapter 6. Here we propose and discuss two approaches to resource management: the first approach is designed for the Microgrid and allocates general-purpose cores to applications, similar to memory allocations, while the second approach is geared towards the opportunistic collaborative networks of the ÆTHER project where resources expose their functionality, and functionality is requested and allocated instead of general-purpose resources. Prototype implementations of both approaches are presented and using some experiments we show how they behave.

Chapter 7 explores a second system service; one that makes up the I/O system. We discuss in detail how such a service can be constructed on top of SVP, to handle and queue I/O requests for devices, but we also propose an underlying I/O interface for the Microgrid architecture. A prototype implementation of this is used to make a preliminary performance evaluation.

While we have already constructed all identified operating system components, we also propose an approach for a run-time system for the S-Net co-ordination language in Chapter 8, that could be run on top of such a system. This run-time system approach was designed to use the resource manager of Chapter 6, and the initial proof of concept implementation of it was used in that chapter at the evaluation. In this Chapter however, we discuss the details of the proposed design.

We end this thesis with a discussion and conclusion on the feasibility of construction a general-purpose operating system for the Microgrid and other SVP based systems in Chapter 9. We reflect on the problems we encountered and the solutions that we proposed during the previous chapters.
CHAPTER 2

Operating Systems

2.1 Introduction

This chapter covers the background and state of the art to lay a foundation for the rest of this thesis. This background information is divided over three parts. We start off with looking at the history of operating systems in order to understand what an operating system is, and what it is designed to do as they developed over time. Then, we identify the fundamental low level abstractions that an operating system is required to provide, which we will use throughout this thesis as a target of what we try to achieve. We conclude our historical perspective with an overview of the main types of operating system (kernel) designs.

After having formed our idea of the minimal set of functions we have to implement, we move over to the hardware side in the second part of this chapter to identify on what kind of hardware architectures we have to apply these ideas. We show why we are moving towards more and more cores on a single chip, and try to give an outlook of what such systems might look like taking recent many-core research into account. We observe that future many-core systems are likely to have a more distributed memory organization, can be heterogeneous, and that there is a case to be made to investigate hardware supported concurrency management.

This outlook to future hardware platforms gives us the context in which we want to look at operating systems, and this is what we do in the third part of this chapter. As we have observed a move towards, and a requirement for, more distributed organizations, we review several pioneering parallel and distributed operating systems. Then, we discuss several recent designs in the field of multi- and many-core operating systems. We conclude the third part with a discussion on contemporary systems, and the problems they face in scaling to many-core architectures. We also show how they have been used to scale up cluster architectures and what kind of programming systems have been proposed to utilize these.
2.2 What is an operating system: A historical perspective

2.2.1 Early history

At the dawn of the computer age in the 1940s, computers were expensive and consisted of very large room-sized installations. When programmers would want to run a program on such a machine, they would get a time slot allocated, go to the computer room at the given time and have the whole machine for themselves. They would have full control over the entire machine, input their program, and receive the results as soon as the machine produced them. This situation started to change in the 1950s, at the same time the first high-level programming languages such as FORTRAN, Lisp and COBOL started to emerge, accompanied by their respective compilers and interpreters. Libraries were developed for often used functions, but also for controlling external devices, so that programmers would not be required to go through the cumbersome process of writing complicated low level device control code for each of their programs. Programming became easier and more people started to compete for computer time. In order to use the expensive computer time as efficiently as possible, the computer would be controlled by a dedicated person, the operator, who would accept input jobs (usually on paper card decks or paper tape) from the programmers, run them on the computer and deliver back the output.

The first pieces of software that could be called an operating system were to automate, and therefore speed up, the process done by the operator; to load and run a job, store or print the output, and prepare the machine again to proceed to the next job. These first operating systems, or batch processing systems, emerged in the mid 1950s; the director tape [128, 127] was developed at MIT to automatically control their WhirlWind system, and around the same time a job batching system was developed at General Motors Research Laboratories for the IBM 701 and 704, which was then integrated with the first FORTRAN compiler from IBM [150].

2.2.2 1960s: Multiprogramming and time-sharing systems

These batch processing systems were further expanded in the 1960s into multiprogramming systems, which could run multiple batch jobs at the same time. The main reason for this was that a computer, and therefore its CPU time, was expensive. It could be used much more efficiently by running multiple jobs and switching between them when one has to wait on a slow external unit, for example by using the (otherwise idle) cycles of the processor for a different task while it is waiting on an I/O operation. This was the first time concurrency and parallelism\(^1\) were introduced in systems, posing new challenges; “The tendency towards increased parallelism in computers is noted. Exploitation of this parallelism presents a number of new problems in machine design and in programming systems” was stated in the abstract of a 1959 paper [38] on a

\(^1\)We separate the concepts of concurrency and parallelism as follows: Concurrency, the relaxation of scheduling constraints in programs, and parallelism, the simultaneous execution of operations in time which is made possible by the existence of concurrency.
The downside of the batch and multiprogramming systems was the long turnaround time of running an individual job. The programmers could no longer interactively debug their programs as they could years before when they had the whole machine for themselves, but instead would have to wait for hours or days before the output of their job was returned to them. With the invention of a new memory technology (core memory), memories became cheaper and therefore larger, allowing multiple programs to be loaded into the system memory at the same time. To increase the productivity and efficiency of the programmers, interactive or time-sharing systems were designed [40] where multiple users would be interactively connected to the system executing commands, compiling and running jobs and getting feedback from the system immediately. This required additional techniques to safely share the resources of the system, for example, using memory protection. CPU sharing was achieved by time-slicing the execution of programs, switching between them multiple times per second, giving the illusion of simultaneous execution to the users. The development of these systems also lead to improved hardware features in the mid 1960s to support them, such as virtual memory and complete machine virtualization [42].

One of the early time-sharing systems was Multics [41], which lead to the development of Unix [124], which together laid the foundations for the principles and design of many operating systems that are currently in use today. Multics was envisioned to be used as a utility computing facility, similar to telephone and electricity services. Therefore it was designed to run continuously and reliably, and to service a lot of users. In order to support this it employed many techniques that are still ubiquitous in systems today, such as virtual memory, paging, dynamic linking and processes [49, 45], multiprocessor support, security, hot-swapping and much more.

2.2.3 Operating system abstractions

These operating systems laid the foundations for the most important concepts and abstractions used in the operating systems of today. Many new developments have been made in operating systems since then, for example networking and integration with Internet technology, as well as graphical user interfaces. However, even though some of these new developments have been deeply embedded into several modern operating systems, the core functionality of these systems did not change. These developments can be seen as systems that build on top of the original foundations, and that might make use of specific I/O devices such as network adapters or graphics cards.

Taking the history of operating system into consideration, we identify the basic definition and core purpose of an operating system as follows:

- An operating system gives a uniform abstraction of resources in a machine to programs running on top of it using well defined interfaces. This allows for portability; a program can run on an other platform without having to be aware of the differences in underlying hardware.
• An operating system arbitrates the sharing of resources between multiple programs. These can be from multiple users, and such arbitration includes things like scheduling, but also protection; assuring that programs are not able to functionally influence each other when this is not desirable, or to restrict their access to certain devices or data.

In this thesis we will focus on these two core purposes and how they can be achieved on future many-core systems by implementing the fundamental abstractions an operating system needs to provide. Other features one would find in modern operating systems, such as networking, filesystems and user interfaces, are what we consider higher level features that are not covered in this thesis. We assume that support for these fundamental abstractions will provide a sufficient foundation to build more complex systems providing such features, some as a part of the operating system services, and some as applications.

In order to enumerate the required abstractions that an operating system needs to support, we take the definition given by Bergstra [15] as a starting point. We then generalize it to the following list of required functionality:

• Loading and starting the execution of a process
• Scheduling of processes and allocation of execution resources
• Address space and memory management
• Managing input/output to/from and control of devices
• Prevention of interference between processes
• Inter-process communication (IPC)

Please note that we condense several separate points in Bergstra’s original definition that cover networks, secondary storage, I/O and control into one generic I/O requirement as this in essence is all a communication with some external device. We add the requirement for Inter-process communication which is missing from Bergstra’s definition, but which is required to build higher level features such as system services. IPC is required for system services in order to be able to communicate with them. Having I/O and IPC is enough for example to implement a file system service, as this effectively delivers an abstraction of the data stored on the I/O device, so this is not a separate required functionality in our analysis.

2.2.4 Operating system design

The early operating systems were monolithic; they consisted of a single large program taking care of everything. One of the first people to isolate the abstractions mentioned earlier was Dijkstra in a highly influential paper on his design of the THE multiprogramming system [50]. He split the operating system in different layers of abstraction, each layer adding more functionality. For example, the lowest layer, layer 0, virtualizes processor resources abstracting the number of processors and processes in the system, layer 1 virtualized memory with demand paging from disk, and layers
2.2. WHAT IS AN OPERATING SYSTEM: A HISTORICAL PERSPECTIVE

2 and 3 dealt with abstraction of I/O. Such a *layered* operating system design is still commonly used in current *monolithic* systems, such as Linux\(^2\).

Another influential design was by Hansen [71], which identified the *nucleus* of an operating system, to provide the aforementioned set of fundamental abstractions. His goal was to identify and implement the core set of functionality required for an operating system, so that this system would be flexible enough to be used for running batch-, multiprogrammed-, or time-sharing systems. It supported hierarchical creation of parallel processes that could communicate with each other with messages. Using these mechanisms, it could host multiple of such operating systems hierarchically, laying the foundations for the later *microkernel* approaches. In fact, his concept of a *nucleus* of an operating system, is what we still know as a *kernel* of an operating system today.

One of the first generation *microkernels* was Mach [3], which tried to solve the increasing complexity of the UNIX kernel and its interfaces. It had an extensible kernel which provided a low level abstraction, and additional functionality was moved out of the kernel into services running as separate processes in *user space*. It supported *inter process communication* (IPC) using capabilities [49] for security, and processes with multiple threads which shared ports to listen for IPC messages. The project was later adopted by IBM [58] in an attempt to unify the core of all their operating systems, where a compatibility layer, an OS personality, would run on top of the kernel to provide the original interfaces of the emulated systems such as OS/2, Windows NT or AIX. Multiple OS personalities could be run simultaneously allowing applications developed for different systems to run side by side on the same machine.

The problem with Mach and other first generation microkernels was the performance of IPC operations. As these systems are designed to have most services run in user-space processes, this heavily impacted system performance. The second generation of microkernels greatly improved on this, where Liedtke showed that IPC can be cheap [100] and microkernels can and should be small [102], reducing overhead and cache penalty. His L4 kernel [101, 103] served as a basis of many third generation microkernels, where the emphasis lies on formal verification of the kernel, for example as in seL4 [88]. Another noteworthy microkernel based operating system is MINIX-3 [76], which emphasizes on reliability [143], introducing a reincarnation server that monitors user-space services and restarts them when a problem is detected.

The *exokernel* approach [54] takes the step of reducing the responsibility of the kernel even further than microkernels do. It provides no abstractions other than exposing the bare hardware in a secure way, arbitrating it between applications. Each application uses a stack of libraries, which can itself implement a library operating system, to provide its own abstractions such as device drivers, file systems and memory management. As all the functionality of the hardware is exposed to the applications, this allows for specific optimizations in tasks the kernel or a system service would normally do, such as special policies for page management or scheduling.

Another special case of operating systems are *hypervisors* that manage hardware virtualization for virtual machine monitors, which have had a long commercial history since VM/370 [42]. They share a lot of similarities with both the microkernel and exokernel approaches as they only form a thin layer to share the resources of

\(^2\)For more information on the Linux kernel, see: [http://www.kernel.org](http://www.kernel.org)
the hardware between multiple virtual machine instances. However, there is quite some variation in their design and structure; for example, a virtual machine monitor can be hosted within another operating system, or run its hypervisor directly on the hardware. Also the type and techniques of virtualization varies, VM/370 for example, provided an indistinguishable virtualization of the hardware, where unmodified operating systems or even other instances of VM/370 could be run hierarchically. In contrast, early versions of Xen [12] were paravirtualized where the guest operating system needed to be aware of running under Xen for better performance, while VMware used binary translation [4] to overcome problems in virtualizing the x86 platform. Both use a hypervisor running on the bare hardware, managing several virtual machine monitors, resembling a microkernel running different operating system personalities, but with slightly different abstractions between the layers.

Since virtualization has been enabled on the widespread x86 architecture, virtual machines have gained a lot of popularity in the last decade as they are attractive for providing strong isolation and security which enables hardware consolidation. This popularity could also be seen as a hint that current common operating systems were not able to provide these features sufficiently. The rise of virtualization was an enabler for the development of cloud computing which can provide on demand general purpose computing with virtual machines [90]. This, besides the grid computing initiatives in the scientific computing community in the last 15 years, has finally realized the utility computing that Multics was originally designed for.

2.3 Hardware platforms: A future perspective

In this section we discuss the trends and perspectives of future hardware platforms, where is it going and why, and try to sketch what these systems might look like.

2.3.1 Multi-core is here and now

It is clear that the easy increase in single core performance by clock scaling has stagnated due to power and thermal limits, and there is a shift towards chip multi-processors. This can be observed in the recent developments in the general purpose processor market where all major vendors have introduced multi-core chips supporting multiple threads per core [130, 29, 136, 158]. As Moore’s law [111] is still going strong, delivering more and more transistors with every generation, there is an incentive to spend this transistor budget on developing chip multi-processors. Multiple cores at a lower clock rate, and therefore lower voltage, can deliver more throughput than a fast clocked single core within the same power envelope, and chip multi-threading can be used to further increase throughput by hiding memory access latencies [117].

2.3.2 Towards heterogeneous many-cores

Intel Labs has already shown that they can integrate 80 simple cores on one chip [154], and have recently delivered a second research chip, the Intel Single Chip Cloud Computer, or the Intel SCC, that integrates 48 general purpose Pentium (P54C) cores on a mesh network on a single die [81]. Such approaches could have the potential to scale
out to hundreds, thousands or perhaps even more cores on a single chip [22, 11], but what the many-core landscape will exactly look like in contrast to current multi-cores has not been clearly decided yet. For example, we could already fit over 10000 very simple cores on a single chip [69], but it is unlikely that we would manage to get any performance out of such a system besides for very specific embarrassingly parallel applications. Therefore, we can not expect to use this for general purpose computing, and the correct balance between number of cores and their individual complexity needs to be found.

Homogeneous many-core architectures are easier to design, implement and verify than a heterogeneous many-core due to their regular structure. They are easier to program, as on a heterogeneous platform the programmer needs to worry about how to partition his problem optimally between the different types of cores. This problem is illustrated by the amount of effort people currently invest to program GPUs to heavily speed up data-parallel computing.

There are two problems with homogeneous many-cores, which limit their scalability. First of all, when we have more cores running on lower frequencies, the throughput of a single core is reduced. This means that sequential parts of a program will not benefit from such an architecture. However, this affects also the performance of the whole application as it will not scale well enough being limited by the performance of its sequential parts, as observed by Amdahl’s law [8]. One of the proposed solutions [138] to this is to have a heterogeneous design which consists of a few large, or fat, cores which have a high single core throughput for executing the sequential parts, and many small simple cores for a high parallel throughput.

The second problem with a homogeneous many-core is that when we try to maximize overall computing throughput by executing multiple applications at the same time, the throughput scalability will be limited by power constraints [55]. To illustrate this; the majority of a chip is currently used as cache memory, which only consumes 10% of the power compared to the logic of an active core of the same area [23]. Therefore, simply replacing the cache with core logic would not fit within the same power envelope, so more energy efficient designs are required.

The alternative is to use more specialized and energy efficient logic for specific operations. One of the candidates for such embedded accelerators is the GPU, and recently products have emerged in the main-stream computing market that embed the GPU into the processor. Another options is to have embedded FPGA fabric which can specialize for specific functions at run-time. The IBM Cell architecture [66] is a good example of a heterogeneous multi-core architecture which embeds accelerators, and it is a common practice in the embedded systems domain which for example often uses dedicated DSPs or ASICs.

Further heterogeneity will be introduced by fine grained frequency and voltage scaling with cores running in different frequency and voltage domains. Adaptive scaling can be used as to overcome some of the aforementioned problems; frequency can be raised temporarily on a core to overcome Amdahl’s law [9, 33], and both frequency and voltage can be scaled to stay within the power and thermal budget of the chip.

One thing that is certain is that these systems need to deal with a large amount of parallelism and asynchrony, with many different frequency/voltage domains that
adaptively switch between different speeds and voltages at run-time. They will have *dark silicon* [55]; parts of the chip that are powered off, either due to the power budget, the limited amount of exposed concurrency, or because they contain specific functionality that is not in use at a given moment.

### 2.3.3 Memory and cache organization

An important challenge for future many-core architectures is the memory organization. While we can increase the computing throughput of the processor with multiple cores, we require a memory system that can provide enough bandwidth to support this. Modern multi-core processors [130, 136] already pack on-chip memory controllers with four external memory channels, but increasing this number is very difficult. We hit physical limitations as we reach a point where we can no longer increase the pin count of the CPU packaging.

An interesting innovative technique to overcome the memory bandwidth problem is 3D stacking [149], where DRAM chips are stacked on top of the processor with many vertical connections through silicon. Then, it becomes possible to provide each core with its own locally stacked high bandwidth memory [105], turning these systems into on-chip distributed memory machines. This distributed nature leads us to the second major challenge for multi/many-cores in the area of memory and cache organization: cache coherence [11].

It is attractive to preserve a shared memory view for the ease of programming on many-cores, but it will be difficult, and therefore unlikely, to support *sequential consistency* [95], as this would require an enormous amount of coherence traffic across the chip. From experiences in the supercomputer community we know that it is feasible to implement cache coherent distributed shared memory systems with many nodes using directory based protocols, however using a relaxed consistency model [2]. Another interesting approach is the Cache-Only Memory Architecture, *COMA* [68, 44], where all on-chip memory resources are combined to form a migrating memory system.

The efficiency of a cache coherence mechanism heavily depends on the scalability of on-chip networks [91] in terms of both latency and bandwidth, but also energy consumption. For example, a packet switched mesh network is efficient for local communications, and very attractive for tile-based designs. However, it has a large communication cost in terms of latency and power over larger distances across the chip due to the number of hops that have to be taken, and the more central nodes might suffer a lot from network contention. One promising future technology that could mitigate this problem is an on-chip nanophotonic network, which can be used as a second level network spanning larger distances across the chip. It can provide a solution that is both a low power and high bandwidth for chip-level cache coherence [93].

The alternative is not to spend our transistor budget on complex cache coherency hardware and have a distributed memory organization instead. The Intel SCC for example has off-chip shared memory and on-chip message passing support [81], but no cache coherency in hardware. There is a case to be made for solving the coherency problem in software instead of relying completely on hardware [74, 168]. This builds
on the premise that software has a better knowledge of how the memory is used, for example how it is grouped in different processes, and can therefore make smarter decisions in which, how and when memory should be moved around within the system. Such implementations would benefit from a hardware/software co-design where the hardware provides an interface so that the software can fully control or provide hints to the hardware coherence mechanisms. A similar method [77] has been proposed in the past for scalable multiprocessor systems.

2.3.4 A case for hardware managed concurrency

If we want to be able to exploit the amount of parallelism offered by many-core architectures, we need our applications to expose as much concurrency as they can. While it is not a problem for high performance computing (HPC) applications to scale up to hundreds of thousands of processors, these are very regular and embarrassingly (data) parallel computations. This is a difference with the general-purpose computing domain, where we have to cope with irregular concurrency patterns and where single applications suffer from the Amdahl bottleneck [8]. In order to be able to expose the maximum amount of concurrency there, creating and handling concurrency should incur the least possible overhead. The smaller we can make the granularity at which we can efficiently deal with concurrency, the more concurrency will become economical to exploit. For example, if we can start concurrent contexts within a few processor cycles, it already becomes feasible to exploit concurrency at the level of 10s of instructions.

Recently, several approaches have been suggested [92, 108, 132] to add hardware support for task management. These provide hardware support for queuing and setting up fine grained tasks, which are fragments of code that can execute within a thread of execution. Software can then efficiently fetch a new task into a thread as soon as a task completes using these hardware mechanisms. These mechanisms complement the coarser level of concurrency management done by the operating system which manages threads and processes which contain these small tasks. While this already provides us with hardware assisted concurrency management that is efficient down to the granularity of one thousand instructions, we have two reasons to take it even a step further.

First of all, having concurrency management fully in hardware and locally on a core has several advantages. It allows for very low latency creation of local concurrent contexts at the granularity of instructions, such as in the Denelcor HEP [137], and can be done on other cores efficiently with messages over on chip networks. Having multiple local concurrent contexts available for scheduling allows for latency hiding between them on a core [157], as has also been demonstrated in recent commercial processors [136] to improve throughput [MWvT9]. This latency tolerance is required for future many-core architectures, even when 3D stacking techniques are used for memory. This technology will provide a major improvement in memory bandwidth [105], however, accessing memory will still have a considerable latency assuming we use SDRAM technology. Another source of latency are on-chip communications which increase with the distance they have to travel, which increases with the number of components on the chip.
Our second reason is that we are interested in investigating a holistic approach where we have a composable and uniform way of managing concurrency at all different levels of granularity, from operating system processes and threads down to fine grained concurrent executions at the level of tens of instructions.

Besides having hardware support for starting, terminating and scheduling of concurrent contexts, it is also desirable to have support for synchronizations and mutual exclusion in combination with the scheduling mechanism. This way, the hardware scheduler can suspend a context at a low cost when it waits for a synchronization or on entering a critical section, instead of having the context in a busy-waiting state unnecessarily consuming processing resources and power.

2.4 Operating systems: State of the art

After the discussion in the previous section, we now have a vision of what our future many-core computing platforms might look like. Likely, they will share a lot of similarities with heterogeneous distributed systems, and therefore we will now investigate and discuss what kind of operating systems have been developed to support such hardware designs. Starting with several multi-processor and distributed operating system projects from the past, and moving on to more recent projects of the last decade that are more specifically targeted to support future generation many-cores. We present a short overview of each system, discussing their design principle, structure, and mechanisms.

After presenting this overview of parallel and distributed systems, we look at contemporary operating systems that are popular today. We discuss what the problems are that they have been facing to efficiently support multi-core architectures in a single system view. As these operating systems are also used on clusters, we end this section with looking at cluster organizations and at approaches to program these with distributed parallel languages and their run-times.

2.4.1 Early multiprocessor systems

Hydra

Hydra [165] was one of the first operating systems that was designed to run on a larger number of processors. Earlier systems such as Multics [41] and IBM STRETCH [38] were geared towards supporting one or two CPU configurations, while Hydra was designed to run on the Carnegie-Mellon Multi-Mini Processor (C.mmp) which would consist of up to 16 processors from PDP-11 minicomputers. The design was heavily influenced by the design proposed by Hansen [71], and was perhaps even the first real implementation of a microkernel. It provided support for a capability based [49] security and protection mechanism based on objects and procedures. The design had a clear separation between mechanisms and policies, where the kernel only takes care of the mechanisms. A good example of this is the flexible capability system which the kernel implements; it does not have predefined operations such as read and write, but these policies are determined by higher level system software. To guard the
state of objects and for synchronizations, the system used semaphores as proposed by Dijkstra [51].

DEC Firefly

Firefly was an early multiprocessor workstation by DEC [144] with one to nine MicroVAX CPUs. Main memory was shared between all processors, where each featured a local cache that was kept consistent with a cache coherency protocol. The system was asymmetric in hardware because only CPU 0 was connected to the I/O devices. However, as most devices would use DMA to access the main memory they would require little interaction with CPU 0 and could therefore be accessed symmetrically in software. Only an inter processor interrupt needed to be sent to CPU 0 to invoke the appropriate driver which would set up the device for the DMA transfer.

The Taos operating system was developed for the Firefly, based on a kernel called the Nub. This kernel provided scheduling, protection, simple device drivers and RPC support [134] over all processors. All other operating system facilities, for example file systems and file based I/O, were provided by Taos which ran as multiple threads in a virtual address space on top of the Nub. Taos could support both emulation of the Ultrix (DEC’s Unix variant of that time) system call interface, or the highly multi threaded Topaz interface for programs [107]. Topaz could be programmed with (and itself was written in) the Modula-2+ language [129], an extension of Modula-2 with added support for multiprocessor concurrency, exception handling and garbage collection.

The system made the creation of threads of execution orthogonal to the creation of virtual address spaces and the creation of operating system contexts. This way it could support efficient multi-threading where multiple threads of a single program could be scheduled simultaneously across multiple processors. The threads supported mutexes, conditionals and semaphores as synchronization primitives [16]. RPC was used to communicate between two address spaces, whether this was local to another process on the same machine or transparently across the network [134].

2.4.2 Loosely coupled multiprocessors

Transputer

The transputer was a processor developed by INMOS [161, 80] which was designed as a building block that would allow scaling for large parallel systems. The name transputer, a combination of transistor and computer, reflected the vision that such processors would form the basic building block for larger systems, similar to what transistors are for processors.

The transputer had a local memory and four bi-directional communication links to talk to neighboring transputers, forming a decentralized network. Using these links an unlimited number of transputers could be connected together, for example in a mesh topology, but with large numbers system performance would suffer for non local communications as multiple hops would have to be taken impacting every transputer on the way. It had built-in support for the creation and management of concurrent processes, which could have a high or low priority. The hardware scheduler would run
a high priority process until it blocked, for example to wait on communication, and
schedule low priority processes with time slicing if there were no other high priority
processes available. The designs did not feature a Memory Management Unit (MMU),
and therefore it had no virtual address space and memory protection support, nor did
the transputer support different privilege levels. One of the reasons behind this was
that running applications on different transputers in the same array already provided
sufficient isolation; They could not access or address each others memory and could
only communicate through messages on the network interfaces.

The hardware design reflected its support for the Occam language [106], a con-
current programming language based on the Communicating Sequential Processes
(CSP) [79] model. Programs are written as collections of concurrent processes that
communicate with each other through one way synchronous communication channels.
An integrated development system, TDS, was provided to write, debug and run Oc-
cam programs on a transputer machine. As the hardware already supported multiple
concurrent processes and hardware channels, no further operating system support was
required to run a single Occam program. However, in order to run multiple Occam
programs at the same time, transputers had to be allocated and the networks had to
be configured [CSA24].

Special transputers with I/O interfaces would be present in the system which
would be able to run the same Occam code, but could also take care of commu-
nication with attached devices and the outside world. In many configurations, a Unix
workstation would act as a head node to launch programs onto a transputer array, or
the transputers would be located on a plug-in board inside a personal workstation or
PC.

As Occam did not gain sufficient momentum in the parallel computing domain,
many parallel operating systems were developed to support C and Fortran style pro-
grams on transputers, often with a Unix-like interface. Some examples were He-
liOS [119], Trollius [27] (formerly named Trillium), TOPS [60] and Chorus/MiX [5].
All of these systems provided a uniform messaging layer to implement IPC between
arbitrary processes on arbitrary transputers, routing them over multiple links. HeliOS
was a rather complete distributed system similar to V and Amoeba (both discussed
in the next subsection on networks of workstations), which would partition the sys-
tem into several per-user allocated domains. Chorus/MiX was designed to be a more
resilient Unix-like distributed system that would provide a single system image. It
targeted the T9000 transputer, which would have offered privilege levels and memory
protection, but this processor was canceled at a later stage.

Thinking Machines CM-5

The Connection Machine 5, or CM-5, was one of the massively parallel machine
designs by Thinking Machines [78]. It consisted of configurations from 32 up to
a thousands of processing nodes which each contained a SPARC processor and an
optional vector processor, some local memory and a message based network interface.
The CM-5 contained three physically separate packet switched networks for data,
control and diagnostics [97].
The first network was for data transmission which would scale up in bandwidth with the number of nodes due to its organization as a 4-way fat tree. The second and third networks were organized as binary trees, which could be split into subtrees to partition the system. These partitions were configurable by software but enforced by hardware, there was a strong isolation and no bandwidth sharing between partitions. The data network would be used by applications for send/receive style message passing communication, and the control network for synchronizations or collective operations such as reductions. Special functionality on the control network and the whole diagnostics network, the latter for example provided JTAG access to most components in the machine, could be only accessed by the operating system.

The operating system for the CM-5 was a Unix system named CMost which was based on SunOS 4.1 [145]. Every partition in the system had one control processor associated with it which would run the full Unix kernel and services to act as a partition manager, and all other processing nodes only ran a simple microkernel. The functions of the data and control network interfaces could be accessed directly by the applications through memory mapped control registers, which resulted in a low overhead as no system calls were required. As each node had full virtual memory support through an MMU, the access to the network interfaces could be restricted by memory protection. Using this network interface, an application can explicitly send/receive messages between its processes on different nodes, or use one of the higher level parallel programming language constructs.

Applications are gang scheduled by the partition manager onto the whole partition, which means that each node in the partition runs the same application when it is scheduled, but the partition can be time sliced to share it with other applications. In order to do this, CMost broadcasts an interrupt using the control network so that all processing nodes will interrupt the current task and switch execution. Using a special mechanism all packets in the network of that partition can be pushed out and saved so that they can be restored as soon as the application is rescheduled again, making sure no communications are lost on a context switch.

Special I/O processors would be connected to external devices but also be part of the three networks in the machine. They ran special services to provide for example Unix style file storage to all partitions in the machine, which could be accessed over the network similar to NFS. Other Unix style communication such as pipes and sockets was supported for communication between applications.

2.4.3 Networks of workstations

In the 1980s, personal workstations became affordable and increasingly powerful. With the advent of Ethernet [109], it was possible to create large networks of these relatively cheap and powerful machines, which sparked the research in distributed (operating) systems. Their common goal was to harvest and combine the computing power of these machines, and present the user with a single large system image.
V operating system

The V operating system is a message based distributed system developed at Stanford [37, 36], with the goal to combine the computing power of a cluster of workstations that is interconnected with Ethernet [109]. Every node runs the V kernel which contains several services for process, memory, device and time management. The kernel supports virtual memory for protection and multiple threads per address space (which they refer to as process groups). Memory is file-mapped from a backing store and a local page cache is maintained by the kernel. The pages in the page cache are kept consistent with an ownership protocol as pages could be present in multiple page caches on different nodes. Using this mechanism, the threads of a process can be scattered across multiple nodes, using a distributed shared memory similar to that of Ivy [99]. As memory pages can easily be attached on another node this way, it also allows for thread migration and load balancing.

All communication is based on synchronous IPC, which is blocking and can either be used in a send/receive message passing form or as a remote procedure invocation. The network transport supported multi-cast for IPC messages which was used in services to efficiently implement time synchronization, the naming protocol, requests for and distribution of load information and replication. Most services are implemented in user-space, such as a file server or debugger.

Amoeba

Amoeba [152, 113] was a distributed operating system developed at the Vrije Universiteit in Amsterdam. As hardware was getting cheaper the Amoeba project set out to develop a system where each user would be able to use 10s to 100s of processors transparently through a single system image. This was done on a mixture of hardware, multi-computer boards to form a processor pool, a network of diskless workstations for user interaction, and specialized servers for databases and file storage, all connected by Ethernet.

Each Amoeba node runs a microkernel that provides scheduling, some low level memory management and three primitives to send, listen for, and respond to messages that are implemented in a special lightweight protocol. Device drivers reside as separate threads within the kernel, probably because the kernel needs to be able to access the network, so it is not a full microkernel approach. The kernel supports a discovery protocol to locate services in the network, and caches this information locally. Higher level functionality such as directories and storage for file systems or terminal services are implemented as servers that publish themselves on the network.

The nodes in the processor pool are exposed by a processing server. These nodes are capable of running tasks with multiple threads that are scheduled non-preemptively, and that can not span across multiple nodes. Amoeba does not support virtual memory, though a segment based protection scheme is used, as it is envisioned that processing resources are abundant, compared to the number of users, and tasks would not have to be multiplexed onto a single processing node. Segments can be detached from a process and then migrated to another process on another node, this can be used to share data between two processes but is also used to start a program on a processing node. Heterogeneity is supported between nodes as they all use the
same RPC interface, and the process description contains architecture and feature requirements so that a program is assigned to a compatible processing node.

The system is based on objects and capabilities that contain an identifier of a server port, object identifier, access rights and a cryptographic protection. By using the latter, the capabilities can be handled completely in user-space and do not require kernel support. The Orca language was developed on top of Amoeba [142]. It added the support for running larger parallel applications across multiple nodes in the processor pool by implementing a distributed shared object model.

**Plan 9**

Plan 9 is a distributed operating system that was developed at Bell Labs [122], which, similar to other distributed operating system projects of its time, had the goal to form a seamless time-sharing system out of a group of networked workstations, file servers and compute servers. It was largely designed by the same group of people that had been working on the development of the Unix [124] system, and was meant to become its successor. Plan 9 would deal with the non uniform extensions that were added after the initial design of Unix as new technologies emerged, such as networking and graphical user interfaces.

The main design decision in Plan 9 was to expose everything as a hierarchical file system, and that this was the interface to interact with resources, such as services provided by the operating system. Unix originally already had this abstraction with its block and character devices, but many more interfaces to the kernel were added later on, such as for example sockets and \texttt{ioctl} calls to drivers. Plan 9 defined a uniform protocol, named 9P, to access all resources, in essence 9P is exposed as a file server for the file system representation of resources, that can be transparently accessed locally or remotely. Each application constructs their own view on the file system with their own private name-space, which can contain both local and remote resources.

Similar to V and Amoeba, the Plan 9 kernel provides a minimal set of functionality to manage memory and implement the 9P protocol on the hardware. However, strictly speaking, it should not be regarded a microkernel. As in V, services can reside in the kernel or outside, as long as they provide a 9P interface to expose their functions. Due to this organization and uniform interface, Plan 9 does not have a large number of system calls, and most of them deal with file handling. As all resources are represented as files, this is also the basis for the security model. This is done both at the file level with access rights similar to those in Unix, and at authentication when \textit{mounting} (attaching) a file system to a private name-space.

Plan 9 does not make a distinction between threads and processes, instead, it can be specified at process creation which resources the new process shares with its parent, which are copied and which are created. Processes can synchronize through the \textit{rendezvous} mechanism; the first process entering a rendezvous registers an identifier, a piece of data, and then blocks its execution. When the second process arrives at the rendezvous the identifier is matched, the piece of data is exchanged and the first process is woken up again. This is the only synchronization primitive supported but can be used as a basis to implement more complex synchronizations. Alternatively,
there is also an interface for atomic operations supported by the hardware, such as *compare and swap*, which can be used to implement *spinlocks*.

### 2.4.4 Multi and many-core operating systems

**Helios**

Helios [114], not to be confused with the operating system for the Transputer bearing the same name that was discussed earlier on, is an experimental operating system by Microsoft Research that targets heterogeneous multi-core architectures. It assumes an organization with a CPU and multiple heterogeneous peripherals that contain cores that support general purpose execution, without requiring a shared memory between them. The CPU runs the coordinator kernel and uses small satellite kernels to execute on the peripheral cores to provide uniform OS abstractions in the system. The system is organized in the style of a microkernel, where operating system services can live anywhere in the system and are transparently accessed through messages. To support the heterogeneity of the target architectures, it uses just in time compilation from architecture independent .NET byte code. To optimally place application components and services in such a system, a method of describing the relation and affinity between components is offered. Using this affinity, services that use certain I/O devices can be placed close to the producer or consumer of their data, making the system more efficient than when this were to be run on the main CPU. For example offloading the network stack service to an embedded XScale core in the network card.

**Fos**

The Factored Operating System (*fos*) is an operating system developed at MIT [159, 160] that targets many-core architectures, clusters and clouds able to scale up to thousands of cores. It is built as a paravirtualized microkernel on top of the Xen [12] hypervisor combined with servers in user-space. The microkernel takes care of protection mechanisms, but not policies, similarly to Hydra [165] that was discussed before. Fos takes an approach of space sharing where operating system services reside on dedicated cores spread around the system, and other cores are fully dedicated to applications. As common in a microkernel approach, even low level system services such as memory management and scheduling are run as such servers, that have a similar design as distributed Internet servers that collaborate and exchange information. The microkernel provides an asynchronous messaging implementation for IPC, and a naming service similar to DNS [110] runs in the system to locate system services. As communication is message based, it does not require the underlying architecture to have a (coherent) shared memory.

**Barrelfish**

Barrelfish is a research operating system developed at ETH Zürich in cooperation with Microsoft Research [135, 13]. It is targeted to support heterogeneous many-core systems that do not require a coherent shared memory, and for example has been ported to the 48-core Intel SCC platform [121]. One of the design principles of the
The operating system is that state should be replicated instead of shared, and the system should have a fully distributed systems approach based on message passing that is hardware neutral. BarreliFish is a multi-kernel based system which has a CPU driver that runs on each core within the system that provides scheduling, communication support and low level resource management. Every core is handled in isolation, which separates it from classic SMP-based microkernel approaches that share state in memory. A monitor runs in user-space that takes care of inter core RPC and long term scheduling policies. Though the operating system itself is fully distributed, it does support shared memory multi-threading for applications. Each application has a dispatcher, on every core it is mapped to, that schedules the individual threads in software. The CPU drivers use gang scheduling [120] to co-schedule the dispatchers of an application.

BarreliFish uses a System Knowledge Base to provide all kinds of information to the system services and applications running on the system. This for example includes system topology and current system load. Constraint logic programming can be applied to the SKB to auto tune mappings of applications, either explicitly by the application itself or by services as part of the system.

Tessellation

Tessellation is a many-core operating system project at Berkeley [104, 39], that focuses on strong functional and performance isolation. It takes a space-time partitioning approach where resources are split into partitions, cells, that are assigned to individual applications or application components. These cells are then, depending on the application requirements, time multiplexed with other cell instances. Non computational resources such as network bandwidth, caches and memory bandwidth are allocated with quality of service (QoS) guarantees. Cells have secure message based communication channels to other cells to provide strong isolation. All hardware concurrency contexts are gang scheduled with a cell, so that they are all available to the application at the same time when a cell is scheduled. The application takes care of the use of resources within the cell, such as thread scheduling, giving a two layer scheduling organization.

The Tessellation kernel provides a minimal set of support for applications running inside a cell, consisting of access to (hardware) execution contexts, virtual address translation and support for the secure communication channels (with QoS) between cells. This minimal support resembles the Exokernel approach [54]; the application uses a library OS for OS functionality, such as scheduling, within its own cell. Operating system services that are available to the whole system run in their own cell.

One important system service in Tessellation is the Policy service which handles requests from applications for allocating or extending cells. It maintains an internal space-time resource graph in which the requested cells and their relation are represented. This graph is passed to the kernel which has a mapping and scheduling component that validates the request and executes the scheduling of cells on resources. The Policy service also gathers performance feedback from applications and hardware metrics to adapt the resource allocation. As application programmers are not expected to know their exact resource requirements beforehand, this feedback can be used to auto tune the application and resources. Resources that are idle but
are reserved for one application can be opportunistically shared with another on a best-effort basis, until they are revoked by the owning application for its own use.

### 2.4.5 Commodity Operating Systems

In this subsection we discuss common modern commodity operating systems, and if and how they can be applied to future many-core architectures, and what some of their problems are. We focus on Linux, as this is a rapidly developing open source operating system with the most publicly available information on its internal mechanisms. However, similar issues apply to its closed source competitors such as Windows, Mac OS X and Solaris to name a few.

Linux, as well as most of the other commodity operating systems such as the Windows NT family, has a layered monolithic design with many subsystems and supports at run time loading and unloading of modules to extend functionality. Unlike a microkernel design, everything, including these modules, runs unprotected in kernel space. Linux has been embraced by the scientific community as well as computer science research as much as Unix in its early days; as an illustration, 456 out of 500 of the top 500 supercomputers\(^4\) at the time of this writing run Linux.

#### Single System Scalability

Support for multiple processors was added in the early 2.0 and 2.2 kernels of Linux, but as with most systems of that time, this was by simply adding a lock around the whole kernel so that its data structures could only be accessed by a single processor at the time. This created the **Big Kernel Lock** (BKL), which later turned into one of the greatest and most complex scalability issues the Linux kernel developers had to deal with.

The situation was improved in the 2.4 kernel \([26]\), where the use of the BKL was made more fine grained and was partially replaced by separate locks. However, the scheduler still had a central run queue and associated locks, which was replaced with a new \(O(1)\) scheduler with per-core queues and locks in 2.6 \([1]\). The virtual memory management was redesigned not to depend on a single process and a single lock and, as well as the scheduler, was made NUMA aware \([19]\), which allowed for a considerable performance improvement in multi-processor systems \([73]\). Another important improvement was the **read-copy-update** mechanism (RCU) \([67]\) that allows concurrent updates and reads from data-structures instead of using a read/write lock, where readers do not need to block on a lock while the writer adds an update.

It is difficult to judge the scalability of Linux for future architectures, but this has been one of the key points of attention in its development over the last few years. A study from 2010 \([43]\) compared the multi-core scalability using OS micro benchmarks such as process forking, opening and closing sockets and memory mapping a file on Linux 2.6, OpenSolaris and FreeBSD. This study showed that none of them scaled perfectly, as all three had increasing overheads when the number of cores was increased from one to 32 on one or more of the benchmarks. However, it also showed

---

3. The Linux kernel map illustrates this: [http://www.makelinux.net/kernel_map/](http://www.makelinux.net/kernel_map/)

4. Top 500 Supercomputers June 2011: [http://top500.org/list/2011/06/100](http://top500.org/list/2011/06/100)
that for most of the benchmarks at least one of the three had a scalable solution. Linux for example scaled well on the process forking while OpenSolaris did not, but it was exactly the other way around in a benchmark of System V semaphores. Another study by MIT [25] explored the scalability of Linux up to 48 cores by running several application based benchmarks. They identified the observed performance bottlenecks induced by the kernel, and developed patches which improved the scalability considerably.

One of the biggest problems is from an engineering perspective, as Linux was not originally designed from the ground up to support many parallel hardware contexts. A lot of effort has to be put into introducing fine grained locking and RCU techniques, often requiring rethinking and redesigning of a subsystem before they can be applied. Moving to finer grained locks also has its downside, as it becomes increasingly difficult to extend and maintain the code while avoiding all possible deadlocks. Furthermore, locking too often will incur more overhead, so finding the right granularity for locks is important, but difficult.

A more general issue with locking is that when multiple clients compete for the same critical section, performance becomes a problem. Besides hitting scalability limitations due to Amdahl’s law, most of the cores potentially waste a lot of time (and energy) on a spinlock not doing any useful work. The problem even gets progressively worse: while multiple cores are spinning on a lock, cache coherence traffic will keep pushing the cache-line containing the lock around. This is not much of a problem with only one or two cores, but when the number of actual clients running in parallel in hardware increases, this contention becomes a major performance bottleneck. There are some solutions to alleviate this problem, such as using an Ethernet style back-off algorithm or lock probing, but these of course add an associated overhead.

**Cluster Scalability**

As mentioned in Section 2.4.5, most of the Top 500 supercomputers run Linux, and most of them are organized as clusters. These large distributed systems scale to hundreds of thousands of cores, and as future many core architectures more and more look like distributed systems, we will now discuss the possibilities of such a cluster approach. In fact, the Intel SCC [81] initially took this approach, as Intel provided a Linux distribution that would run individually on each of the 48 cores, providing an MPI-like message passing implementation [151] directly using the on-chip network.

These large computing clusters are usually operating with batch processing systems where execution times are measured in days rather than in milliseconds. Their typical computation bound workloads are quite different from general purpose computing, they contain coarse grained parallelism, are embarrassingly data parallel and the exposed concurrency is often static. This allows such systems, both hardware and software, to be designed and tuned for the specific type of workloads that will run on them.

The same holds for *Grid Computing* where the resources of clusters are bundled using frameworks such as Globus [59, 6], which can be used by many different applications. While this seems to produce more heterogeneous types of workloads, they will still be large long running applications with coarse grained parallelism with embar-
rassingly data parallel computation – they have to be, or they would not be able to benefit from being run on the grid. In such settings, a node is generally only allocated to a single application, and any operating system interference is rather considered a nuisance that induces additional latencies.

The third type of approach with computing clusters that has been becoming increasingly popular over the last few years is Cloud Computing. Popularized by Amazon with their Elastic Compute Cloud (EC2) service\(^5\), which started out when Amazon decided to rent out their spare server capacity to third parties. Instead of allocating nodes with all their specific properties, (operating system, architecture, software versions), it uses virtual machines so that the user can fully control what to run on a node and is presented a uniform machine interface even though the hardware underneath might not always be the same and physical machines can be safely shared. It is mainly used for services that can be easily replicated, such as web servers, and instances can be flexibly be added and removed from the pool.

The problem with all these systems are the non uniform programming models and interfaces, and that responsibility is scattered between one or more run-time systems and the operating system who share almost no information on their activities. For example, starting a new process locally or on another node remotely is done through a different interface. These are the same kind of problems that Plan 9 \cite{plan9} attempted to tackle, which we discussed earlier. Another more recent project, XtreemOS \cite{xtreemos}, is attempting to close this gap between Grid computing and other interactions with the OS by providing POSIX style interfaces for Grid programs by integrating Grid support into the Linux kernel. However, there are also issues with such transparent integrations of local and distributed computing as was identified by Waldo et. al. \cite{waldo} who argued that parallel and distributed computing should not be treated transparently. The two major issues are differences in expected performance or latency, i.e. orders of magnitude difference between local and remote communications, and differences in failure patterns. While this is clearly true for distributed systems, this performance gap closes more and more when we move to many-core architectures with networks on chip that provide a high bandwidth and low latency.

**Distributed run-times**

To complete this section, we will discuss several approaches for programming distributed systems, as over the years, many ways of programming such environments have been developed. Many of the discussed approaches rely on new languages or language features, while others will work as pure library implementations. Even though MPI is very dominant in the large scale cluster community, we present these as an illustration of the uniformity of programming models, and as a general background on programming distributed systems.

One approach are distributed shared memory (DSM) implementations, where all nodes have access to a shared state. For example the implicit or explicit sharing of objects \cite{implicit, explicit, regions, shared_address_space}, regions \cite{regions, regions, regions, regions}, or an entire shared address space \cite{shared_address_space}. The other end of the spectrum has been dominated by explicit message passing techniques \cite{message_passing, message_passing}, and in between we have remote calls (possibly to remote objects)

\(^5\)Amazon EC2: \url{http://aws.amazon.com/ec2/}
Ivy [99] was one of the first DSM systems that attempted to act as a transparent single address space shared memory system by sharing memory on the page level and using handlers on page misses to transfer data. However, this did not turn out to work efficiently enough, false sharing being one of the issues, and many later DSM implementations are based on explicitly acquiring, reading or modifying and releasing state. CRL [86] for example uses a region based approach where special global pointers are used to map and unmap shared regions of arbitrary size to code running on a node. After a region is mapped, the code can enter either a reading or writing section, where writing sections guarantee exclusive access. Munin [30] also uses the acquire/release principle, but allows the consistency protocol, which is based on release consistency [65], to be configured for individual objects; i.e. invalidate or update copies on write, enabling replication and fixed home locations. Cid [115] also implements acquire/release with single writer multiple readers, but also exposes the location of objects with the ability to start a computation on an object on the node where it is located, providing the flexibility of moving either the computation or the data.

In Orca [142] the acquire/release happens transparently on shared objects that get replicated. The objects are not globally visible but are passed by reference between (concurrent) invocation of functions, limiting their visibility to a relatively local scope. However, when multiple functions operate on the same object it is kept coherent by updating or invalidating copies on write. Emerald [87] provided similar mechanisms, however it did not support replication and therefore did not have to deal with coherency.

CICO [77] is a cooperative model in which memory regions in a shared address space can be checked in, out, and prefetched, which provides a hinting mechanism for a hardware based coherency implementation. Cilk [21] has a restricted way in which data is moved following the DAG-consistency [20] model; there are well defined points when data needs to be communicated, as there is no strict coherency which requires propagation of updates as soon as data is modified. Another approach that identifies inputs and outputs using compiler pragmas to annotate functions CellSc [14], which is used to efficiently write programs for the distributed memory in the Cell [66] architecture. Sequoia [57] is a programming model in which a (distributed) system is viewed as a hierarchy of memories, and programs in Sequoia can be automatically adopted to the granularity of the target system. Sequoia uses call-by-value-result semantics, where for each function argument is specified if it describes an input, output or both. GMAC [64] is an implementation of an asynchronous distributed shared memory which attempts to unify the programmability of CPU and GPU memories. The Batch-update mode of GMAC matches closely with the approaches mentioned above, but it also supports more elaborate coherency protocols where the GPU can receive updated data from the CPU asynchronously.

Another aspect of distributed systems is the ability to start remote computations, and waiting for them to complete. As mentioned earlier, one of the issues discussed in [156] was the difference in failure patterns between starting local or remote computations. Checking for a failure on any concurrent invocation would still be expensive
in a many-core system with fine-grained concurrency, but can be done at a software component level. R-OSGi [123] is a system that takes this into account, it distributes transparently at the software module level, and does not introduce any new failure patterns. Furthermore, it does not impose any role assignments i.e. whether a node acts as a client or server; the relation between modules is symmetric.

Chapel [32] is a new programming language aimed to bridge the gap between parallel and sequential programming. It hierarchically expresses both task and data level concurrency, which transparently can be executed locally or remotely in parallel, or sequential, but it currently does not deal with (partial) failures in remote computations. X10 [34] is similar in that respect and is developed with the same goal as Chapel. It uses futures and final variables for communications and synchronizations, and it uses places to express locations that have sequential consistency, which provides a handle for expressing locality. Cid [115] has this feature as well, as the home node of a piece of data can be extracted. This can then be used with its fork if remote construct, executing sequentially if the referenced object is local, or otherwise remotely in parallel.

Other approaches such as Active Messages [155], CORBA [116], Legion [98], RPC [17], Java RMI [164] and SOAP [24] but also message passing approaches such as MPI-2 [63] and PVM [62] are based on coarse grained parallelism where finer grained local parallelism must be expressed using a separate paradigm, for example using a shared memory multi-threading implementation. MPI-2 and PVM support the dynamic creation of tasks, but again, only at task level parallelism. Most of these approaches support partial failure, but at the cost of not making remote communication transparent. None of them provide a distributed memory abstraction, though CORBA, Java RMI and Legion do this in a way by transparently accessing remote objects, which in turn requires a look-up service to be able to locate these objects.

2.5 Conclusion

In this chapter we have discussed the basic concepts of an operating system, and what the influential classic design structures look like. In this process we identified the basic abstractions that are required in order to implement a general purpose operating system. Following this, we reviewed the state of the art of computer architectures, and discussed the likely directions of future developments; a move towards heterogeneous, on-chip distributed many-core architectures is anticipated. From this future perspective we studied and discussed the design of both classic distributed systems as well as current state of the art research projects that attempt to tackle the challenges of efficiently supporting such architectures.

While distributed general purpose operating systems seem to have gone out of fashion almost twenty years ago, we see a lot of their ideas re-emerge in current many-core operating system projects. It is clear that central structures have inherent scalability problems, and that operating systems are required to adopt a distributed design. This quickly leads to an analogy with microkernel designs, where most operating system tasks are made available by distributed services.
2.5. CONCLUSION

As we learned in our distributed run-times section there are many ways to implement distributed services. For example, we can send messages, use a remote procedure call, transfer or copy and update the state locally, or use methods on objects that do all of this transparently. However, we see in the operating systems that we investigated that they are mostly IPC/RPC based where services reside on specific nodes in the system. The reason for this is probably that the updates sent to services involves less communication than the data required for a service to act upon, therefore it is cheaper to contact a service remotely than to migrate its state and update it locally. Another interesting aspect of dedicated service nodes is their placement and locality effects, for example as we saw in Helios, offloading certain I/O processing services close to the I/O device can provide good benefits.

Older distributed systems such as Amoeba and the CM-5 had a separation between nodes that ran services and user applications, and we see this again in recent systems such as fos. In Amoeba specifically, computing nodes were uniquely allocated and not shared between applications, while the CM-5, Tessellation and Barrelish would multiplex multiple applications making sure each part of the application gets gang scheduled at the same time, and Plan 9 uses Unix style time-sharing. Avoiding time-sharing between applications and operating system services avoids switching between different working sets, and often privilege levels, that translates into cache and TLB misses. Similarly, if we assume to have a sufficient amount of cores, we can go a step further and eliminate time-sharing between applications, as assumed by Amoeba and to a lesser extent by fos.

We summarize the common design choices and directions of the systems we have discussed as follows, which we can use as a guideline in this thesis:

- Message based communication
- OS split in minimal support on each core and dedicated cores with services
- Space-sharing instead of time-sharing
- OS does not require (coherent) shared memory

This chapter would not be complete without discussing commodity operating systems used today and their ability to scale to future architectures. We specifically focused on Linux, exploring the scalability issues and challenges that it encountered in the last few years. Deciding if such systems will scale for the future many-core architectures remains a question unanswered, but it will likely become increasingly difficult. Another option would be to use a cluster based approach where the hardware is partitioned between multiple operating system instances, but this is difficult to manage as it presents many non uniform abstractions. We also gave an overview of many programming approaches and run-times targeted to distributed, as these are closely related to distributed operating systems; they provide an abstraction on top of a concurrent distributed machine.
CHAPTER 3

SVP

3.1 Introduction

In Chapter 2 we have made an argument for hardware support for concurrency management in future many-core architectures, and in Chapter 1 we have introduced the SVP model and briefly commented on its history and context. In this chapter we describe the SVP model of concurrency, and discuss several implementations that are based on the concepts from this model, such as the Microgrid many-core architecture. We will use SVP as an abstract model for architectures that implement hardware concurrency management, though we would like to stress that we do not argue it to be the best possible concurrency interface, but merely a model under investigation that we find interesting, and it was the context in which the research for this thesis was conducted.

SVP was proposed as a generic concurrent programming and machine model [CSA20], having evolved as a generalization of the Microthreading [CSA18, CSA1] model. The main purpose of SVP is to describe concurrent activities and their relations, instead of explicitly mapping and scheduling them, covering all levels of granularity; from task level parallelism to instruction level parallelism. Mapping and scheduling is done dynamically by an SVP implementation. Due to the structure of the model, an ordering can be established over all concurrent activities in an SVP program, which guarantees there is always a well defined sequential scheduling. This allows for transformations, that can be applied at compile or at run time, that adapt the granularity of the exposed concurrency by folding the concurrency at the lowest level.

In order to be able to program SVP based systems, the $\mu$TC language [CSA19] was defined in early work to capture the semantics of SVP. It is based on the traditional C syntax, extending it with syntax to support all SVP actions. This was superseded at a later stage by the SL language [CSA37], which has a more explicit syntax that is better suited for code analysis and transformations, and overcomes many of the compilation issues [CSA2] imposed by $\mu$TC. Both languages are intended as system level languages, rather than concurrent languages aimed at programmer productivity such as X10 [34] or Chapel [32]. They expose a low level interface to SVP and are
intended for system level programming and as compilation targets for higher level languages such as SaC [CSA13, CSA10] and S-Net [CSA14].

Throughout this thesis we will use a simplified version of µTC when discussing example code. We refer to it as pseudo-µTC, to identify that it is µTC based pseudo code. For example we omit family and place identifier declarations where trivial, and omit braces, using indenting instead to group code, for brevity. We feel more comfortable using µTC over SL as a basis for our examples: we think it has a more simple human readable syntax and it was used in most of the work covered by this thesis before the newer SL language and toolchain emerged.

One of the implementations that follows the SVP model is the Microgrid [CSA22] many-core architecture. The Microgrid evolved from the Microthreading work and implements efficient fine grained hardware threads at the instruction level, supporting hundreds of threads per core. We will discuss the Microgrid architecture in more detail in Section 3.3. Another implementation based on SVP is a more coarse grained software approach that uses POSIX threads on commodity multi-cores [MWvT12], which was later extended with support for distributed systems. This implementation and the extensions required to SVP in order to support distributed memory are discussed in this Chapter in Section 3.4 and Section 3.5. As the SVP model has evolved over the years, we will define the model that we assume in this thesis in the next section.

### 3.2 Definition of SVP

The SVP model defines a set of actions to express concurrency on groups (families) of indexed identical threads. Any thread can execute a `create` action, which is a split-phase operation to start a new concurrent child family of threads. Then, it can use the `sync` action to wait for its termination, implementing a fork-join style of parallelism. The `create` action has a set of parameters to control the number and sequence of created threads, as well as a reference to the thread function that the threads will execute. This thread function can have arguments, defined by SVP’s communication channels explained later on. `create` sets an identifier for the created family, which is used for example on `sync` to match a specific family. The `create` and matching `sync` action need to be executed in the same context; the identifier for the created family can not be passed to another thread to be used there to `sync` on the created family.

When the iteration space of a family is large, not all threads may be able to run at the same time limited by the available amount of slots the SVP implementation has available at that moment. The number of concurrently scheduled threads from a single family can also be constrained with a blocking parameter specified at `create`. Initial threads of the sequence are started until the block is full, then further threads in the sequence are started only as earlier threads complete.

As any thread can create a new family, the concurrency in a program can unfold into many hierarchical levels, which we refer to as the concurrency tree of a program as shown with an example in Figure 3.1 that creates families of different sizes. This is a representation of the program at a given point in time, as families are created and complete over time, this tree evolves.
3.2. DEFINITION OF SVP

A special case is a detached create, where a context can use create to start a new family but does not have to execute the sync action. One of the uses for this is continuation style programming, such as starting a registered call-back function, where a next continuation is started and the original context can terminate without having to wait for it to complete. In the µTC language support for this was added with two flags at the create, one to define it as detached, and one to define a container family.

The issue with detached families is that we lose the handle on them after they are created and we can not use their identifier to sync on them. However, for example, we want to be able to make sure that all detached families have completed before releasing the resource they were running on. In order to facilitate this, we introduce the container family, which is a family which can only complete when all detached families that have been created below it in the concurrency tree have completed. A sync on the container family effectively becomes a bulk synchronization on all detached families in that sub-tree.

Besides the two basic actions of create and sync, there is the kill action to asynchronously terminate an execution, and break to abort the family of the thread that executes this action. A fifth action, squeeze was described in earlier literature [CSA20] to pre-empt the execution of a family of threads, but this has been abandoned. An analysis was made of the use cases and how it could be applied in programs, and we concluded that there were many issues that made its use very limited. In Chapter 4 we look at squeeze in more detail and analyze its problems, and we will discuss why it did not provide us with sufficient support for pre-emption which it was originally intended for.
3.2.1 Resources

The SVP model has an abstract notion of resources, and is resource and scheduling naive. Resources are represented by the concept of a place, which acts as an abstract resource identifier. On a create action a place can be specified where the new family should be created, which results in a delegation of the execution to that resource. What this place physically maps to, and how it schedules the work it gets assigned, is left up to the underlying implementation; for example, on a many-core architecture like the Microgrid, it would be a group of cores. On other implementations it could be for example a reserved piece of FPGA fabric, an ASIC, or some time-sliced execution slot on a multi-core system. The SVP model does not require a single place to map to a single physical resource, the implementation can decide to multiplex places.

One important property that a place can have is that it can be exclusive, which we will refer to as an exclusive place, which is used in the model to achieve critical sections with mutual exclusion. A create to an exclusive place is analogous to having a locked mutex until the created family terminates and the sync completes. When multiple families are delegated to an exclusive place, the access to the place will be sequentialized and only one family can be executing on the place at a time. The SVP model does not specify or restrict the ordering in which multiple outstanding families are executed on an exclusive place.

3.2.2 Communication and Synchronization

Synchronized communication is provided through a set of channels, which connect to threads within a family and to their parent thread. There are two types of uni-directional write-once channels of which multiple can be present: global and shared channels. Writes to these channels are non-blocking while reads block when no data has been written to the channel yet. A global channel allows vertical communication in the concurrency tree as a broadcast from the parent thread to all threads in the family. A shared channel allows horizontal communication, as it daisy-chains through the sequence of threads in the family, connecting the parent to the first thread and the last thread back to the parent. In $\mu$TC, these channels are specified by arguments to a thread function, similar to normal function arguments, and identify the data dependencies between the threads.

Due to this restricted definition of communication, and under restricted use of exclusive places, we can guarantee that the model is binary composable and free of communication deadlock [CSA44]. Furthermore, this implies that every family of threads has a very well defined sequential schedule. If concurrent execution is infeasible, it is guaranteed that a family can run to completion when all of its threads are executed in sequence as long as there is a single execution resource available. This property enables program transformations that sequentialize families into loops at the leaves of the concurrency tree, folding the concurrency which allows us to adapt the granularity and amount of exposed concurrency for a given platform.
3.2.3 Memory Model and Consistency

The definition of SVP assumes a global, single address space, shared memory. However, the memory is seen as asynchronous as it cannot be used for synchronizations, as no explicit memory barriers or atomic operations on memory are provided. It has a restricted consistency model where we only enforce consistency when concurrent activities interact; at the points of `create` and `sync`. There are several reasons for this approach to main memory. First of all, as we discussed in Chapter 2, it will become increasingly difficult to provide global sequential consistency [95] in future many-core architectures. Secondly, by disallowing synchronizations through memory, we only have forward dependencies between threads, expressed by `create`, `sync` and the global/shared communication channels, which guarantees the freedom of communication deadlock and the ability to trivially sequentialize execution.

We define the consistency model with the following three rules:

- Upon creation, a child family is guaranteed to see the same memory state as the parent thread saw at the point where it executed `create`.
- The parent thread is guaranteed to see the changes to memory by a child family only when `sync` on that family has completed.
- Subsequent families created on an exclusive place are guaranteed to see the changes to memory made earlier by other families on that place.

The memory consistency relationship between parent and child threads somewhat resembles the release consistency model [65]. In that respect, the point of `create` resembles an acquire, and the point of `sync` resembles the release. However, the difference is that nothing prevents the parent and the child from modifying the same memory at the same time, which leads to undefined behavior.

The third rule of the consistency model allows exclusive places to be used to update shared state. It can also be used to implement a service; state is resident at the exclusive place and instances of the functions implementing that service are created on the place by its clients. We will elaborate further on this mechanism in Chapter 4 when we are discussing system components.

Data passed through the global or shared channels is not affected by the consistency model, as it is either there or not and cannot be written multiple times. It is likely that in certain implementations, the Microgrid being one example, the contents of the channels are limited to scalar values. Therefore, when larger data is passed through the channels at the programming level, the implementation should pass a reference to a data-structure in memory through the channel instead of the data itself. It then has to guarantee that there is memory consistency for the referenced data when this is read from the channel.

3.2.4 Example

The basic concepts of SVP are illustrated in Figure 3.2 and Figure 3.3 using an example code that generates the Fibonacci sequence and stores it in an array. It
1 thread fibonacci(shared int p1, shared int p2, int* result)
2    index i;
3    result[i] = p1 + p2;
4    p2 = p1;
5    p1 = result[i];
6
7 thread main(void)
8    int result[N];
9    int a = result[1] = 1;
10   int b = result[0] = 0;
11   create(fid; pid; 2; N;;) fibonacci(a, b, result);
12   sync(fid);

Figure 3.2: Fibonacci example in pseudo-$\mu$TC code

must be noted that this example yields little exploitable concurrency, but is easy to understand and illustrates all the concepts.

In Figure 3.2 we show the pseudo-$\mu$TC code that implements Fibonacci, with the iterations of the algorithm defined as a thread function in lines 1-5. The definition on line 1 identifies the two shared channels for the two dependencies in Fibonacci, as well as a global that will contain the pointer for the result array. The shared channels are read implicitly on line 3, and written on lines 4 and 5. Line 7 to 12 show the main function of the program that will start the concurrent Fibonacci iterations. \texttt{fid} is a variable that can hold a family identifier which is set by the \texttt{create} on line 11. Similarly, \texttt{pid} is a place identifier which controls where the fibonacci threads are delegated to. The initial values for the algorithm are set in lines 9 and 10, and the spawn of concurrent iterations is done with the \texttt{create} statement on line 12 creating a family of indexed threads from 2 to \texttt{N} on the place identified by \texttt{pid}. The two omitted parameters can be used to further control the creation and indexing of threads by \texttt{step} and \texttt{block size} which control the iteration space. Information to identify the created family is stored in \texttt{fid}, and the \texttt{sync} statement on line 12 uses this to have the main thread wait until all threads in the Fibonacci family have terminated. On line 11, the variables \texttt{a} and \texttt{b} are used to initialize the shared channels for the fibonacci family, providing the values that the first thread will read, as well as the pointer to the array to store the results.

In Figure 3.3 the time-concurrency diagram is shown that corresponds with our example, which shows the interactions between threads. \texttt{T_o} is the main thread that executes the \texttt{create}, which then waits immediately using \texttt{sync} on the termination of the created family of threads. The fibonacci threads \texttt{t_2}, \texttt{t_3},...\texttt{t_n} are then started, and all but the first will immediately block and suspend on reading the shared channels. The first thread that received the shared values from the parent can execute, and then passes on the values to the next thread. As Fibonacci requires the value of the \(n-1\)th and the \(n-2\)th iteration, the value from the shared channel \texttt{p1} is forwarded to \texttt{p2} in each thread. Only when its shareds are written, a suspended thread will continue
its execution again. When all threads have completed, the sync in the parent thread completes and it resumes its execution and can now safely use the results array. The writes to \( p1 \) and \( p2 \) by the last thread could be read by the parent again after the sync, but are not used in this example.

3.3 An SVP Many-core: The Microgrid

The Microgrid is a many-core architecture design [CSA17, CSA45, CSA22, CSA30] and [MWvT2, MWvT5], based on the SVP model of concurrency, which has directly evolved from the early Microthreading work as it consists of clusters of Microthreaded DRISC cores. The main research on this architecture is done using a low level architectural simulator [CSA29] that is designed to be a cycle accurate simulation of all the chip components and hardware mechanisms. Additionally, it has recently been verified that the key mechanisms can be built in real hardware as they were implemented as part of a single core FPGA prototype [CSA7, CSA42]. It should be noted that the development and design of this architecture is not part of the research behind this thesis, with the exception of suggested architectural extensions that we will cover in later chapters. In this section, we will discuss an overview of the architecture and its features which forms the main target of our operating system work.
3.3.1 Many simple fine-threaded cores

A Microgrid consists of many simple in-order RISC cores, of which the current designs are based on either the Alpha or SPARC instruction set. This choice is not a requirement nor a restriction for architectures based on SVP; we believe that the instruction set and architectural extensions to support Microthreads can be applied to many other architectures, however applying them to non-RISC architectures is less trivial. An example Microgrid configuration with 128 cores is shown in Figure 3.4.

Each core supports a large number of threads that are scheduled by the hardware. Each thread has its own register context stored in the core which allows context switching between threads on every clock cycle. This low overhead allows data-flow-style scheduling of threads in the pipeline: when an instruction requires data that is not yet available, for example a memory load that has not yet completed, the thread is suspended and is resumed as soon as the data becomes available. To support this the registers act as data-flow synchronizers that we explain in the next subsection. The low overhead context switching and data-flow scheduling of threads allows for latency hiding of long latency operations, as other threads can occupy the pipeline and perform useful work in the meantime. A long latency operation can be anything from a multiple cycle arithmetic instruction (e.g. multiplication, division) to a memory load that can take hundreds of cycles.

A single core would typically support tens of families and hundreds of thread contexts at the same time, which is limited by the size of the on chip data structures to hold family and thread information, as well as the size of the register file. Sizes of 32 and 256 entries for families and threads respectively are currently the default configuration of the Microgrid simulations. As each thread has its own register context, the
core has a large register file (1024 registers) and an allocator in hardware to assign contexts.

The create action of SVP is implemented on the Microgrid by a set of instructions that acquire and set up a family entry and its parameters of the iteration space, and to start the actual creation of threads. As most of these mechanisms are implemented in hardware, a new family of threads can be started within just tens of cycles. An overview of the latency of these operations is shown in Figure 3.5. For example, before the allocation of a family context is completed, the network needs to be traversed twice and it depends on the number of cores in the place (3P). Then setting each parameter takes an additional cycle, and the actual creation will start after 6 cycles and one traversal of the network. Using the presented values we can derive that the minimal latency of a create to start an additional thread locally is 23 cycles. It is worth noting that this is faster than most C style function calls on contemporary architectures, as these would include memory operations to push and pop variables from the stack.

If we assume a delegation network (discussed in Section 3.3.3) with properties similar to the packet switched network of the Intel SCC [131], with a communication latency of 4 cycles per hop, we can calculate the minimal delay for a create to a different place. If we want to start a thread remotely on a 4-core place that is 3 hops away, this incurs a network traversal latency of 3 · 4 = 12 cycles. Therefore, the Allocate step will take 14 + 3 · 4 + 2 · 12 = 50 cycles, and the actual create trigger takes 6 + 12 = 18 cycles to reach the other core. This means the new thread(s) can start after 68 cycles, and the higher level create action completes after 79 cycles. These examples assume that the instruction cache at the target core already contains the instructions that the new thread will execute. Therefore, we can see that fetching the instructions from memory will be the dominant factor in the latency until a new thread starts to run.

### 3.3.2 Synchronizing registers

The register file contains synchronizing registers that can be in either full or empty state implementing non-blocking write and blocking read semantics. These are used to synchronize on long latency operations; the result register is set to empty by the

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency in issue</th>
<th>Latency to effect</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Core</td>
<td>Network</td>
</tr>
<tr>
<td>Allocate (Reserve context)</td>
<td>14+3P</td>
<td>2x</td>
</tr>
<tr>
<td>Configure (Set parameters)</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Create (Trigger creation)</td>
<td>5</td>
<td>2x</td>
</tr>
<tr>
<td>Remote channel write</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Remote channel read</td>
<td>5</td>
<td>2x</td>
</tr>
</tbody>
</table>

Figure 3.5: Latency to complete thread management operations on the Microgrid. Latency at the core is in cycles, and network in the times it needs to be traversed. Latency until the target core is affected is shown in the two columns at the right.
write-back mechanism of the pipeline, and a thread attempting to read an empty register is suspended. When a thread has suspended on attempting to read an empty register and data is written back to it, the register is set to full and the thread is rescheduled again by the hardware. When a register is already in the full state, it can be read and written normally.

![Physical and virtual layout of the integer registers of a family’s threads.](image)

The synchronizing behavior of the register file is also used to implement the shared and global one-shot communication channels. The shared channel is implemented by mapping the shared registers of the previous thread in sequence into a thread’s register context, and is illustrated in the virtual to physical register mapping shown in Figure 3.6. These are initialized in the empty state, and therefore guarantee the synchronizing semantics of the communication channels; a thread reading from its neighbor blocks on the empty state unless it has been written. When a family is distributed over multiple cores, their register contexts are linked through a communication network to implement these semantics. The parent thread of a family has special instructions to put values into and retrieve values from this shared register dependency chain.

### 3.3.3 On-chip networks and memory

Cores are organized in groups, typically of four cores, that share an L2 cache on a snoopy bus [MWvT5], as shown previously in Figure 3.4 on page 42. These caches are then interconnected on a network of hierarchical rings to form a distributed cache based memory system inspired by COMA [68], which provides a single address space,
coherent shared memory to all cores [CSA45, CSA22]. The top level ring connects the memory system to one or more memory controllers which provide access to off-chip memory.

This memory system actually provides more coherency then we required in our definition of the memory consistency model of SVP that we described in Section 3.2.3. Consistency is enforced by an implicit memory-write barrier on create and on the completion of a family before the sync is triggered in the parent context. When all outstanding writes have been committed to the memory system, it is guaranteed that they are visible to any other context due to cache coherency. This means that after a family completes, its updated state is guaranteed to be visible to any other context in the system, which is more than only the parent context as mandated by the consistency model.

Besides the memory network, there are two other on-chip networks that connect all the cores to handle delegations and to support the SVP communication channels. Unlike the memory network and associated protocols, these were not fully specified and simulated yet in the Microgrid simulations at the time of the research covered by this thesis.

The second network is used for delegating work to other cores, and should be designed to have a low latency. Work delegation is not data intensive, as only a few requests for starting new work have to be sent, as we showed in Figure 3.5 on page 43. The bandwidth of this network is much less important than the latency, as the memory network will take care of moving the bulk of data. As work can be delegated
anywhere, we currently assume a mesh topology for this network to connect all the cores.

A third network interconnects the cores for synchronizing communications required to implement the shared and global channels. As these communications happen between neighboring cores, it is designed as a wide linear network to be able to quickly shift these values around. For illustration, a set of 32 cores, their L2 caches and the three discussed on-chip networks are shown in Figure 3.7.

3.4 SVP on pthreads

A second implementation used in this thesis is a software run-time based on POSIX threads (pthreads) that can emulate SVP on commodity multi-core systems, described in [MWvT12]. This was developed by the author in order to be able to experiment with the programming and concurrency model using $\mu$TC, as the lack of a working compiler and continuous changes in the architecture prevented us from targeting the Microgrid directly at the time. We will refer to this implementation as svp-ptl, the SVP POSIX threads library.

Another motivation for svp-ptl was to develop a high level event-driven simulation of the Microgrid. This would aid in the research on resource management and operating systems, as it would allow for efficient simulation of larger systems than the Microgrid cycle-accurate simulator [CSA29] could efficiently support. This simulation should provide feedback from the modeled hardware so that we can adapt program and system behavior on parameters that are not known statically. For example, to adapt the amount of threads or families created depending on resource availability, and this can not be done with static traces. Therefore, we used svp-ptl in a co-simulation to generate events on SVP actions, as a fully functioning application model. While this direction was initiated by the author, this work was much more complex than anticipated, and this simulator only matured recently during the research of a separate PhD project specifically on this topic. Therefore it has not been possible to use it for the systems research described in this thesis. The details about this simulation are considered outside of the scope of this thesis and we suggest our publications on this topic [MWvT7, MWvT6] for further reading.

3.4.1 Implementation

The svp-ptl implementation is based on a C++ library that implements all SVP actions as functions. C++ was chosen over C as its template features made it possible to design the interface to the library as close to the original $\mu$TC syntax as possible. Therefore we did not require to design a parsing translator for $\mu$TC, but we could use a context-free translator based on regular expressions to rewrite the syntax to C++ with calls to the svp-ptl library interface.

In principle, every SVP thread runs as a single pthread; the create action starts a background pthread to implement the split-phase behavior of the create process. This pthread then takes care of starting each instance of the thread function. Internally, it maintains a tree structure of created families, similar to the diagram shown earlier.
3.4. SVP ON PTHREADS

in Figure 3.1 on page 37. In contrast to the Microgrid, which keeps family entries in a fixed size hardware table, svp-ptl has no limit on the number of families but the available memory. The number of supported threads is only bounded by either the number of pthreads the host system supports, the amount of address space, or the amount of memory. The pthreads are scheduled by the host operating system, and are only blocked waiting on synchronizing actions in SVP that are implemented with standard pthread mutex and conditional primitives.

While this implementation supports the notion of a place, its use has no effect on where threads actually execute in svp-ptl. However, it does support exclusive places for mutual exclusion, and it is required to keep track of places when coupled to the high-level simulation. In the next section of this chapter we discuss an extension for distributed systems, where place is used to identify resources on other nodes.

3.4.2 Efficiency

It was not our goal for svp-ptl to be an efficient concurrent run-time system or to promote \( \mu \)TC as a general parallel programming language. We focused on providing a fully functional implementation of SVP, of which the efficiency is dictated by the efficiency of pthreads on the platform it is running on. This means that creating a thread takes about \( 10^4 \) to \( 10^5 \) cycles on contemporary systems, as we illustrate with the following measurements.

Figure 3.8 shows two histograms for measuring the time it takes for a call to \texttt{pthread_create()} to return and the time between the call and the moment the new pthread starts executing. For the measurement the pthreads are created in detached mode. Using joined mode reveals slightly different behavior, but it does not have a significant impact at the order of magnitude of the cost. We refer to [MWvT9] for a more detailed discussion of those results and related measurements.

We compare three platforms: a X4470 [139], a 24-core 2-way hyper-threaded system with Intel Xeon 7530 [84] processors running CentOS 5.5 64-bit Linux, kernel version 2.6.18. Second is a 48-core \textit{Magny cours} system [46] with AMD Opteron 6172 [7] processors, running the same CentOS. Third is a T3-4 [140], a 64-core, 8-way threaded SPARC T3 [136] system running Solaris 10 9/10. All systems run at a clock frequency between 1.6 and 2.1 GHz, and the measurements are normalized to these frequencies to produce a number of estimated core cycles for comparison.

We see in Figure 3.8 that it takes around \( 3.2 \cdot 10^4 \) cycles to start a thread on the X4470, \( 5.5 \cdot 10^4 \) on the Magny Cours, and between \( 1.3 \cdot 10^5 \) and \( 1.6 \cdot 10^5 \) on the T3-4. On two other Intel dual core systems, not shown here, with the more recent 2.6.32 Linux kernel we observed a latency as low as \( 1.0 \cdot 10^4 \) cycles. The large difference between the T3-4 and the Linux systems is likely caused by how Solaris handles pthreads, and not by the architecture of the SPARC T3. We have observed [MWvT12] similar values on older UltraSPARC platforms in the past.

When creating threads is this expensive it is clear that, in order to be used efficiently, this implementation has to work at a different granularity of concurrency than the Microgrid which can start threads in tens of cycles as we discussed in Section 3.3.1. We improved the efficiency by implementing a pool of pthreads. Using this we observed a 10 to 50% execution time improvement in our test cases, but this heav-
Figure 3.8: Distribution of 100000 measurements of time between the call to \texttt{pthread_create} and the start of execution of the pthread on several platforms, normalized to core clock cycles.

Maintaining a central pool guarded by a mutex is not a very scalable approach, even on contemporary multi-core systems such as the three mentioned before. Therefore we investigated using atomic compare-and-swap (CAS) operations to manage the pool. While this improved performance by about 10% on a dual core system, this resulted into a slowdown of three to four orders of magnitude on the three large multi-core systems [MWvT9]. This observation of exploding overhead is similar to the scalability issues of contemporary operating systems that we discussed in Chapter 2, and emphasizes the need for scalable and distributed approaches for software on future many-cores.

Another method to improve the efficiency of svp-ptl is to exploit the property that SVP threads in a family can always be executed sequentially. Fine grained concur-
rency at the leaves of the concurrency tree can be sequentialized until an acceptable granularity is reached. Another SVP run-time implementation named Hydra [CSA32] used this approach, but this was neither the work of the author nor based on the svp-ctl run-time. A problem of this technique is coping with load imbalances when large concurrent families are aggregated to one single sequential execution. Hydra attempts to solve this by splitting the index range of a sequentialized family over different pthreads, but this still leads to an imbalance with families that have an irregular distribution of load between their threads.

A well known solution to such load imbalance problems is to implement a user-space work scheduler that schedules and balances, often using work stealing, the work of SVP threads onto pthreads. It has been shown by the implementations of Cilk [21, 61] that this can be done very efficiently and a similar approach could be used for an SVP run-time. However, it was our opinion that there would be little scientific contribution in such an implementation as the work on Cilk has already shown us 15 years ago that this can be done efficiently.

The second reason we did not implement these proposed optimizations is that they would break the support for our high level simulation [MWvT7]. This requires the SVP threads to be able to interleave similarly to the threads on the Microgrid that it simulates, where each progresses over time, even on the same core due to fine grained interleaving. For example, when a whole family is sequentialized for optimization, the synchronization through shared channels disappears and can no longer be used to send events to the simulation or influence the scheduling of the individual threads.

3.4.3 Application

The primary technical contribution of svp-ctl is that it was the first SVP target that can be programmed with the $\mu$TC language and that runs on top of commodity hardware. This allowed us to program and explore SVP based systems, as well as exploring SVP itself, and made it the primary research tool used for this thesis. For example, it was used to rapidly prototype an I/O system design [MWvT3] (cf. Chapter 7) for the Microgrid architecture; a modeled device interface, a few devices and a prototype software stack to drive them was built by the author using svp-ctl in only a few days.

In terms of exploring SVP itself, svp-ctl was the first platform to support the notion of a place and exclusive place. We used it to experiment with the recursive cleanup actions break and kill, which made us realize that these would be too complex to implement recursively in the Microgrid hardware. Instead, kill on the Microgrid has been implemented to target a place, effectively becoming a place reset. For each use case in which we want to use kill (cf. Chapter 4) we show that this conceptual difference is not an issue, and we will discuss the resulting trade offs needed to cope with this. Similarly, we used it to investigate the use of the squeeze action to pre-empt execution, and the exposed implementation difficulties resulted in abandoning it from the model. We have a more detailed discussion on this in Chapter 4.

It was also used to develop run-time systems on top of SVP. As it was the first to support detached creates, it was used in the development of a prototype of the Graph-
walker [CSA25] run-time for the S-Net co-ordination language [CSA14]. We propose
an alternative SVP based, but related, run-time for S-Net in Chapter 8. Early work
on the compilation strategy [CSA10] for the SAC functional array programming lan-
guage [CSA13] to SVP systems was based [CSA41] on svp-ptl before the compilation
tool-chain for the Microgrid had matured.
As svp-ptl is based on pthreads, it is portable and allowed us to experiment with
applying SVP to several different platforms. It supports Linux (x86, x86_64 and DEC
Alpha), Solaris (SPARC), Mac OS X (x86_64 and PowerPC) as well as more exotic
platforms such as the Microblaze processor in FPGA fabric, running XilKernel or
uCLinux, and the Intel SCC 48-core experimental research processor. The run-time
was extended to support the SVP extensions for distributed memory systems, which
allowed us to construct larger heterogeneous systems of the previously mentioned
architectures using SVP. We discuss this extension and implementation in the next
section.

3.5 Extending SVP for Distributed Memory

As we discussed in Chapter 2, it is a big challenge to scale up both the memory
bandwidth and cache coherence for future many-core architectures. We noticed this
in several benchmarks [CSA21, CSA3] with the current design of the memory system
for the Microgrid that would either saturate the external memory interface, or one of
the on-chip memory ring networks. One promising solution to the bandwidth problem
is 3D stacking of memory [105], which allows for a Microgrid design where each cluster
of cores around an L2 cache has its own local stacked memory, and the ring networks
are only used for transferring data from non-local memories.

This is in line with the trend we observed in Chapter 2, where future many-
core architectures are moving towards distributed memory organizations. Another
example of this is the experimental 48-core Intel SCC chip [81] which has an off
chip shared memory, but no cache coherence. If we have an architecture without
automatic cache coherence in hardware that we want to apply SVP to, we need a way
to implement the SVP memory consistency model across such distributed memories
by identifying which data needs to be made coherent. In this section we will look
at such a uniform extension, which also allows us to apply SVP to other distributed
systems\(^1\), ranging from distributed many-core architectures to networked computing
clusters. The differences between these are in the latency and bandwidth between the
distributed memories, which affects the granularity of computations and data that we
can efficiently delegate between nodes.

We will introduce our extension as follows; first we will discuss the terminology
on how to apply SVP to distributed memory systems, then we isolate SVP program
components that we pick as our unit of distribution, and then we show how these
can be distributed in distributed memory systems. We conclude this section with

\(^1\) Lamport [94] defined distributed systems as: "A distributed system consists of a collection
of distinct processes which are spatially separated, and which communicate with one another by
exchanging messages. (...) A system is distributed if the message transmission delay is not negligible
compared to the time between events in a single process."
discussing an extension of svp-ptl which adds support for distributed memory, which we used as a tool to explore such systems.

3.5.1 Distributed Environment in SVP

We define our distributed environment to consist of a set of computing resources, for example cores, which support SVP in either software or hardware. We define a node to be the largest collection of computing resources that are supported by the original SVP model based on asynchronous shared memory. A node has a single addressable, coherent (according to the SVP consistency rules from Section 3.2.3) access to some memory.

The nodes are interconnected by an infrastructure consisting of one or more, possibly heterogeneous, networks, on which each node can, directly or by routing, send messages that can reach any other node in the system. A place is identified as a subset of one or more (or all) resources within a single node, which therefore inherits the properties that we have just described. We will now give some concrete examples;

- Non coherent NUMA: In a NUMA (Non-uniform Memory Access) system which is not fully cache coherent, a node would be a group of processors that are in a single NUMA domain that is internally cache coherent, which can be as small as a single processor. A place can then contain between one processor and the number of processors within the domain. The Intel SCC [81] is also such a system as it does not provide any cache coherence. All cores have their own private memory area, even though they share memory controllers. On the SCC, both a node and a place can only be a single core on the chip.

- Cluster of Multi-cores: In the case of a networked (e.g. Ethernet or Infiniband) cluster of multi-core machines, each machine is a node and again, a place can contain one or more of the cores of such a machine. However, if these multi-core machines are cache coherent NUMA architectures themselves, one can choose to subdivide these into a separate places that do not span NUMA domains, enabling the expression and exploitation of memory locality.

- Microgrids with 3D stacking: Assuming a Microgrid with 3D stacked memory as mentioned at the start of the section, a node would be the set of cores that share one local memory; for example a group of cores that share an L2 cache in the current Microgrid design. A place can be any set of one or more of these cores.

SVP is already supported within a node, therefore we only need to consider interactions that are remote, i.e. that are between nodes, in order to apply it to a distributed environment. All SVP actions such as create and sync can be trivially translated into messages that can be sent across the predefined network between nodes, and a place identifier can capture the necessary addressing information on which node this place it is physically located. By using a place on a remote node, a create transparently becomes a concurrent remote procedure call\(^2\). Threads in a family created

\(^2\)This assumes that the program code to execute the remote thread function is already present. The mechanisms to ensure this fall outside of the scope of our discussion.
this way can then again create more families there locally, or at some point decide to delegate their child families to other nodes again.

The challenge does not lie in sending SVP actions, but in defining a way to handle the distributed memory organization instead of asynchronous shared memory. We need to define how, and at which level of granularity, we can identify parts of our program that we can delegate to other nodes and which data they require.

3.5.2 Software Components

Using the restrictions that SVP imposes, we can make some assumptions about the structure of SVP programs. Because a program is structured as a hierarchical tree of concurrency, we can assume that most computation, and therefore data production and/or consumption, takes place at the more fine grained concurrency in the outer branches and leaf nodes in the tree. An application can be seen as a collection of software components at the outer branches, connected together with control code higher up the hierarchy, as illustrated in Figure 3.9. Also a subtree with control code and components can be seen as a component itself, as shown at Control 2 in the figure. Due to the restrictions in communication and synchronization that SVP imposes, we can assume that these software components are relatively independent, and therefore are very suitable for delegation to different nodes.

![Hierarchical composition of SVP software components.](image)

Having this view in mind, and by taking the memory consistency model defined in Section 3.2.3, we can make some further assumptions about the communication of data within an SVP program. As communication is restricted at the family level, where a thread can communicate data through the shared and global channels to a family it creates, we can make the following observation; The created threads will, disregarding global static references, only access data that is either passed through these shared or global channels, or data in memory that is accessed through a reference that is passed this way. Newly generated data that needs to be communicated back to the parent, has to be passed back again through the shared channel. Therefore, we can claim that the data dependencies of software components are identified by the shared and global channels to the top level family of such component. Threads accessing
objects in memory through a global reference are the exception to this, but they have to be delegated to a specific exclusive place in order to guarantee consistency, or are local read-only data.

Our strategy for supporting distribution of SVP programs is based on the previous observations; we can identify a software component at the level of a family and its children, that can be delegated to a remote nodes with a create using a place on that node. This component can then internally create more threads locally on places on that node, or can decide at some point to delegate further sub components to other nodes. However, the whole component has a single interface at its top level family, and its dependencies are identified by the shared and global channels to that family.

### 3.5.3 Distributed Memory

When we delegate across nodes only at the granularity of families, we can use the information contained in the channels to the created family to determine which data needs to be transferred. At the point of create, we synchronize or copy all objects that the family receives references to, to the node it is created on. As all threads of the created family run on the same place and therefore within the same coherent memory, such replication is not required for internal communication of objects between sibling threads. When the family completes, at the point of sync, they are synchronized or copied back again, taking into account newly created references the family might send back through its shared channels. The second case, in which a family updates global state on an exclusive place is not an issue; as each family accessing this data is created on the same exclusive place, it shares the same consistent memory, and no data communication is required besides the earlier defined inputs and outputs.

Such as approach delivers less consistency than the original consistency model, as only the memory areas that the child family will use are made consistent. However, this approach is often too naive; for example, it does not keep track of how data is used. Depending on data being consumed or modified by the created family, we would like to avoid communicating back unmodified data for efficiency, so an implementation has to detect or receive hints on which data has been modified. Furthermore, on more complex large objects, e.g. a linked data-structure, it is unclear if we suffice with a shallow copy or if we need to do a potentially expensive deep copy of the object. Another issue is dealing with objects of which the size can only be determined at run-time.

The decision to make shallow or deep copies of objects very much depends on the application, so we decided not to make this part of our distributed SVP definition. Instead, we observe that having fine grained control over the input and output data of a family would be desired. Using these, higher level mechanisms can be constructed. For example by extending object references with place identifiers, a software mechanism could be constructed that retrieves the referenced object from the specified place only when it is accessed, as a place also identifies a memory that belongs to the node it is part of. Such a fat reference to an object would be similar to the global pointers used in Cid [115].

Having extended control over inputs and outputs requires a programming language where we can either analyze or specify in detail which data goes in, and which data is
generated or modified by a software component. In the next sections we will discuss our extension to svp-ptl for distributed memory which use μTC, a C based language, in which this analysis is infeasible, and consequently we leave it to the programmer to explicitly specify this. After all, the designer of a component has the best knowledge of what its inputs and outputs are, and higher level languages such as SAC and S-Net should have sufficient knowledge to generate this automatically when compiling or executing on top of SVP.

3.5.4 Extending μTC: Data description functions

In order to explicitly specify the inputs and outputs of an SVP component, we introduce the notion of a data description function. They are similar to input and output definitions defined on objects for example in Amoeba [153] and Modula-2+ [129], or for functions in CellSs [14] or Sequoia [57]. However, our input/output definition is not just a set of parameters but is a function itself, which allows for greater flexibility.

```
1 DISTRIBUTABLE THREAD(fibonacci)(int p1,
2 int p2, int* result, int N)
3 {
4     INPUT(p1);
5     INPUT(p2);
6     for(int i = 2; i < N; i++)
7         {
8             OUTPUT(result[i]);
9         }
10 }
```

Figure 3.10: Data description function for Fibonacci

A data description function is a powerful way of expressing data dependencies and controlling which data goes into and comes out of a family of threads. They are specified as a special function for each thread function which describes the inputs and outputs using special statements, allowing the corresponding thread function to be distributed to other nodes by our extended svp-ptl run-time. This function receives the same arguments as the thread function, and is called by the implementation at the creating and completing stage when the corresponding thread function is executed on a remote node. The data description function contains INPUT(v), OUTPUT(v) and INOUT(v) statements, which trigger data transfers at the different stages. Data tagged with INPUT is copied to the remote node at the stage when the thread function is started by a create, and OUTPUT data is copied back to the creating node at the stage when the created family finishes and sync completes. INOUT is a shorthand notation for the combination of the previous two.

Within these data description functions, loops and conditional expressions can be used around the statements describing input and output. This provides the flexibility needed in order to express the dynamic nature of family input/output data, for example dynamically sized arrays or the traversal of more complex data structures. In
3.5. EXTENDING SVP FOR DISTRIBUTED MEMORY

Figure 3.10 on page 54 we show how we can make the Fibonacci example code shown earlier in Figure 3.2 on page 40 distributable by defining such a data description function. The initial values of the shared channels are only used as input to the Fibonacci function, and the array with the generated sequence is sent back as output. Please note that we needed to add the size parameter to the thread function to support a non-fixed size for the result array.

3.5.5 Distributed SVP Prototype

The third implementation that was used for the work of this thesis is the extension of the svp-ptl run-time for distributed memory. The initial version of this extension was developed jointly with VTT Research as a part of the ÆTHER project, and further developed by the author to support different platforms as well as to improve integration with the original run-time.

Initially the implementation used SOAP web services [24] for communication and XDR [53] for (de)serialization of data. The implementation of the data description functions took care of packing and unpacking the data with XDR, and this packed data payload is then sent using SOAP together with a request to create or sync a family. XDR also took care of the necessary pointer rewriting when sending over linked data structures, as well as mixed data representations in a heterogeneous mixture of systems. This implementation was used as the middle-ware of the ÆTHER platform in combination with a resource management system that we discuss in Chapter 6, and that were presented in [MWvT14].

Being integrated into svp-ptl, this implementation suffers from the same level of overhead for exposing local concurrency, but using the combination of SOAP and XDR incurs a lot of additional processing and communication overhead. Therefore, we changed the communication layer to a modular design and implemented a more lightweight TCP/IP socket protocol. The modular approach also allowed for other possible back-ends, for example an Infiniband/RDMA based implementation.

This modularity was again used to port this implementation to the Intel SCC 48-core experimental processor [81]. We added a communication back-end to efficiently use the SCC specific features. As cores are abundant on the SCC, we proposed methods to use dedicated cores for efficiently copying memory between cores that are doing computations [MWvT10] and we developed a second approach where cores can map to the memory of another core to fetch the required data using only a single copy operation [MWvT1]. As the SCC consists of a homogeneous set of cores, we could also omit the XDR (de)serializations by directly accessing our communication primitives within the implementation of the data description functions, reducing communication overhead.

Covering further technical details and performance characteristics of the implementations and their different communication channels on the mentioned platforms is not of sufficient interest for this thesis. However, the discussed implementations show that our distributed approach for SVP is generic enough to be applied to a broad range of distributed platforms, as we claimed in Section 3.5.1.
3.6 Summary

In this chapter we have introduced and described the SVP concurrency model. We have shown how it can be extended to distributed memory architectures, and we have discussed several implementations that are based on it; the Microgrid many-core architecture that support SVP in hardware, and the software svp-rtl run-time and its extension for distributed memory that can run on a wide spectrum of platforms.

We have discussed the Microgrid which is a scalable many-core processor architecture with hundreds of cores that supports the SVP model in hardware, tightly coupling the cores with mechanisms to create fine grained concurrent hardware threads on multiple cores with a very low overhead. We then presented our software implementation of SVP which attempts to mimic the execution model of the Microgrid, which allowed us to develop systems when the full tool-chain for the Microgrid was not yet available. This also allowed us to apply SVP to other architectures, which lead to a distributed implementation of SVP that can be run on distributed memory architectures such as clusters or the Intel SCC. For this implementation we developed a definition of an SVP software component, with defined inputs and outputs, which is the granularity at which we can distribute computations across memory boundaries.

These three implementations form the skeleton on which the research presented in this thesis is conducted; all systems and mechanisms that we have developed were based on these implementations.
CHAPTER 4

Processes and Synchronization

4.1 Introduction

In this chapter, we will start with taking our list of minimal operating system abstractions from Chapter 2, and see how these can be implemented based on the mechanisms provided by SVP that we described in Chapter 3. In this chapter we focus on providing the fundamental support for multiprogramming in the form of process management and synchronizations, as this is the primary task of an operating system as we have seen in our historical perspective in Chapter 2. This chapter will cover the definition of a process on an SVP system, starting and scheduling of processes and synchronizations that are the basis for inter process communications. In most of the traditional operating systems since Multics, processes are combined with address space and protection management, where a process is tied to an address space which is protected from accesses by other processes [45]. However, we defer the discussion on protection and address space management until Chapter 5.

4.2 Processes & Scheduling

In traditional systems, a process is a virtualization of a computer, which contains one or more schedulable entities, threads of execution, where each thread can be seen as a virtual processor in this computer. The distinction between processes and threads is dictated by address space organization, where processes each have their own address space, and multiple threads within a process share an address space. Please note that this definition that binds processes and address spaces is not the only possible way to define a process, but it is the most common approach. Alternatively, more orthogonal approaches to threads and address spaces have been mentioned in the related work in Chapter 2.

While in traditional systems the operating system is responsible for scheduling threads and processes onto the physical hardware, the scheduling and handling of threads is already supported directly by the platform that implements SVP. Effec-
tively, we could say that the Microgrid contains a kernel that provides the layer 0 abstraction of Dijkstra’s design [50], which is directly implemented in the hardware. We will look at the resulting implications for scheduling in the next few sections, but we first need to define what we identify as a process in SVP based systems.

### 4.2.1 Defining a Process

The SVP model makes no distinction between threads and processes; everything is a thread in a family that is started with a create action somewhere in the hierarchical concurrency tree. Starting a new process can therefore be implemented trivially by loading the executable into memory, potentially including dynamic linking, followed by a create action to start its execution. However, we need to introduce process boundaries in order to support reclamation, isolation and accountability [CSA38]. This requires making a distinction at some level in the concurrency tree, based on other criteria such as ownership, protection management or resource allocation. We identify a process to be the sub-tree starting from a certain family in the concurrency tree, started by a different process with a create action. This definition corresponds with our earlier definition of an SVP component in Chapter 2, and therefore we can define a process to be a collection of one or more components, which are executing or have state on one or more places that were allocated to the process.

There are several reasons why we decide to define our processes at this granularity. First of all, defining processes at the granularity of individual thread contexts is counter intuitive, as a family of threads expresses a certain grouping of a set of activities. Furthermore, this would no longer allow the very fine grained thread scheduling in the Microgrid when every thread switch could mean a full context switch to a different process with different protection and access rights. Similarly, when we would treat any family as an individual process we would have the same problem on the Microgrid with multiple families that are mapped on the same core, resulting in a mix of threads crossing process boundaries. Additionally, cheap creation of families would no longer be possible if every family would be a new fully blown process. Therefore, when we raise the granularity level of a process one step further, only a delegation to a different resource (place) can be a crossing into a different process domain. This means that the whole SVP model works normally within a place, with all advantages of efficient low level concurrency on the Microgrid, where a place corresponds with a group of one or more cores. Following from this approach comes a requirement for scheduling where computational resources (i.e. a place) are assigned solely to a single process.

We now continue to further investigate the scheduling issues this imposes, where we make a distinction of fine grained scheduling of threads within a process and coarse grained scheduling of processes.

### 4.2.2 Fine grained scheduling

Fine grained scheduling of threads is done by the underlying implementation that supports SVP, where the scheduling is constrained by the blocking synchronizing operations of SVP such as sync, reading an empty communication channel, or waiting for
an exclusive place. SVP does not define any other scheduling constraints or required
equality in the scheduling between threads or families, or the guarantee that they will
execute in parallel. For example, when the technique is applied where threads within
a family are sequencealized to adopt the granularity of concurrency, they clearly will
not execute in parallel even though they have been expressed concurrently. However,
it should be safe to assume a certain guarantee of progress between two SVP compo-
nents that are delegated to two different places, as these represent hardware execution
resources.

When we look more concretely at the Microgrid, a core has a guaranteed fairness
in scheduling between the thread contexts that are able to run. Each thread context
will execute for a maximum of 15 instructions (one I-cache line) before a switch to the
next context is made to avoid stalling the pipeline on I-cache fills. Besides this, it uses
a data-flow style scheduling between thread contexts where a context is suspended
when it needs to wait for data of a long latency operation, and only is rescheduled
when the data becomes available. However, when a core is low on resources, such
as thread or family table entries or registers, the number of concurrently scheduled
threads at the leaves of the concurrency tree is reduced accordingly with reverting to
sequential execution being the most extreme case.

4.2.3 Coarse grained scheduling

As an SVP implementation takes care of the fine grained scheduling of threads on a
place, within a process or SVP component, the question is how to implement coarse
gained scheduling of processes in an SVP based system. Due to the way we have just
defined processes it is clear that this coarse grained scheduling has to be based on
how places are assigned to processes and how these are scheduled. If it is possible to
virtualize multiple places on physical resources is left up to an SVP implementation,
which is for example not the choice for the current design of the Microgrid. As
the Microgrid implementation focuses on low overhead concurrency management, it
currently does not support pre-emption or virtualization of thread and family contexts
in the core to keep the core logic as simple as possible. Also one would not want to
mix threads of different processes in the pipeline as this could require heavy context
switches between different processes potentially on every other instruction executed,
which no longer allows for the efficient fine grained thread scheduling. An advantage
of the approach to schedule at the place level is that only thread switches within a
process are made on a resource, and we no longer require a kernel with a scheduler
to run on every core to manage our processes.

A good analogy here is to see the assignment of a place to a process as assigning
a process a guaranteed time slice on a core in a contemporary operating system.
From that perspective, the most straightforward way to implement coarse grained
scheduling would be to virtualize places of processes onto the physical resources.
However, this requires a mechanism to pre-empt the execution of a component on a
place, and there is currently no interface defined in either SVP or its implementations
to do this. The definition of SVP in principle does not prohibit the pre-emption of
threads or places, for example, it is not a problem under svp-ptl that threads get
pre-empted by the host operating system. Having pre-emption on places is similar to
the pre-emptible cells in Tessellation [104], or how the CM-5 [78] could pre-empt an entire partition of nodes through the use of its control network [97].

There are certain limitations and pitfalls to what SVP implementations can do to implement pre-emption. First, we will discuss why the originally proposed squeeze action to pre-empt a component was not sufficient in Section 4.2.4 followed by an analysis that shows that it is not trivial to add support for pre-empting the execution on a Microgrid core with the current design in Section 4.2.5. After that, we will look at the alternatives of what we can achieve without pre-emption support.

4.2.4 The problem of squeeze

The squeeze action was similarly defined as the family based kill in earlier SVP publications [CSA20]. It terminates a specified family, but unlike kill, it does so in a graceful way bringing the family down into a deterministic state so that it could be restarted at a later stage. Its semantics are best described with the following example.

We assume the action targets a family with a large iteration space, of which not all threads are executing concurrently, either to limits of resources or the block parameter. Then, the process that starts new threads when others complete is stopped, and the existing threads in the family continue to run until completion. After they complete, the state of the shared and global channels is saved and the index of the next thread in sequence that has not run is returned. This way, the family can be restarted by a create of the same thread function but with the iteration space starting at the index returned after completing the squeeze. The shared/global channels are re-initialized with the stored values, and the execution of the family resumes.

```c
1 thread t_sum(int* array, shared int sum)
2   index i;
3   sum = sum + array[i];
4
5 int sum(int* array, int size)
6   int sqz_idx = 0, sum = 0;
7   create(fid ;; 0; size ;; 5;& sqz_idx ) t_sum(array , sum);
8   squeeze(fid);
9   if(sync(fid) == EXIT_SQUEEZE)
10      create(fid ;; sqz_idx ; size ;; 5) t_sum(array , sum);
11      sync(fid);
12   return sum;
```

Figure 4.1: Pseudo μTC code to show pre-emption using original squeeze action

We clarify the semantics of squeeze further with the example pseudo μTC code shown in Figure 4.1. This code computes the sum of variables in an array, then attempts to pre-empt this process with squeeze, and if this is the case, resumes it again from the point where it was pre-empted. The initial create on line 7 takes an additional reference to a squeeze index parameter named sqz_idx which will store...
the index at which the family was pre-empted when it was squeezed. At the \texttt{sync} we test how the family terminated, as it is possible that it already finished before it was squeezed. When it was squeezed, the \texttt{sqz_idx} contains the index of the next thread that was not started yet, so this is used again as the start of our iteration space to continue execution. The variable \texttt{sum} holds the last value in the shared channel, so is used again to initialize the shared channel for the second \texttt{create} of the computation.

While it is clear how \texttt{squeeze} can be beneficial to pre-empt such large families of threads, this example already directly highlights some of its problems. First of all, it can only be applied to families of threads where only a subset of the iteration space is scheduled at a time. Secondly, as the currently running threads are allowed to complete, the pre-emption of a family by squeezing it can take an unbounded amount of time. A third problem is that the parent family needs to be aware that the child family can be squeezed; it needs to take care of this by catching the squeeze index value and storing the value of the shared channels for when the family is later to be resumed.

However, the problem becomes even more complex when we try to apply \texttt{squeeze} recursively to families of threads that have created child families themselves, i.e. by applying it not to just a single-level family but to a whole branch of the concurrency tree. Of course we could again let every running thread in the top family and all its created children run until completion just as in the previously discussed situation, but to have a more efficient pre-emption we would want to recursively \texttt{squeeze} these child families as well. But when we decide to propagate \texttt{squeeze} to child families, it means that threads higher up the concurrency tree need to be re-run when the squeezed families should resume, as otherwise the deeper nested families that are squeezed can not re-created.

1 \texttt{thread t\_sum(int* array , shared int sum)}
2 \hspace{1em} \texttt{index i ;}
3 \hspace{1em} \texttt{sum = sum + array[i] ;}
4
5 \texttt{thread t\_spread\_sum(int* array , shared int sum)}
6 \hspace{1em} \texttt{index i ;}
7 \hspace{1em} \texttt{int localsum = 0 ;}
8 \hspace{1em} \texttt{create(fid ; ; i*2 ; (i+1)*2 ; ;) t\_sum(array , localsum) ;}
9 \hspace{1em} \texttt{sync(fid) ;}
10 \hspace{1em} \texttt{sum = sum + localsum ;}

Figure 4.2: Pseudo \(\mu\)TC code of a parallel reduction

Let us consider the parallel reduction code in Figure 4.2 which spreads out the reduction into parallel pair wise adds followed by a reduction over the result. When we want to recursively \texttt{squeeze t\_spread\_sum}, it needs to take care of testing the return code at \texttt{sync}, and in case it is squeezed store the state of the child \texttt{t\_sum} family. A thread in \texttt{t\_spread\_sum} can be in either of three states at the moment of squeeze; it can have completed, it can not have started yet, or it can be running. In the first two
cases nothing has to be done, but when it is running we need to propagate \texttt{squeeze} to any child families if they exist. Then, we must find the index of the first thread in the sequence that was still running and has propagated \texttt{squeeze} to a child. This is the index at which \texttt{t\_spread\_sum} should be restarted later on. The code should be altered to detect that it is being re-run, and instead of starting the computation normally, load the stored state of \texttt{t\_sum} and continue from the point it was squeezed.

1 \texttt{thread t\_func(shared int a, shared int b, shared int c)}
2 \hspace{1em} \texttt{int a\_in = a;}
3 \hspace{1em} \texttt{create(fid ; ; 0 ; 100 ; ;) t\_foo(a\_in);}  
4 \hspace{1em} \texttt{sync(fid);}  
5 \hspace{1em} \texttt{int b\_in = b;}  
6 \hspace{1em} a = a\_in + b\_in;  
7 \hspace{1em} \texttt{create(fid ; ; 0 ; 100 ; ;) t\_bar(b\_in);}  
8 \hspace{1em} \texttt{sync(fid);}  
9 \hspace{1em} \texttt{int c\_in = c;}  
10 \hspace{1em} b = a\_in + b\_in + c\_in;  
11 \hspace{1em} \texttt{create(fid ; ; 0 ; 100 ; ;) t\_baz(c\_in);}  
12 \hspace{1em} \texttt{sync(fid);}  
13 \hspace{1em} c = a\_in + b\_in + c\_in;

Figure 4.3: Pseudo $\mu$TC code of a thread function that is very complex to \texttt{squeeze} and resume recursively

A problem here can be the state of shared channels, especially in the case where a family contains multiple of these channels that are not written to at the same time just before the thread ends, but in between computation stages, for example as shown in Figure 4.3. When we \texttt{squeeze} the \texttt{t\_func} threads, they can have already written some but not all of their shared channels, as they depend on values produced by one of the child families that can be pre-empted. In such a case, we would not know which value to save to later on re-initialize the shared channels. This requires families that can have such restartable threads to have even more complex check-pointing code and conditional reading or passing on of shared channels to cope with this that is all but straightforward.

Our conclusion is that using \texttt{squeeze} becomes too complex when applied recursively. Furthermore, it relies on the program to save its state properly, include code to resume properly, and to promise to pre-empt \textit{as soon as possible}, which gives no guarantees. Therefore we can say that \texttt{squeeze} is not a viable approach for pre-emption on SVP systems, and due to its complexity of use we decided to abandon it.

### 4.2.5 Pre-emption on the Microgrid

In traditional systems and processors, pre-emption is implemented by raising an interrupt, often based on a timer, and then saving the register state in the interrupt handler. The scheduler is entered and another process is loaded by setting the appropriate stack pointer, page tables and program counter amongst other things. While
we have proposed interrupt and exception mechanisms for the Microgrid [MWvT8] before, they are based on executing a handler thread in its own context as it is unclear which of the many threads should be interrupted when an external event arrives. Furthermore, this mechanism has never been implemented in the architecture up to now, nor any other mechanism to support pre-emption. However, if we assume such an exception mechanism is available and we can access all thread contexts as well as thread and family table state from an exception handler, we will now show why this is still not enough to implement pre-emption on the Microgrid.

In order to pre-empt a Microgrid core, all relevant processor state must be saved elsewhere, usually in main memory. This includes the family and thread tables, as well as the register file. First of all, a mechanism must be introduced to stop the execution of all thread contexts within the place, for example by stopping the scheduler and draining the pipeline. Note that all contexts should be stopped at the same time and can not be stopped and saved individually, as otherwise registers that were already saved could be written by other threads on that place as part of the synchronization channels. Then, the process that takes care of saving the state, running in the exception handler context mentioned before, should wait for all bounded long latency operations to complete. For example, when threads are suspended on receiving values from ALUs or memory. Assuming that this special context can access any thread’s registers, this can be done by reading every register and suspending until it is written.

However, we identify the following special cases that should be handled with care, as they regard unbounded latencies:

1. Reading from registers belonging to a Shared/Global channel that are still in empty state
2. Reading from family synchronization registers related to child families
3. Parent contexts on a different place writing Shared/Global channels
4. Child families on a different place writing their synchronization register

The first case can be solved by identifying for each context which registers are mapped to the channels, which is contained in the corresponding family table entry, and having a special test instruction to test if a register is empty without suspending. The pre-empt mechanism needs to store this additional information, and the resume mechanism should be able to initialize the register to empty again. Case 2 is a similar problem, as attempting to read that register is in fact the sync action, and therefore an unbounded operation as it depends on the completion of a child family. For this, the same test mechanism can be used.

However, a bigger challenge lies in asynchronously arriving messages for the communication channels or related to sync. Case 3 can expose a WAR (write after read) hazard: the parent context can attempt to write to one of the channels of a thread while the context switching process has already has stored its state. Case 4 is a similar WAR problem but then coming from a child family on a different place. To avoid the WAR problems while pre-empting, the delivery of these messages will have to be delayed until the pre-emption has completed. Then, they can be handled just as
Asynchronous messages that arrive after a place has been pre-empted have to be dealt with in some way. Besides the ones concretely related to pre-empting a context, also a message for a `create` or `kill` could arrive at that place. These messages can not be buffered at the core as the size of such a buffer can not be bounded, the size would depend on how many program contexts you would multiplex on a place. Therefore, they have to be handled some way at the core either by a software or hardware mechanism. This could be done by introducing another exception handler which updates the stored state of the pre-empted program context according to the incoming message. However, this means that such messages need to contain additional information to uniquely identify to which corresponding context they belong, when multiple are multiplexed on a physical place using pre-emption.

When we implement these pre-emption mechanisms in software using exception handlers, this means that these handlers will have to execute with different privileges. This in order to make sure that a process running at a place can not access the pre-empted state of an unrelated process that previously occupied that place. Also the use of instructions to access the registers of other thread contexts and the thread and family tables should be restricted. However, the current design of the Microgrid core does not support different privilege levels.

The conclusion that we draw here is that the synchronizing memory of the Microgrid (i.e. the register file) can not be virtualized in the current design, even when assuming support for exceptions. Therefore, pre-emption support requires quite some additional non trivial mechanisms and modifications that need further architectural research that lies outside the scope of this thesis. A sufficient fine grained support for pre-emption can also enable the virtualization of the bounded thread and family table resources in the Microgrid core; or conversely, if these can be virtualized (this has to include the associated registers), we also have the required support to virtualize a place by simply swapping out all family and thread contexts.

It should be noted that the lack of pre-emption and virtualization has been a deliberate design decision for the Microgrid in an attempt to keep the core design as simple as possible. This problem of pre-emption in an asynchronous system is also much more general, for example, it is also an issue in distributed operating systems that support migration such as V [36], where incoming messages and events for processes that are being migrated need to be deferred and forwarded. It is also a more general challenge for other possible hardware implementations that want to support SVP, and care should be taken to allow the virtualization of the special synchronizing memory. This is not an issue in software implementations such assvp-ptl, as they implement synchronizing memory in normal memory, which is not a problem to virtualize.

### 4.2.6 Space-sharing with SVP

As we have seen in the previous sections, time-sharing at a coarse grain level is only possible if an implementation of SVP has some support for pre-emption of resources. However, this does not necessarily pose a problem for designing an operating system
for the Microgrid, though it induces a more strict bound on resources. In this section we explore an alternative strategy that does not require support for pre-emption or virtualization.

When there are sufficient interchangeable computational resources available, i.e. general purpose cores on the Microgrid, we can multiplex programs in space rather than in time, taking a space-sharing approach for scheduling. In fact, we prefer space-sharing over time-sharing for future many-cores as it is counter productive to move state in and out of cores at pre-emption as it pollutes caches and consumes time, energy and bandwidth that is then not spent on computing. A space-sharing approach is also advocated by the work on Tessellation [104] and fos [159]. They have dedicated resources for system services, and allocate dedicated sets of resources to applications.

When interchangeable computational resources are ubiquitous, as memory is these days, we can have programs dynamically request and release computational resources similar to how we manage memory. Furthermore, operating system services can reside on dedicated resources, not interfering with programs. One of these services manages resources by handling the requests for places by applications. This service, that we named the SEP (System Environment Place), with its resource management and protocols are discussed in more detail in Chapter 6. The resource assignment policies implemented by this resource manager facilitate the high-level scheduling and mapping of processes which form our space-sharing approach. Effectively this still gives a kind of time-sharing as well, but it will be on a more voluntary basis as processes acquire and release resources as they need them.

However, the people from Barrelfish [120] argue that space-sharing alone is not enough, and time-sharing will still be necessary even in future many-core architectures. They give the following three reasons:

- Multiple applications will compete for resources, and might each be able to expose enough concurrency to exhaust all of them.

- Architectures will be heterogeneous and a scarce type of resource required by multiple applications needs to be time-multiplexed.

- Interactive workloads will cause varying loads and time-sharing gives a means to provide QoS.

They add to this that these points imply that static partitioning of resources when an application starts is not enough to manage resources, as demands change over time. It is clear that indeed a static partitioning is not sufficient, and therefore we introduced our approach of dynamic resource allocation which can overcome this problem. We will now discuss how these three cases can be handled in our space-sharing strategy.

When there are multiple parallel applications that can exhaust all resources they will compete for them at the resource manager, both in time and space. However, it is more efficient to reduce the amount of exposed concurrency of an application to match the amount of resources it gets assigned, instead of time-multiplexing all applications over all resources which gives the additional context switching overhead and cache trashing penalties. Besides efficiency for locality reasons this also provides a form of QoS (as requested by the third case) by having dedicated computing resources
instead of guaranteed time slices, as long as resources are ubiquitous. When they are not used, they can be powered down to save energy, while retaining their state so that they can quickly be activated on demand.

The second point however, is the most important. In any case an arbitration is required on such scarce resources, and this would be done by the resource managers that handle dynamic allocations and deallocations of these resources. One alternative to providing pre-emption is to constrain the allocation by a contract in terms of time and/or energy, and by terminating or constraining a delegation if the contract terms are exceeded. The problem with this approach however is that it requires a worst-case execution time analysis on the component that would have to run on such a resource to guarantee that it could run to completion, which might not be feasible. However, without pre-emption it is the only way to guarantee that an application can not allocate a scarce resource and never release it again until it terminates.

In addition to this, when an application can request specific functionality from the resource manager, the latter can decide either to queue the requests for the resource that provides this, optionally using the strategy outlined above, or to assign a different resource that can emulate the functionality at a cost of performance. Whether this trade off can be made depends on the requirements of the application, and a resource manager approach that supports this is discussed in the second half of Chapter 6. An approach with specific functionality (as an SVP component) can also provide the guaranteed bounds of execution time required by our first strategy, for example by allocating the resource for only one invocation. When the allocation and invocation of the component are combined into one request, it can be guaranteed again that the resource will be freed up again and can be re-used for another application. Therefore, we can say that it is only required for a heterogeneous SVP platform to support pre-emption on certain scarce resources, when it can not guarantee that they are only occupied for a bounded amount of time using one of the described strategies.

Another class of applications are periodic processes or processes that are triggered by events. Depending on their nature such processes could be suspended as a continuation [52] stored in memory, as memory state and a reference to an entry point in the form of a thread function, and registered at a system service. This service will then have the responsibility to allocate a resource and start the continuation with a create action on it when such a process is triggered. However, if such a process needs to meet some QoS with a specified response time this might not be feasible, and a resource will have to be reserved to guarantee this. This approach can be seen as a software analogy of the hardware mechanism in the Microgrid where a thread can suspend on a register, with the advantage that the suspended process is only stored in memory and does not take up any resources in a core such as a thread or family table slot.

The biggest obstacle for the space-sharing strategy that we have discussed in this section is what to do when we run out of resources, as there is no mechanism to easily revoke resources from processes. First of all, the policies of the resource manager on handing out resources to processes should already attempt to prevent such a situation. It should be noted that it is perfectly valid for SVP programs to receive smaller places (less cores) for an allocation request, as they will still be able to run, unlike the case with memory allocations. Which brings us to the similar situation; what can an
operating system do when it runs out of memory? Surely it can page memory to disk, but this usually causes the system to grind to a halt on disk I/O, and it only temporarily stretches the limit. Many contemporary operating systems will resort to invoking the *Out-Of-Memory Killer* routine which selects and kills a process to free up memory, clearly, a similar strategy can be applied to processing resources.

It is clear that simply killing a process when all resources are in use is not always the right choice. When for example there are non critical processes running that optimistically allocated all unused resources, one would not want to immediately kill those (or even worse, kill a critical process by accident, something that has often been observed with the OOM-killer in Linux [125]) when more processes are started. Instead, such applications should support an up-call mechanism where the system can notify it to release some resources within a certain grace period, and kill the process when it does not comply, similar to the design of the user level paging mechanism of Nemesis [70]. Support for this should be part of higher level run-time systems (cf. Chapter 8) and their interactions with the resource manager (cf. Chapter 6).

During our research it was also suggested that checkpointing could be used in combination with *kill* to support a crude form of pre-emption and resource reclamation. In such a setting, applications or some of their components could be removed from their resources by a *kill* on their place, which can then be restarted later or elsewhere using a previous checkpoint. However, we see several issues with such an approach. First of all, the state of a process needs to be entirely self contained, i.e. no communication with other processes, or with the user, unless this can also all be transaction based. Secondly, the complexity of restartable code quickly approaches the complex restartability problems that we saw were required for the suggested *squeeze* action in Section 4.2.4, and it is not clear at which granularity this should be done. At a fine granularity this will be very complex and deliver much overhead, but at a coarse granularity a process might not be able to make any progress when it gets pre-empted regularly this way.

In any case, independent of which strategy and policies are implemented for space-sharing, it is clear that the overhead for resource allocation should be low to efficiently facilitate dynamic resource allocation to processes. To illustrate the strength of such strategies for the Microgrid, an experiment with a simulation of a 128-core Microgrid chip has shown that it takes just 4 microseconds to allocate a cluster of 64 cores, delegate a small compute kernel which creates and executes 4K threads on that cluster (Livermore Kernel 7) and to synchronize on completion, assuming a 1GHz clock. This is considerably less than the time slice of a conventional time-sharing operating system.

### 4.2.7 System Services & Identification

Since we have now determined our approach for scheduling, it is important to define how a process can interact with the system. First of all there are the low level controls to create threads, but, in addition, system services need to be contacted for example to allocate a place, do I/O, and to set up higher level synchronizations. To use an analogy with microkernel operating systems, the SVP implementation can be seen as
the "kernel" of the system, on top of which the rest can be implemented as system services.

A service does not need to be a running process that is continuously listening to service requests, but can be a continuation [52] which operates on some stored state, that is started concurrently with a create delegation to a system place. Needless to say, such service functions need to be re-entrant in such a highly parallel system, and require critical sections to safely operate on their state. Such critical sections are supported in SVP by using exclusive places or higher level locking and synchronization primitives that we will discuss in Section 4.3.

Another important thing required to implement system services, is to be able to determine the identity of the process that delegated to the system place to access the service. As a process consists of many families of threads that can be arbitrarily spawned and terminating, we can not use the identity of the parent family that did a create to the system service. However, as resources are allocated to processes, we can identify a process by the collection of places allocated to it. Therefore, it is sufficient to determine from which place the delegation arrived. This means that having support in an SVP implementation for something like a get_parent_place() function which returns a reference to the place of the parent should be sufficient. As the place can only be owned by one process, the process identity can be determined by asking the resource manager to whom the place was allocated.

4.2.8 Summary

To wrap up this section, we summarize our vision on processes in SVP based systems. Processes are composed of components, composed of families of SVP threads, and a distinction is made at an arbitrary level based on protection, resources, or ownership. They are allocated a starting set (place) of resources, and can dynamically request and release resources by contacting the SEP, the resource management service. This and other services are instantiated by a create to a system place, distinct from the places that run regular processes. Higher level scheduling is controlled by the resource management services, implied by allocation strategies and by imposing constraints on the energy or duration of an allocation.

This is in contrast to classic approaches with time slicing, and we have discussed the issues with implementing such a classic approach on SVP. This is especially a problem on the Microgrid as it has a bounded, non virtualizable, synchronizing storage. Concurrency resources are therefore also bounded and not virtualizable, so we will assume for the rest of this thesis that the Microgrid hardware either provides sufficient resources and that our proposed space-sharing strategies are sufficient, or that the hardware will be extended to support pre-emption and virtualization of resources in the future.

4.3 Synchronizations

One of the core features required by a concurrent system is a mechanism for synchronizations. At the lowest level, these are usually based on atomic test and set
4.3. SYNCHRONIZATIONS

operations supported by the underlying architectures such as fetch and add or compare and swap instructions to atomically read and update a counter to implement Dijkstra’s semaphores [51]. The most common low level use is a spin lock, to implement a binary semaphore, or mutex, to guard access to a critical section or exclusive access to shared state. Every actor that tries to access this state attempts to acquire the lock by performing an atomic test and set, and when this operation fails it simply repeats it until it succeeds, hence the name spin lock.

As we have seen in Chapter 2, such spin locks can have several scalability problems in systems with many hardware contexts executing in parallel. First of all a lot of computational resources and energy are wasted on spinning on high contended locks, and the performance drops due to all coherency traffic. Solutions to this, besides trying to avoid locks as often as possible, can lie in the use of an Ethernet-style exponential back-off algorithm, or by suspending the waiting threads on the lock, waking them up as it becomes available again. The downside of such a suspending lock mechanism is the much larger overhead to wake a thread up again when the lock is released, and that it can not always be used in the lowest level of system services due to its dependence on the scheduler. Such locks could of course not be used in the implementation of the scheduler itself or anything it depends on, unless this suspending and resuming of the threads is supported directly and efficiently by the hardware.

In SVP we have synchronizations based on our create and sync primitives, where we can use create to start a critical section on an exclusive place, and use sync as a barrier synchronization on the threads of a child family. However, we can not use these for synchronizations between arbitrary threads, or in the light of our previous section, between arbitrary processes. Therefore, if we want to implement higher order communications between processes, for example a FIFO channel such as the Unix pipe, we need a way to implement higher order synchronizations. We will explore several approaches in the rest of this section, but first we will more closely look at exclusive places.

4.3.1 Exclusive places

Scheduling and mutual exclusion are directly supported by an SVP implementation. Critical sections can be executed on an exclusive place which automatically takes care of mutual exclusion of multiple actors trying to access it. A create to an exclusive place that is already occupied is buffered and only scheduled when the exclusive place is free again. When multiple create actions are waiting for a single exclusive place, there is no guarantee which will be the next to receive access to the place. This is not defined by the SVP model, and is left up to the implementation which will most likely implement it as a FIFO, as there is little reason to enforce a different ordering. Actually we will show in the next sections that it is important that some fairness is involved in this ordering for certain synchronization mechanisms to work properly.

While exclusive places are ideal for simple critical sections or exclusive accesses, they also provide the only means for two arbitrary unrelated threads to synchronize and share data due to the consistency model (cf. Chapter 3). As with any general synchronization mechanism, this introduces the risk of deadlock into the model that
was designed to be deterministic and free of communication deadlock [CSA44] before we introduced exclusive places. We give an example, illustrated with Figure 4.4;

![Diagram](image)

**Figure 4.4: Deadlock situation with two exclusive places $P_A$ and $P_B$.**

Assume two exclusive places, $P_A$ and $P_B$, and two threads $T_1$ and $T_2$ that execute independently elsewhere in the system. Thread $T_1$ creates a family $F_{1A}$ on $P_A$ which creates another family $F_{1B}$ on $P_B$. The second thread, $T_2$ tries to do the same vice versa; it creates a family $F_{2B}$ on $P_B$ which then tries to create $F_{2A}$ on $P_A$. In a scheduling where both threads first create their first family ($F_{1A}$ and $F_{2B}$) before the second is created, deadlock ensues as $F_{1A}$ can’t create $F_{1B}$ as $P_B$ is already occupied by $F_{2B}$ which will not terminate as it tries to create $F_{2A}$ on $P_A$ before it can complete.

The deadlock issue of this example could be avoided by restricting the use of exclusive places, for example by disallowing threads on exclusive places to create new families. However, this is not enough as we will show in the next sections how to create arbitrary locks or synchronizations using exclusive places, which leaves endless possibilities for deadlock. It should be noted that this is not always an issue, but it constrains the set of SVP programs or components to which we can attribute the determinism and free of communication deadlock properties. Simple algorithmic concurrency patterns for data parallel operations for example will not require the use of such synchronizations, and will still have these properties.

Another side effect of the determinism and communication deadlock free property of SVP was that programs always have a well defined sequential schedule which can be used to sequentialize the leaves of the concurrency tree and therefore adapt the granularity of concurrency. Again, care must be taken here when using a synchronization using exclusive places. Another example with a very basic synchronization between two arbitrary threads;

Assume $P_A$ to be an exclusive place, and two threads $T_1$ and $T_2$ that execute independently somewhere in the system. Thread $T_1$ creates a family $F_{set}$ on $P_A$ which sets a flag. Thread $T_2$ repeatedly creates a family $F_{get}$ on $P_A$ which reads the flag until the flag is set. A problem can arise when the part of the concurrency tree that contains $T_1$ and $T_2$ is entirely sequentialized; if $T_2$ is sequenced before $T_1$ it will go in an infinite loop as the flag will never be set.

This clearly has an important implication on the ability to sequentialize parts of the concurrency tree. It is guaranteed that a sub-tree of families can be safely sequentialized if every family within it only shares state with its parents and children, and no state is shared through exclusive places. This implies that it contains no synchronizations between unrelated threads, and therefore the determinism and deadlock freeness as proven in [CSA44] holds, allowing for safe sequentialization.
4.3. SYNCHRONIZATIONS

Having identified these important limitations that we should be aware of, we can now proceed to define higher level synchronizations based on exclusive places in the following sections. These can be used for our earlier defined processes to synchronize and communicate, or for waiting on completions of requests done to system services. We start off with a spinlock style synchronization based on the use of an exclusive place.

4.3.2 Spinlocks in SVP

Atomic operations such as test and set instructions contradict the consistency model of SVP that we discussed in Chapter 3. While there is no limitation specified on the instruction set of an SVP implementation, the consistency model could not guarantee that a modification in memory made by an atomic operation in one thread would be seen by another thread. Of course an implementation can decide to implement these instructions and specify them as an exception to the consistency rules, but we try to define a general SVP based solution here.

This issue is solved by delegating all instances of an atomic operation on a specific piece of data to the same exclusive place. In fact, special atomic instructions are no longer required for this as the sequence of instructions executed by a thread on the exclusive place are guaranteed to execute atomically in relation to the other instances.

```
1 thread t_lock(shared bool lock_failed, bool* state)
2     lock_failed = *state;
3     if(*state == false)
4         *state = true;
5
6 void lock(spinlock_t* lock)
7     bool lock_failed = true;
8     while(lock_failed)
9         create(fid;lock->Px;0;1;;) t_lock(lock_failed, lock->state)
10        sync(fid);
11
12 thread t_unlock(bool* state)
13     *state = false;
14
15 void unlock(spinlock_t* lock)
16     create(fid;lock->Px;0;1;;) t_unlock(lock->state)
17        sync(fid);
```

Figure 4.5: Pseudo µTC code to implement an SVP style spin lock using an exclusive place

This method allows us to implement a spinlock on an SVP based system using an exclusive place. An example implementation is shown in Figure 4.5 using pseudo µTC. The spinlock_t type is a structure containing a memory address (*state)
where the lock state is kept and a place identifier ($P_x$) for the exclusive place used to guard this lock.

The locking function delegates a single thread family $t\_lock$ to the associated exclusive place of the lock which then inspects the lock state. If the lock was not locked (state set to $false$) it is locked by updating the state to $true$. The result of this operation is propagated back through the shared channel identified by $lock\_failed$. After the completion of $t\_lock$ has synchronized, the lock function inspects $lock\_failed$ and reattempts the locking procedure when it did not manage to acquire it.

The unlocking function is much more simple, as it is just a spin lock without registration of ownership, it delegates the single threaded $t\_unlock$ family to the associated exclusive place which then resets the lock state to unlocked.

Such an SVP based spin lock implementation suffers partially from the same problems as a spin lock in classic systems. A lot of energy and execution resources is potentially wasted on attempting to lock an occupied lock. However, as the update process is delegated to the location of the lock data instead of the data pulled to the update process, it should not suffer as much from the coherency traffic problems. Always delegating the locking procedure to the same place in fact has an advantage in exposing more data and instruction locality.

Multiple of these SVP spin locks can be multiplexed using the same exclusive place, as the clients accessing the locks will only occupy the exclusive place for a short bounded time. Also when a client needs to acquire two locks at the same time it is not a problem when they are mapped to the same exclusive place, which is in contrast to delegating whole critical sections to an exclusive place where this time is not always short or even bounded. Even worse, when executing on multiplexed exclusive places, the situation where two locks are required could lead to an unexpected deadlock.

Another issue with SVP spin locks is the contention on the exclusive place guarding such a lock; fairness on accessing the exclusive place has to be guaranteed, or the delegation of the unlock procedure might suffer from starvation when it is under heavy pressure from other clients trying to lock it. Effectively one could say that the exclusive place then suffers a denial of service attack from the clients trying to acquire it.

### 4.3.3 Suspend and resume with sear threads

In order to have a more resource and energy efficient implementation of locks, we prefer a mechanism that is able to suspend a thread and be able to have an arbitrary thread wake it up again, instead of having the waiting client continuously spin as in the spinlock approach. In this section we will introduce the notion of a sear thread, named after the mechanism in a gun that releases the hammer when the trigger is pulled.

A sear thread is a single thread family that is created by a thread that needs to suspend awaiting some synchronization. The thread $syncs$ on this family and therefore suspends, and the sear thread makes sure it does not terminate until the synchronization takes place without taking up execution resources. So far, we have discovered two possible implementations of sear threads, that we will now discuss. The way to suspend a thread is to perform an unbounded latency operation that is
4.3. SYNCHRONIZATIONS

guaranteed not to complete. This can be either a sync on a family that has been forced into a deadlock, or attempting to read a communication channel that is never written.

Deadlock approach

```plaintext
place store_place;
family stored_sear_fid;

thread sear_suspend(place target_place)
    create(fid;target_place;0;1;;) sear_suspend(target_place);
    sync(fid); // never syncs as it can't be created

thread store_sear_fid(family fid)
    sear_fid = fid;

thread get_sear_fid(shared family fid)
    fid = sear_fid;

void suspend(void)
    create(sfid;sear_place;0;1;;) sear_suspend(sear_place);
    create(fid2;store_place;0;1;;) store_sear_fid(sfid);
    sync(fid2);
    sync(sfid); // suspends here

void resume(void)
    create(fid;store_place;0;1;;) get_sear_fid(sfid);
    sync(fid);
    kill(sfid); // kill sear, wakes up suspend
```

Figure 4.6: Pseudo $\mu$TC code to implement a sear thread mechanism using deadlock and kill

A deadlock based sear thread is created on an exclusive place, and then tries to create another family on the same place, immediately deadlocking itself. The sear thread can be released using the kill action; depending on which type of kill is implemented, either by killing the family of the sear thread or by resetting the exclusive place it resides on. Example pseudo $\mu$TC code is shown in Figure 4.6, which assumes the family based kill.

The suspend function sets up a sear thread sear_suspend that deadlocks itself by attempting another create on the same exclusive sear_place that is passed as an argument. The function then stores the family identifier of the sear thread in stored_sear_fid, but does this with a delegation to the exclusive place store_place to guarantee that this value can be accessed by unrelated threads due to the consistency rules.
The `resume` function then delegates to the `store_place` to retrieve the family identifier of the sear thread that was written by the `suspend`, and then sends a `kill` to that fid. As the first instance of the `sear_suspend` is now killed, the `sync` in `suspend` (line 18) returns and effectively wakes up that context again.

Obviously the same mechanism can be applied with a place based `kill` implementation. Storing the family id is then replaced by storing the exclusive place the sear is delegated to, and then using the `kill` action to reset that place in `resume`. Regardless of the way `kill` is applied, the issue with this approach is that it requires two exclusive places to function. While the storage place can be easily shared to store the information of multiple sear thread synchronizations, the other place is tied up for the time that the sear thread is holding its parent suspended.

**Shared channel approach**

As the deadlock method potentially takes up too many resources, we now present an alternative implementation that uses threads suspending on a shared channel. Multiple implementations are possible, but not all of them follow the generic SVP model that will work across all our mentioned SVP implementations. One of the reasons was that the older \( \mu \)TC language did not support defining a shared channel as uninitialized in order to later write to it in the parent thread after the `create`, making this mechanism unsupported in the software SVP implementations. These issues were later solved in the SL language by making these explicit actions.

On the Microgrid, it is possible to use a sear thread based on a family with a single thread and one shared communication channel as follows. The parent leaves the initial state of the shared channel empty and the sear thread tries to read from it, suspending itself on the read as the channel is not written yet. The parent then saves the family identifier similar to the deadlock based mechanism, and `syncs` on the sear to also suspend. The `resume` function reads this family identifier, and then releases the sear again by writing a value in the shared channel. This is a Microgrid specific solution as it has a special instruction to write a value to a shared channel of a specified family, as long as a valid family identifier is used and works regardless of the fact if the target family is a child of the current thread or not.

Alternatively, in a more generic SVP way, the sear can be released in the same way as the previous example. Using a `kill` on the family identifier (or place) will work. However, as we can’t define such a suspending sear thread in \( \mu \)TC, we present one more alternative. We show the related `sear_suspend` and `suspend` code in Figure 4.7. Instead of creating a sear consisting of one thread, we create a family of two threads with a shared channel. Only the second thread attempts to read and use the value from the channel. The first thread will do nothing and the second thread will block waiting for the first thread to write the channel which never happens. To release them, the same mechanism based on `kill` can be used, and therefore this code is not shown in this example. This mechanism would work under the assumption that a shared channel from a thread is not implicitly written by the SVP implementation when that thread terminates, potentially leaving it with an uninitialized value.
4.3. SYNCHRONIZATIONS

4.3.4 Suspend and resume with continuations

While the previous approaches all relied on heavy use of exclusive places and other resources in the duration a lock is kept, we now discuss an alternative that requires much less resources. It is based on saving a continuation [52] instead of suspending or spinning a thread context. This synchronization using continuations or callback threads can only be used when the synchronizing threads do not rely on maintaining a strict parent-child relation within their process.

When the synchronization point is entered, the first thread that arrives saves a continuation for itself there and terminates. Then, when the second thread reaches the synchronization point, it starts continuation of the first thread by creating it in a new concurrent context, effectively resuming it again. As state needs to be shared between two independent threads, this exchange needs to take place on an exclusive place. However, this is only held for a fixed amount of time to either store or resume the continuation, and therefore the use of this place could be multiplexed for multiple of these synchronizations.

While having such a synchronization within a single process works quite well, it is a bit more complicated to implement between two unrelated processes. First of all, the threads might not know the type signature of the thread they need to start with the continuation, but this can be solved with a wrapper with a well defined interface that is generated with the continuation. Another issue is one of security and process identity, as a process should generally not be allowed to create arbitrary threads in the space of another process. This means that either the credentials to be able to do this should also be generated in the wrapper, or the wrapper would have to contact a system service to have it create the thread at the right place with the right credentials.

Many other synchronization constructs can be built with this mechanism, besides the rendezvous style synchronization that we have just discussed, which was the basic synchronization primitive between processes in Plan 9 [122]. For example, it is trivial to queue up multiple contexts by saving a list of continuations and either releasing them one by one as another actor reaches a certain point (similar to signaling a POSIX
conditional for example), or to release them all at once when a certain number of actors have reached the synchronizer (i.e. implementing a barrier synchronization).

4.3.5 Summary

We have shown how we support critical sections using exclusive places in SVP, but have also highlighted the effect it has on the sequentialization properties of SVP, which can not reliably be applied to code that uses exclusive places. Then, we have shown several approaches of designing synchronizations based on exclusive places, which have different resource trade offs. The latter is important as we have seen in Section 4.2 that we have a bounded set of physical resources as long as we can not virtualize them, and therefore the number of available places or exclusive places might be limited.

First of all a spin lock style approach which has the downside of not being energy efficient, as a lot of pressure is put on accessing the exclusive place. While multiple spin lock synchronizers can be multiplexed on an exclusive place, this will have performance issues as all clients of all synchronizers would continuously try to acquire the exclusive place.

The second implementation we discussed was a set of sear thread implementations that rely on deadlocking or infinitely suspending a thread which is then released. As all threads are suspended this is more energy efficient, but it ties up potentially valuable resources. The deadlock approach uses up an exclusive place for each lock and is therefore even more expensive than the spin lock synchronization. However, the shared channel approach only uses up a family entry when a family based kill can be used or the instruction on the Microgrid to write to the shared channel.

Finally, we presented a third approach using continuations which has the advantage that it is both energy and resource efficient. Only a single exclusive place is required to store the continuation state at, which can be multiplexed between multiple synchronizers. However, this synchronization primitive has limitations on the relations between threads in the processes that synchronize.

While neither of the three synchronization approaches seem very elegant, we have shown it is possible to build arbitrary synchronizations on top of SVP using exclusive places with different resource trade offs. Which synchronization approach is the most suitable depends on the SVP implementation, and the limitations on its resources.
CHAPTER 5

Protection and Security

5.1 Introduction

In the previous chapter we dealt with the definition of a process, process management and process synchronization on SVP based systems. In this chapter we will deal with protection, which covers two other important tasks of an operating system that were part of the definition made in Chapter 2. We discuss how interference between processes can be prevented, and how we deal with address space and memory management.

Let us start with a more precise definition of what kind of interference between processes we want to prevent. We strive to implement a form of isolation which guarantees that a software component can not influence the functional behavior of another software component unless they have a defined relationship that allows this. Therefore, we decide to emphasize our investigation on protecting functional behavior rather than temporal behavior as the latter would require Quality of Service (QoS) mechanisms reserving bandwidth everywhere in the system, such as on the delegation and memory networks. Simply put; we do not want one component to be able to break the functionality of another component by creating or killing threads or by corrupting memory, but it can, for example, hog the memory network that will impact the performance of other components.

We can now distinguish two required mechanisms to protect against this interference. First is to control the access to computational resources, i.e. access to places to be able to start and terminate threads with \texttt{create} and \texttt{kill}. Second is to control the access to memory locations, which is implemented by limiting the access rights of a component, often done by using private virtual address spaces in traditional systems. We will discuss our mechanisms in the next sections, first how we restrict access to our computational resources using \texttt{capabilities} [49], and then we present our approach for memory protection.

It should be made clear that the protection mechanisms in this chapter are currently not implemented in any existing SVP implementation, and that this chapter serves as an exploration of approaches that would suit the model. Only a very basic
implementation of capabilities is currently supported in the Microgrid implementation, which will be discussed. As research on the Microgrid architecture so far had not yet focused on operating systems or simultaneously running multiple applications, it was not a priority to fully implement these security mechanisms as they would complicate and slow down the simulation platform without adding required functionality for the research at the time.

5.2 Protecting Computational Resources

Capabilities are a well known approach to govern the access to resources or services in concurrent and distributed systems [49, 165, 113], as we have seen in our overview of systems in Chapter 2. Each actor in such a system has an associated list of capabilities of which each grants a certain type of access to a specific resource or service.

One issue with capabilities is to prevent malicious programs or users from forging them, attempting to gain illegal access to resources in the system. There are three different ways of dealing with this as identified by Tanenbaum in [141]. A first approach is to have a tagged memory architecture in which memory locations can be tagged so that they can only be modified by special operations, and therefore provide a safe storage space for capabilities. A second approach is to keep all capabilities in kernel space which prevents tampering with them using standard memory protection mechanisms, which was used in Hydra [165]. The third approach is to have them in user-space but to add a cryptographic checksum to prevent forgery, which was the approach used in the Amoeba distributed operating system [113].

![Figure 5.1: Layout of a cryptographically protected capability in Amoeba](image)

Server | Object | Rights | Cryptographic Checksum f(Object, Rights, key)
--- | --- | --- | ---

The nice thing about the third approach is that the capability can be handled by the user program and the service that provided it, without any special mechanisms in the system. The layout of a capability in Amoeba is shown in Figure 5.1. When the service hands out the capability, it generates a cryptographic key and takes the fields of the capability which describe the object and access rights and computes a cryptographic checksum using this key. When the service is accessed, the checksum of the provided capability is checked which guarantees that the fields of the capability are unchanged. The downside of this approach is that on every access the checksum needs to be computed which is unfeasible in a low latency hardware implementation.

5.2.1 Applying capabilities to SVP

In SVP based systems, we want to control the access to execution resources, and to currently executing contexts. This means controlling the access to places and to
families in SVP terminology, as this is the only granularity at which this is managed in the model. Both are accessed through identifiers; a place identifier is used to specify a resource at a create, and a family identifier is set to identify a family after it has been created. These two identifiers give us a straightforward location to implement capabilities in SVP based systems, we only need to prevent them from being forged.

When a process requests a new resource from the SEP resource manager (cf. Chapter 6), this will find and set up an appropriate place. It configures the place to only accept delegation requests that present the proper capability. Then, the place identifier which embeds this capability is returned to the process. When this is used to delegate a computation to that place with create, the capability is automatically checked by the target place. When the capability does not match, the delegation is aborted and will return an error code when the process tries to sync on it. Additionally, it may raise an exception [MWvT8] that notifies the system, for example the SEP, that the process attempted to access the resource. Depending on the situation, the SEP might decide to terminate the process or ignore this action, for example when it just recently revoked the capability of that process to use that place.

Capabilities for family identifiers work in an even more simple way. When a family is allocated on a processing resource, a capability is generated that identifies it. This is combined with an identifier of the processing resource and returned as a family identifier as the create action completes. When a sync is executed on the family, the associated processing resource is contacted which then checks the capability. Again, when this fails it returns an error code at the sync, and might optionally signal the SEP of an illegal action by the program. A family based kill implementation would work similarly with respect to capability checking and handling capability errors.

Choosing which of the three aforementioned implementations of capabilities to use is left up to the SVP implementation as all three could be applied in the use cases we have just described. However, as we would like to be able to pass around family and place identifiers, for example through memory or shared channels, this implies that the first two implementations that rely on secure memory locations would have to use references to these as part of the identifiers instead of embedding the capability itself. As for the cryptographically protected capability, this might have too much impact on performance depending on the granularity of concurrency in the SVP implementation. Computing a checksum taking 1000s of cycles is not acceptable when you are able to create concurrent threads in the order of 10s of cycles. We will now discuss how this has been tackled in the Microgrid architecture, as described in [CSA30].

5.2.2 Capabilities on the Microgrid

The Microgrid avoids the problem of the expensive computation of a cryptographic checksum as there are no fields to protect besides the addressing field. This means that the checksum can be replaced with a key value which is looked up as soon as the address is contacted. The key then only needs to be generated when the capability is set up, and can be done with a simple pseudo-random number algorithm that is much cheaper than computing a cryptographic checksum.

There are no other fields than the addressing field as the Microgrid does not discriminate between different access rights. The possible ways of accessing the resources
is limited in any case, only the actions create, sync and kill are used on families or places. Figure 5.2 shows the layout of the 64-bit place identifier in the Microgrid. It contains an encoding of the location and size in \( A + 1 \) bits where \( A \) is the \( \log_2 \) on the numbers of cores in the Microgrid. As \( N = 63 - A \), this means that for example on a 1024-core Microgrid there are \( 2^{53} \approx 10^{16} \) possible key combinations.

Figure 5.3 shows the layout of the 64-bit family identifier in the Microgrid. Besides the capability key it contains an index into the family table of the core, using \( F \) bits which is the \( \log_2 \) of the family table size, and the address of the core which is size \( A \). This means there are \( N = 64 - (F + A) \) bits available for the capability key, and on a 1024-core Microgrid with 64-entry family tables this corresponds to \( 2^{48} \approx 10^{14} \) possible key combinations.

While this seems to give sufficient control and security over accesses to resources, this is not sufficient for securing any calls to the system as we will discuss in the next section. Furthermore, while the capabilities for families can be automatically generated in a core, the capabilities for a place still have to be set by a system service which requires the use of a privileged interface. And when multiple actors have the capability to access a place, their rights can only be collectively revoked, and not on an individual basis.

### 5.2.3 Special case: System calls

The approach to capabilities that we have just described gives us the ability to specify who is allowed to start new executions where in the system, but it does not control what is being executed there. This is important to constrain, as to keep a separation between user processes and the system services; we can’t allow a process to create an execution of arbitrary code on a system place. While we could employ a memory
5.2. PROTECTING COMPUTATIONAL RESOURCES

A protection mechanism to restrict such executions to only a small trampoline function that then safely jumps to system code, this is still not sufficient. It does not guarantee that the execution starts at the beginning of this function, as any program location (program counter) could be specified at `create`. This means that the execution could start somewhere in the middle of this function, with potentially catastrophic results.

Figure 5.4: Indirection of a `create` of a system service through a vector of registered functions

To safely interact with the system, we require a way where we not only restrict which component can interact with which resource, but also the specific thread function executed on these resources. Our solution here is to have a `syscall` variant of `create` that uses a vector of registered functions that can be started, shown in Figure 5.4, not unlike a vector of interrupt handlers. This `create indirection vector` can not be read or modified by the process directly, but by specifying an index into the vector using the special `create` action, a specific function can be instantiated on its corresponding place. The vector contains the place identifier for each function and therefore also the capabilities to access this place. These do not have to be granted to the process in general, which avoids the problem of arbitrary code execution on system places. In fact, this create indirection vector implements another form of a capability store, where an entry is a capability to invoke a certain service on a certain place.

Where this vector is stored is not an issue here, this can either be in the hardware directly or with a base pointer to a memory address, as long as the contents of the vector can be protected. However, we suggest it should not be a global vector that is used by every component in the system. For example, on large systems certain OS services might be duplicated to avoid bottlenecks, and a component would want to contact the nearest location providing the service. Alternatively, a component might require a specific set of customized services from the system. This means that we want to specify this vector for at least every place separately, and optionally even every individual computing element, which we leave up to the SVP implementation.
5.2.4 Configuration

Capabilities for families are generated as soon as an entry is allocated, but capabilities for places need to be managed and configured. Also the create indirection vector needs to be initialized with the calls that are allowed to be made from that place. As we have discussed before, the way this vector and the capabilities are stored depends on the SVP implementation, and the interface to initialize them is therefore not specified as it would be implementation specific. Whatever these interfaces are, they will have to be used by the SEP resource manager (cf. Chapter 6) to initialize a place when it is allocated, and can be used to revoke the rights of a process in an attempt to reclaim a place. On the Microgrid, configuring the capabilities for a place would happen by accessing a control network and writing special core registers.

Regardless of how the configuration interface is implemented, it is interesting to see how this could be protected and used itself. If this would be protected using capabilities itself, this means that an SEP could delegate the responsibility for configuration of a set of places to another entity in the system by giving it the capability. For example, it could delegate a large set of resources to a private resource manager of a large application. Another use would be to have a virtualized or contained SVP system running on a partition of the resources. Of course the SEP would still require to have a separate capability to still access the place which can not be revoked with the delegated capability, to prevent it from being locked out.

There are still two problems with this approach. The first is a chicken and egg bootstrapping problem regarding capabilities to configure the system. We assume this can be solved by having the SVP implementation initialize all capabilities for the configuration interface to a predefined value when it powers/stops up, and then when booting the system during SEP initialization those capabilities are regenerated and stored by the SEP. The second problem involves the delegation of the responsibility of managing capabilities, but also involves the problem of selectively revoking rights. Both potentially require an unknown number of capabilities to be stored, which we will now investigate.

When delegating responsibility, the question is if this can be done hierarchically. If the responsibility is delegated, can the process responsible for the resources then further delegate the management of a (sub)set of those resources to another entity? As every level in this hierarchy still wants to retain a capability to manage all the resources at its level to prevent being locked out by a lower level manager, this requires a stack of capabilities for accessing the configuration of those resources. Another question here is how common would it be to do such a delegation and sub partitioning of resources, something that future research will have to show, but we will give our initial insights.

If we assume that in the worst case at every level resources are divided in two, this means it does not grow very rapidly as we only have $\log_2(N)$ levels, where $N$ is the number resources that can be managed. For example, a 1024-core Microgrid would only require 10 levels of capabilities per core then. While this number would not grow very large, the question remains if just having two levels would not be enough. Only a new level of capability needs to be entered when resources are delegated to another entity that can not be trusted. For example if the SEP system partitions the
resources itself (for example for locality of SEP access) to a set of subsidiary SEPs, they are implicitly trusted and do not require a separate capability level. Similarly, if a run-time system of an application has acquired a set of resources under its own capability level, it would not be likely that it requires additional levels of capabilities when delegating to sub partitions. In fact, the number of supported capability levels could be seen as a number of different privilege levels in our system.

The second half of the problem is related to being able to selectively revoke rights. When multiple actors are granted access to a resource by means of a capability, one can only revoke the capability and therefore revoke access of all these actors. If only the access of a single actor needs to be revoked, this requires generating a separate capability for each actor so that they can be selectively revoked. However, this also requires a unknown number of capabilities to be supported by a resource which is undesirable as for example checking the capability will no longer be feasible within a fixed amount of time but would depend on the size of the list.

This problem can also be easily solved, as we would never share resources that can be used arbitrarily between two or more independent actors. A resource would either be a system resource that is accessed through the indirect create mechanism that we described before, or it would be cooperatively shared between actors. When using the indirect create mechanism we can selectively revoke rights by modifying the create indirection vector on the places that are occupied by that process. This does not require modifying the capability attached to the resource, so other actors are not affected. Furthermore, this also allows for safely replacing services; when a service in the system is replaced by a new service, all vectors can be updated to the new service and as soon as all current instances of the old service terminated it is guaranteed that it can be safely cleaned up and only the new version is used from that point.

5.2.5 Summary

We have identified how capabilities can be applied in SVP by using them for accessing resources (places) or currently executing code (families). However, to safely access system routines we need the equivalent of a system call which restricts the code that can be executed on a place for which a capability is present. This is done by introducing an indirect create using a vector of thread functions with their corresponding target places and capabilities. The configuration of the access to these resources can also be protected with capabilities itself, which requires multiple levels of capabilities to be able to delegate resource management.

5.3 Memory Protection & Management

5.3.1 Concurrency granularity of protection

We consider the discussion of at what size-granularity protection should be managed (viz. memory words, cachelines, memory pages, etc) part of future work which should be evaluated with implementations and use case scenarios, that heavily depend on the specific implementation that supports SVP. However, we need to decide a strategy for
applying memory protection in our model and at which granularity of concurrency. What do we want to protect from what? Threads? Families? Places? Components? Processes? And does this depend on the granularity of concurrency in the SVP implementation? We will now discuss our analysis on this.

In fine grained SVP implementations it is obvious that handling protection at the thread level causes too much overhead, so at least for implementations such as the Microgrid this is not a good option. Switching protection domains while trying to switch thread contexts potentially every other cycle would for example require a separate translation lookaside buffer for every thread table entry. Also setting up new protection contexts when creating a new family of threads would make the low overhead creation of threads much more expensive. Furthermore, a family of threads indicates a close relationship, often expressing locality and concurrency in a computation, often operating on individual parts of a shared dataset. Therefore it also does not make sense to handle protection for each thread individually.

The next level at which we can manage protection is at the level of creating families. When a thread creates a new family, it can enforce protection by drawing a line in the sand so that everything running below that runs in another protection domain [96]. This is the strategy that we presented in [MWvT11], but this does not mean that we would want to enter new levels of protection on every possible family creation.

While protection was never implemented in any SVP implementation so far, an implementation for memory protection on the Microgrid was proposed in the course of the Apple-CORE project [CSA31]. It would support separate protection domains for every family in the system, and at any arbitrary create, a component could decide to have its children either share its protection domain or to run in a new protection domain. Again this gives an overhead for each family, and the question remains if we need to be able to define a new protection domain for each individual family creation. My answer to this question is "no"; it is sufficient to manage protection at a create only when crossing place boundaries, and we will now discuss why.

Applying protection domains implies trust relationships; the reason to start a child family in a new protection domain would be that it is not trusted. If we run this child on the same place, this means that it has access to exactly the same resources, except memory. A non trusted child could for example still exercise the capabilities attributed to that place or use up all of its execution resources. This is something we want to avoid, and therefore a non trusted child should be delegated to a separate place. Therefore, there seems to be no use case for exercising protection on arbitrary family creations, but only when delegating families to different places.

When protection is only enforced on family delegations, managing protection at the level of places is the most sensible option. No protection information needs to be kept on a per family basis, but can be registered in the place, similar to the capabilities discussed before. If we would want to manage protection at a coarser level, for example on applications, this would again require support for families of different protection domains on a single resource which sounds counter intuitive. However, this would be needed in order for two applications to communicate by sharing an exclusive place, for example using the locking mechanisms from Chapter 4. With place based protection, this shared place can be set up as a separate protection domain which
only has access to the data required from both applications, and both applications
delegate their functions into it. We explore this further in Section 5.3.4 by discussing
several use-cases.

As we manage our memory protection on the level of places, this does not prevent
us from having multiple places that are all configured to be in the same protection
domain. Alternatively, as mentioned before, two places can be configured to both
have access to a shared part of memory. How large these parts are depend on the
granularity of the protection management in terms of address space, which we will
now discuss in the next section.

5.3.2 Address space granularity

It is difficult to say what a good granularity for address space and protection man-
agement would be for SVP systems without evaluating it with an implementation
and use cases. Furthermore, it can also largely depend on the SVP implementation
itself, for example, it is our current opinion that it is likely too expensive in terms
of transistor budget to have a memory translation unit on every single core on the
Microgrid. In this section we will discuss several options and what their trade-offs
are.

Single or multiple address spaces?

The first thing to consider is the number of address spaces; should we have a single
large shared address space, or multiple private address spaces? While private address
spaces intuitively give a rigid boundary for the protection of data, they do require ad-
dress translation before the shared memory network is accessed. A large shared single
address space has the advantage that memory protection can be fully decoupled from
memory translation [89], and sharing data between different protection domains is
much more convenient as memory addresses have the same meaning in every context.

Single address space systems have been under investigation since the dawn of 64-bit
microprocessors in the 1990s, as they require a wide address space to be implemented
comfortably [89, 35, 75]. On 32-bit systems, having multiple address spaces using
virtual memory [48] is a way to overcome the scarce number of addresses available to
programs, which is the approach taken by many contemporary systems.

The major advantage of a single address space approach is the support for efficient
zero-copy inter-process communication, the data that needs to be communicated only
needs to be mapped into the protection domain of the receiving process allowing read
access. While this can in theory also be done in private address spaces, the practical
problem there is to map the data on the same address range in the receiver which
might not always be possible. This lack of referential transparency means that pointer
based data structures can not be shared without resorting to pointer translation or the
use relative addressing, while they can be shared transparently with any protection
domain in a single address space system.

A second advantage of the single address space approach is that protection and
address translation can more easily be decoupled. While protection checks still need to
happen on every single memory access of each core, the cache hierarchy can (partially)
be virtually addressed and the translation can happen at a coarser level further down the path to the backing store. In contrast to translation, protection can be checked concurrently with a memory access [89], which can contribute to a lower access latency to the first level of memory.

A problem with address translation at every core is to keep the translation mechanism consistent when changes to the translation rules are made. Mechanisms such as TLB (Translation look-aside buffer) shutdown [18] can be used to invalidate stale TLB entries, but need to be constructed carefully in order to scale well [13]. In a single address space approach, a similar mechanism would be required, but only to propagate changes in protection.

One of the disadvantages of a single address space approach was that the Unix fork call which makes a copy of a process and its address space could not be implemented, as no two copies of the memory of the process can be made when the newly forked process needs to see it at exactly the same addresses. A solution was provided in the implementation of the Mungi Operating System [75], where they required that all programs are relocatable and therefore use relative or offset addressing using a base pointer. It can then use the same optimizations such as lazy copy-on-write as traditional systems; a second mapping using new virtual addresses is made to the same physical addresses for the copy, and only when a part is written, a physical copy is made.

A second challenge with single address space approaches is the naming and binding of objects [126], as process specific data can no longer be stored at well known addresses as it has no private address space. There are several ways to achieve this, for example using a register which points to a process specific area of memory and where the well known addresses are replaced by offsets into this area. However, this requires an additional indirection or offset calculation on every access, or a different approach altogether to achieve linkage in programs than the traditional C/C++ or UNIX approach.

Another potential problem of a single address space approach could be address space fragmentation, though it is doubtful that this would soon become a problem with a 64-bit address space. For example, the address space could be split between programs similar to a 32-bit private address spaces by partitioning it using the higher order bits to address each partition, providing 4 billion contexts this way and no worse fragmentation problems than traditional 32-bit private address space based systems.

Cross domain calls in SVP

Another interesting way to look at a shared single address space or private address spaces is how they would affect use from the perspective of SVP programs. This is encountered when we want to make a cross domain call in SVP, which means a create of a family to a different place which is configured to a different protection domain. But how do we pass data between the protection domains that does not fit in the limited number of scalar parameters that we can pass through shared and global channels?

In a single address space approach this is simple; the sending side can grant the receiving place read access to the data it wants to send along with the call. Then
5.3. MEMORY PROTECTION & MANAGEMENT

it makes the call to the receiving place, which then consumes the data that it can directly access, or makes a private working copy. This can be done explicitly by asking the system to set up permissions just before the call is made.

In a multiple private address space approach this is more complicated, due to the problems with referential transparency. However, we can resort to our distributed memory approach for SVP described in Chapter 3 and use our data description functions to identify which data needs to be copied on a cross domain call. Furthermore, as this describes the structure of the data, it can be used to make a copy including rewriting of pointers.

However, there are some further issues that also concern the memory consistency guaranteed by the SVP model. We continue to explore this further in Section 5.3.4 later on.

Granularity of protection

Depending on the approach with address spaces, the granularity of protection and address translation can be decoupled or not. As translation in a single address space approach can happen elsewhere, this also means it can happen at a different granularity. For usability, the granularity of protection is of greater importance than the granularity of translation. The choice for the granularity of translation will be dictated by issues such as performance and fragmentation of a specific implementation. Again, as we can not make a substantiated choice yet, this means that we can only discuss what kind of mechanisms we would see fit. The parameters of these mechanisms should be determined during an evaluation of an implementation in a real SVP based system such as the Microgrid, which is considered future work.

However, we do believe that in the case of shared address spaces, SVP programs would benefit from fine grained memory protection as described in the previous section on cross domain calls. An approach such as Mondriaan Memory Protection [163] could be interesting, as it provides fine-grained protection down to the individual memory word level and can be implemented without adding a great amount of overhead. This would be our preferred candidate to investigate for future implementation in the Microgrid.

5.3.3 Managing protection

We have now discussed at what level of granularity in terms of places, families and threads we want to manage protection, and we have discussed the advantages and disadvantages of single or multiple address spaces. Regardless of the address space organization, we can now discuss how and what we manage in terms of protection.

As we have argued that protection should be managed at the level of places, this is something that needs to be set up whenever a place is allocated, similar to how capabilities need to be configured as discussed in Section 5.2.4. This means that handling protection is part of the task that the SEP needs to take care of when setting up a place. We propose that this is handled by a separate service, named the Protection Domain Manager (PDM) which is contacted by the SEP for this task on resource allocation. We prefer to separate the PDM and the SEP services, to allow
programs to access them independently. For example, when a user program would want to change the access rights for another application to a piece of its memory in order to share data, it will contact the PDM to add this region to the other program’s protection domain.

The protection domains that are handled by the PDM describe what kind of accesses from that domain are allowed to parts of the address space. There are several ways how this information can be structured and laid out [96], but the chosen approach will depend on how protection is implemented as part of an SVP implementation. For example, a protection domain can consist of a page directory and page table structure which defines the access rights to each memory page. At this point it is not important for our story what the organization exactly is, but we need to make some assumptions on what it provides. Needless to say, the protection domain information should not be write-accessible by the process it describes.

We assume we can select any combination (including none) of four rights on memory regions per protection domain: read rights, write rights, execute rights, and ownership. While the read, write and execute rights speak for themselves, we now explain the semantics of ownership in our design. Every piece of memory has one single protection domain that is the owner of this memory. Having ownership over a region of memory means being responsible for it; only the owner can ask the PDM to configure this region of memory for another protection domain with one or more of the other three rights set.

Ownership can be transferred by the PDM, but is an exclusive property so that only a single protection domain can be the owner of a certain location at any given time. The transfer of ownership should be an action that is also initiated from the receiving side to prevent malicious programs to allocate resources and then move the ownership to some unaware and innocent process.

The ownership property gives us a handle to reclaim resources that are not explicitly released. As soon as the last place belonging to a protection domain is released (or forcibly cleaned up), all memory owned by it can be cleaned up and released as well. This means that all the other protection domains that received rights to this memory also need to be modified before the region is reused, which requires the PDM to keep a shadow bookkeeping of all rights handed out for a specific region so that it can backtrack this usage without having to scan all possible protection domains in the system.

5.3.4 Data communication and consistency

We will now show and discuss some examples of how data can be communicated between two places in different protection domains, which gives us some examples on how the protection mechanism is used. We assume that the capabilities discussed in the first half of this chapter to start executions are already set up, and only consider the issues with memory protection and the consistency model of SVP. The latter appears to provide quite a challenge, which we will discuss in detail for every example.

In every example we have two processes running on their own places $P_A$ and $P_B$, in their own protection domain, and we will refer to them using the same names. Their internal structure is not relevant, and is therefore not shown in the diagrams.
The dashed boxes around the places denote their protection domains, and illustrate who are located in the same domain. In the first two examples we also explicitly list the access rights bound to the protection domains. We also show the calls that the processes make to the PDM service to modify their protection domains.

Both processes have their own private memories, M_1 and M_3 respectively, and the gray shadows indicate which parts of memory are mapped to the corresponding protection domain of the place(s). In every case we discuss, the process on P_A wants to send some data to the process on P_B.

Zero-copy

The first approach we consider is a zero-copy approach shown in Figure 5.5; the data that P_A wants to send is in memory area M_2. It sends a request to the PDM, the Protection Domain Manager, to add the read permission for M_2 to the protection domain that encompasses P_B. The dashed rounded boxes indicate the protection domains around P_A and P_B, and list their permissions where the permission for M_2 is just added to PD(P_B).

After the permission for M_2 has been set, P_A invokes create on a function using this data on P_B, effectively invoking a Remote Procedure Call (RPC). This function can include making an accept call for an ownership transfer of M_2 through the PDM from P_A. While this function will execute on place P_B and in the context of its protection domain, allowing access to M_2, there are some issues with this approach. First of all, the function will have no parent-child relationship with any other part of the process executing on P_B. Therefore, according to the loose consistency model of SVP, there is no guarantee that it can safely access anything in M_3. In fact, if the function itself is located in M_3 when it is called, there is not even a guarantee that it can be executed, as the consistency view of P_A might not include the function being loaded there.
Zero-copy with exclusive place

To solve the issues with the parent-child relationship of our first example, we present a second zero-copy approach in Figure 5.6. This adds an exclusive place $P_{Bx}$ to the same protection domain of $P_B$. Then, using this exclusive place, a synchronization is made between the executions in $P_A$ and $P_B$, based on one of the synchronization primitives that we described in Chapter 4. This could for example be a rendezvous synchronization where a pointer to the relevant data in $M_2$ is exchanged between $P_A$ and $P_B$. Then, after the synchronization returned, new computations can be started in $P_B$ consuming the data.

While this seems to solve the parent-child problem, it has interesting implications for consistency. In order for $M_2$ to be consistent for $P_B$ after the synchronization, it means that the `create` from $P_A$ to $P_{Bx}$ followed by the `create` from $P_B$ to $P_{Bx}$ needs to 'magically' make $M_2$ consistent for $P_B$ without the data being touched. It is unclear how the underlying SVP implementation should be made aware of this, unless it is aware of the view and permissions that any place in the system has on memory. Alternatively, our suggested annotations with data description functions could be used to indicate which areas of memory need to be made consistent.

Of course, such consistency issues are always present with a zero-copy approach. Alternatively, we can decide to copy the data, which we also have to do in cases where zero-copy is not feasible. For example, in situations where $M_2$ is unreachable by $P_B$ in a distributed memory environment.

Copy through dedicated place

In this example we try to solve the consistency issues by explicitly making a copy of the data. The mechanism is shown in Figure 5.7; we introduce two private memory space $M_{2a}$ and $M_{2b}$ belonging to $P_A$ and $P_B$ respectively, where $M_{2a}$ contains the data to be sent, and $M_{2b}$ is the area to receive the data in. A third place $P_C$ which lives in its own protection domain together with an exclusive place $P_{Cx}$, and $P_A$ and $P_B$ both ask the PDM to grant $P_C$ access to $M_{2a}$ and $M_{2b}$. Please note that we
have omitted the lists of permissions per protection domain for compactness of the diagram.

![Figure 5.7: IPC example using a data copying service on P_C](image)

The communication mechanism then works as follows, P_A delegates a request using create to P_C to send area M_{2a}, while P_B delegates a request to P_C to receive into area M_{2b}. Both requests are then suspended using one of the earlier discussed suspend or synchronization mechanisms, using the exclusive place P_{Cx}. When both requests have arrived, a copy operation is executed on P_C to copy the data from M_{2a} to M_{2b}. After it completes, the two requests are released again and P_A and P_B can unregister the regions for P_C again, and P_B can start to use the data in M_{2b}.

Effectively, P_C implements a memory copy service, an approach we have also proposed and explored [MWvT10, MWvT1] in our work on implementing SVP in software on the experimental 48-core Intel SCC processor. There, we used dedicated cores that ran a copy service to parallelize communications between two other cores by offloading the copy operation. While we did this as a part of our SVP implementation and not in SVP itself, to implement a communication mechanism controlled by data description functions, the mechanism is very similar. That implementation did not implement memory protection, and provided a stronger consistency than required.

As for the mechanism presented here, the consistency is again an issue, similar to the two examples presented before. As the question remains how P_C is guaranteed to read the correct data from M_{2a}, and how is P_B guaranteed to read the correct data from M_{2b} after P_C has written it? Again, similar to the other examples, non trivial relations of delegations to P_C and/or P_{Cx} have to make clear to the SVP implementation which areas to make consistent.

Copy through private memory of dedicated place

Finally, we present our fourth approach to communicate data in Figure 5.8. In this approach P_A delegates a function to the exclusive place P_{Cx} that copies the data from M_{2a} to the private memory of P_{Cx}, that we named M_{Priv}. Then, after this
completes which might require an out of band synchronization elsewhere for efficiency, \( P_B \) delegates a function to \( P_{Cx} \) that copies the data from the private memory space \( M_{Priv} \) to \( M_{2b} \).

![Diagram of IPC example using double-copying in a data copying service]

While it is all but efficient to copy the data twice, it is the only approach that is guaranteed to work with the original definition of weak consistency in SVP systems, without requiring the implementation to ’magically’ understand what is going on. The first copy operation is guaranteed to succeed on the right data as it is a child from the process running on \( P_A \), and the second copy operation is guaranteed to read the correct data from \( M_{Priv} \) as it was just written by the previous occupant of the exclusive place \( P_{Cx} \). Then, after this completes the copy to \( M_{2b} \), further computations in \( P_B \) are guaranteed to see the correct data as the copy operation was a child of \( P_B \).

**Note on consistency**

As we have seen in the four examples that we have just discussed, the current definition of consistency in the SVP model is a severe problem. In the ideal situation, we would like to use the first and second approach, depending on if a parent-child relationship is required, however we have shown that this has issues with the current consistency model, or any potential implementation of it.

A solution could be to define consistency domains, and forcing consistency between two domains whenever a delegation is made from one to another. For example, \( P_A \) and \( P_B \) in the previous examples could be separate consistency domains, or they could coincide with the protection domains. However, investigating new consistency models for SVP is beyond the scope of what we attempt to achieve in this thesis. One suggested approach for a formal consistency model has been described in [CSA38]. For the remainder of the thesis we ignore these issues and assume that these are solved, and that there is a reliable and efficient way to construct inter-process communication.

It should be noted that the consistency issues that we described here are not an issue on the Microgrid, as the first and second approach would work just fine. The reason this is the case is that it implements a much stronger consistency model than
what SVP dictates; it performs a memory barrier before a create is done, and at the completion of a family before the sync is released. This means that after that all writes are visible in the entire system, as the writes have been acknowledged by the distributed cache network.

5.3.5 Memory management

While we have extensively discussed how memory protection can be managed, we have not yet considered where this memory comes from. The way memory and address space management is handled also depends on the address space approach discussion we had before. For example, in a single address space approach, addresses could be partitioned statically over a set of predefined process contexts, and in a multiple-address approach each process will have its own unique address space and can manage the addresses it uses itself. However, physical memory still needs to be allocated to these addresses as needed.

To manage memory allocations we could design a system service that effectively implements malloc(), but having this as a single service would quickly become a bottleneck when many applications make a lot of small allocations and de-allocations. This is why this is not managed by the system in contemporary operating systems, but it is part of the run-time implementations that extend the memory space when needed through system calls. We envision a similar solution; to have applications or their run-times manage their own memory for fine grained allocations and de-allocations, and have coarser grained allocations handled by a Memory Allocation Manager (MAM) service. Such run-time managed fine grained memory allocations have been implemented for the SaC language run-time on the Microgrid [CSA15].

While this provides a solution for heap-style memory management, the threads in an SVP system will also require Thread Local Storage (TLS). Especially on fine-grained SVP implementations such as the Microgrid it is clearly infeasible to allocate and deallocate stacks for threads as soon as every thread starts or when a create is done for a new family. Several approaches have been suggested and analyzed in [CSA38], and the current Microgrid implementation uses a static address space partitioning. As a more general solution for SVP systems we propose that TLS is bound to a specific resource, where the TLS is allocated and configured as soon as the resource is requested. This means that the MAM is contacted during the SEP (cf. Chapter 6) configuration step of a resource when it is allocated. An approach specifically for the Microgrid here can be to specify the required TLS size per thread context when allocating a place, where this is configured inside the cores and where the TLS address of a certain thread can be calculated using a TLS base address stored in the core and the identifier of the hardware thread context.

The implementation of the MAM could be split into multiple layers of services with corresponding granularity, where the granularity increases when going up the hierarchy. Such hierarchical decompositions can help solve problems of contention at the top level memory manager. The MAM has to work closely with the Protection Domain Manager (PDM), as to set up correct permissions on newly allocated areas, and to remove permissions from areas that have been released. It also needs to work together with the SEP resource manager, as when the last resources (place) of a
process are released and therefore the process is cleaned up, the MAM (as well as PDM) need to be contacted so that all memory that was associated with the process can be released.

5.3.6 Summary

We have discussed the different approaches to address space management, single address space vs multiple private address spaces, and have come to the conclusion that choosing one depends on the trade-offs for a specific SVP implementation and needs to be investigated. However, independently of how address spaces will be managed, we presented a strategy for managing protection in SVP based systems. We define protection domains which have access rights to areas of memory, and each place in the system is a member of a single protection domain. As a memory area has a single protection domain as owner, we know it can be cleaned up when the last place leaves the protection domain.

We introduced the concept of services to manage memory protection and memory allocation. The Protection Domain Manager, PDM, manages the protection and ownership information attached to protection domains, and can be used by processes to give other processes access to memory or to change ownership. The Memory Allocation Manager, MAM, manages the coarse-grain allocation of physical memory to addresses, while applications are responsible for fine-grained memory management. PDM, MAM and the SEP Resource manager work closely together, to be able to reclaim resources when processes are cleaned up.

Finally, we have shown how these mechanisms can be used to implement arbitrary sized communications between two processes. The most important conclusion was that the original consistency model of SVP causes a lot of problems; either in using it or in implementing it. We proposed a place or protection domain based consistency alternative that might be too strong, but which would be sufficient to make inter-process communication much more efficient. Further investigations for a consistency model of SVP have to be made, but are outside the scope of this thesis. However, the communication mechanisms that we presented can be used as relevant use-cases for such research.
6.1 Introduction

In this chapter we will discuss dynamic resource management, and therefore the implementation of the SEP resource manager that we have mentioned several times before in previous chapters. In terms of our list of required operating system abstractions from Chapter 2, this service will provide the basis for scheduling and allocation of execution resources to processes, enabling our space-sharing approach that we discussed in Chapter 4. This requires that programs, or their language run-times, do explicit resource management, (explicit in respect to the system), by requesting and releasing resources as they need them, similar to how they dynamically allocate and release memory. The SEP, System Environment Place, is a service that sets up, allocates and releases places as requested, for which we have investigated two approaches.

The first approach focuses on homogeneous architectures such as the Microgrid, where we have many interchangeable general-purpose computing resources in the form of cores. The strategy of this approach is to allocate cores similar to memory; a process requests a certain amount of computational resources and the SEP tries to satisfy this request as good as it can. We describe this in Section 6.2 and refer to it as Resource-oriented Resource Management.

The second approach focuses on heterogeneous architectures which have a specific set of limited functionality. This was used as a basis in the ÆTHER platform which consisted of a collaborative network of nodes that share resources based on the functions they can perform. Here, a process requests a specific function and the SEP finds a node in the system that can perform it. This approach is discussed in Section 6.3 and we refer to it as Service-oriented Resource Management, to distinguish it from the approach where general purpose resources are requested.

For these two approaches we present two corresponding implementations based on our svp-ptl emulation platform. We use these in experiments to show how they behave with applications that dynamically request and release resources. These experiments are geared to demonstrate the mechanisms to manage resources, and do not exhaustively explore the possible resource management policies. We do not focus
on policies in this chapter as they will largely depend on the final design of the under-
lying platform and in which context the system is being used. For example, a group 
server used by multiple students will likely require a different resource management 
policy than a personal embedded device.

6.2 Resource-oriented Resource Management

Our first approach to resource management is based on explicitly requesting a number 
of resources, which assumes a relatively homogeneous system where resources can be 
used interchangeably. Similar to a malloc() memory allocation call, we introduce 
the SEP resource manager as a service that can be contacted to allocate a number 
of cores. As explained in Chapter 4, calling the SEP service consists of a create 
action of sep.alloc to the SEP place to start an instance of the service. After 
the service finished the resource allocation, the instance terminates and the result is 
returned at the sync to the calling process. To satisfy a request, the SEP configures 
one or more resources into a place, sets up the appropriate access controls using 
the protection mechanisms discussed in Chapter 5, configures the memory protection 
domain using the PDM service, and finally returns the corresponding place identifier 
to the component that made the request. When the application no longer requires 
the resources of a place it similarly calls sep.free.

The SEP holds several pieces of information for internal bookkeeping and account-
ing. This at least contains a pool of free resources and an accounting of processes 
and the places they have been allocated. This accounting allows for policies that 
can prevent a single process from monopolizing all resources in the system so that 
a certain level of fairness is achieved. An example policy could be to disallow any 
process to allocate more than half of the available resource in the system, or to assign 
it progressively fewer resources on consecutive allocations.

How the internal structures of the SEP are organized also depends on the proper-
ties of the underlying SVP platform. One of these properties is how resources can be 
combined into places, or if they are static. For example, there is a difference between 
cores that can be grouped arbitrarily, grouping that is limited by the topology; i.e. 
only neighboring cores can be grouped, and fixed or pre-configured groups of cores. 
The SEP should be aware of the platform specific interfaces to configure the resources 
into places, which might include configuring (on chip) networks or simply generating 
and writing the place access key to the resource. This is another reason that an SEP 
implementation is specific for a specific SVP platform. On early designs of the Micro-
grid each place was a pre-configured group of cores, which was later replaced by an 
addressing mechanism that allowed a string of cores to be split into places with a size 
in a power of two. The organization and grouping of resources affects the allocation 
strategy and policy that should be used by the SEP.

It should be noted that, unlike with memory allocations, returning an allocation 
request with more or fewer resources than asked for, even none, is not necessarily a 
problem. In well behaved programs (cf. exclusive place discussion in Chapter 4) we 
are guaranteed to be able to run any families of threads on any sized places, ultimately 
reverting to sequential execution. Of course this will impact performance, so this is
undesirable in (soft) real-time scenarios where a certain level of performance needs to be guaranteed.

When all resources have been allocated, it depends on the underlying SVP platform what the SEP can do. If it can not support pre-emption such as the current implementation of the Microgrid (cf. Chapter 4), then there is no way to gracefully revoke resources that have been assigned to a process and it will have to decline further allocation requests. However, in co-operation with higher level language run-time systems it could implement an upcall mechanism to request the run-time to release some resources, similar to the user level paging mechanism in Nemesis [70]. When an application misbehaves, the SEP can use `kill` to terminate it and reclaim its resources.

### 6.2.1 Scalability

With a service such as the SEP, it is clear that a centralized approach can suffer from contention and we require a scalable solution. Contention on the SEP can become a problem when we have a lot of dynamically evolving concurrency in the processes we run, and a lot of allocation and deallocation requests for resources are made. A solution to this is to have a hierarchical decomposition of resources, similar to what we discussed for the Memory Allocation Manager, MAM (cf. Chapter 5). This has both the advantage of locality for computations and communications within a process when it receives its resources from a local pool, as well as exposing locality for accesses to the SEP itself. We describe two approaches how a hierarchical decomposition can be organized.

In the first approach, the high level SEP conservatively delegates more resources to the lower level SEPs when these require more than they can satisfy locally. This is what we also described for the MAM, where the granularity of resources that are allocated increases with the higher level resource managers, i.e. the higher level SEP adds large chunks of cores to the lower level SEP. However, this approach has a similar resource revocation problem as in a single SEP, as it is unclear when the lower level SEP should return a chunk of resources again to the higher level. This could use a similar upcall mechanism where the higher level SEP asks the lower level SEP if it can return some resources.

In the second approach, all resources are partitioned between the lower level SEPs and a higher level SEP acts as a mediator between them. This means that when an SEP can not satisfy a resource request locally within its own domain, it can ask the higher level SEP if there are resources available elsewhere in the system. We will highlight this more with the example discussed below, and this is similar to the service-oriented approach that we will discuss in the second half of this chapter. One obvious issue with this approach is when the allocation of resources gets fragmented across many partitions. First of all this loses the locality advantages, but also most allocation and deallocation in the whole system would have to go through the top level SEP which then becomes the bottleneck that we attempted to avoid.

Determining which of these two approaches is the most suitable requires an extensive evaluation. It will most certainly depend on the underlying SVP implementation and the behavior of the applications. In terms of evaluating this for the Microgrid,
this requires a simulation platform which can support a high number of cores and a
detailed model of all communication networks, as well as sufficiently large applica-
tions with dynamic resource demands. As this was not available, we only evaluate
a non hierarchical version of the SEP with a highly dynamic application in Section
6.2.3 using our svp-ptl software platform to emulate a Microgrid. However, we will
first discuss two examples of a single and double level SEP to give an impression how
these mechanisms work.

6.2.2 Examples

Using a few illustrations, we will now walk step by step through two examples of
how resources can be allocated using an SEP in an SVP system with homogeneous
resources. We first show a single level example where resources are allocated locally,
and then we show an example of the hierarchical allocation where resources are found
elsewhere.

Single level SEP

In Figure 6.1 we show a partition of this system of 16 general purpose cores on a mesh
network, where the SEP place to manage this partition is assigned to the top left core.
A group of 6 cores is configured and allocated as place P_A, and another group of 4
cores is allocated as place P_B. In this example the resources are homogeneous general
purpose cores, and we assume that the cores are on a reconfigurable mesh that allows
adjacent cores to be grouped into places.

Other services such as the MAM and PDM, as well as the interactions with them
are not shown in these diagrams, and we assume they reside elsewhere in the system.
Also, the number of cores shown is chosen arbitrarily just to serve this example, as a
partition of 16 cores is rather small for one SEP.

A program executing on P_A requires more resources to delegate additional work
to, and makes a request for 4 additional cores by a create to start an instance of
the SEP (Figure 6.1a). The SEP checks its resource map and finds four free cores
in its partition, shown in Figure 6.1b, and it verifies that the resource usage of the
process that P_A is part of and the resource management policies allow the allocation
of additional resources. Then, the SEP configures the cores into place P_C, and sets
up the required information for capabilities and security (cf. Chapter 5). Then, it
returns the generated place identifier which includes the capability to access P_C to
the program on P_A (Figure 6.1c). The program on P_A can now delegate additional
work to P_C using the create action (Figure 6.1d).

Hierarchical SEP

In our second example, we show how a hierarchical structure of SEPs can help to
locate resources elsewhere, using the second approach that we described in Section
6.2.1. First, we continue in the situation where our previous example has left us, and
now the program on P_B tries to acquire 10 additional cores (Figure 6.2). Obviously,
the local SEP can not satisfy this request, as there are not enough free cores available
in this partition. It could allocate and return the only one remaining core, but its
6.2. RESOURCE-ORIENTED RESOURCE MANAGEMENT

Figure 6.1: Steps of the SEP resource allocation example showing 16 cores and the allocation procedure of place $P_C$ by a program on $P_A$ using the SEP.

policies are set to first investigate if the requested amount of resources is available elsewhere in the system.

The SEP will then create an instance of the higher level SEP, which is aware of which other partitions exist nearby in the system. This SEP will then concurrently do a create to each of those partitions to start an instance of their SEP to see if they can satisfy the resource request in their partition. These two steps are shown together in Figure 6.3, where we have the original partition on the top left as well as three more 16-core partitions, the state of their locally allocated places and the location of
CHAPTER 6. RESOURCE MANAGEMENT

Figure 6.2: Hierarchical SEP resource allocation example, first step; not enough resources available locally

their SEP places. The actions performed by the lower level SEPs are similar to those from Figure 6.1 in the previous example.

One of these SEPs has the whole 10 cores available, and the other SEPs do not have sufficient cores available to satisfy the request. This information is reported back to the higher level SEP while the SEP that has the resources available keeps them temporarily reserved. The higher level SEP will then select one of these responses and notify the other SEPs that the request is already satisfied, or if the request could not be satisfied, it makes another request for the lower level SEPs to report what their closest match for the request would be. In our example case it is satisfied and the SEP that offered these resources configures the cores into place $P_X$ which is returned through the SEP hierarchy back to the requesting program on $P_B$ (Figure 6.4). After the configured place has been returned to the program, it can start delegating work to $P_X$, similarly as discussed in the previous example and shown in Figure 6.1d.

6.2.3 Evaluation

To evaluate our approach with resource-oriented resource allocation, we have implemented a single level SEP based on a set of statically configured places in $\mu$TC. We use this in combination with a modeled application that exposes a highly dynamic and unpredictable use of resources for a dynamic resource allocation experiment on an emulation which mimics a Microgrid using the svp-plt run-time system. The modeled application in this experiment is run using a proof of concept implementation of an S-Net [CSA14] run-time system, but the details of this are not important for the experiment here and we defer the discussion of its structure to Chapter 8.

The application consists of a simple streaming application that is shown in Figure 6.5, which could for example be part of a multimedia playback application. A stream of data records comes in and is first processed by box A, and its output data then goes through either box B or box C. For example, A demultiplexes an audio/video stream, and produces either a video or audio record, and B and C respectively do the audio or video decoding. We make the behavior of the application dynamic by having a non deterministic processing time for B and C for each piece of data, and by the
non deterministic choice between B and C as the target for data that is produced by A.

The execution of box A, B and C is independent for each record they process, therefore multiple instances can be executing in parallel on different records. The run-time system keeps track of queues in front of each box and monitors these at regular iterations to make resource allocation decisions, requesting more resources and starting more parallel box instances. In the experiments this Observer component used the following algorithm for these adaptations;

1. Determine the largest queue and the fastest growing queue (or which decreased the least)

2. If any queue grew, take the fastest growing queue
   - If this is currently the largest queue, request a large number of cores
   - Otherwise, request a medium number of cores

3. If none of the queues grew
• Select a queue that is not expected to empty in the next 4 intervals, with preference for the largest queue, request a small number of cores

For the experiments we ran we determined that using the values 16 cores, 8 cores and 2 cores for large, medium and small allocations delivered the best result. However, this greatly depends on the workload of each component. A resource executing a box is released again when there are no more input records to be processed for this box. As this might release resources too soon, a delay can be set to control how long it will wait for an input record before it gives up and releases the resource. We show the effect of this parameter in our experiments.

We assume that the boxes have some internal concurrency that can be exploited when they are assigned to a place of multiple cores. To calculate the execution time for a box instance we use the following equation:

\[ T = R \cdot \frac{W_p}{N} + W_s \]

In this equation, \( T \) is the resulting execution time, \( W_p \) is the parallelizable part of the workload, \( N \) the number of cores, \( W_s \) is the serial part of the workload, and \( R \) is the
random factor that is used for $Box_B$ and $Box_C$, which is always set to 1 for $Box_A$. We only apply the random factor to the parallel part to model the varying computation required on the data, and assume that the sequential part is a preparation with a static amount of computation.

We assume an SVP based system similar to the Microgrid, which has a fixed number of fixed-size (powers of 2) places with homogeneous cores. We emulate this system with our pthread based svp-plt [MWvT12] run-time that we discussed in Chapter 3. On top of this, we have implemented a single level SEP service to manage the emulated resources, and the discussed implementation and run-time (cf. Chapter 8). In order to emulate the execution of this many components on a commodity computer, we emulate the box workloads with an `usleep()` call to suspend for the calculated time period. We must stress that this experiment is a mere emulation instead of a simulation, as there is for example no modeling of network or memory bandwidth.

The SEP implementation is based on a `create` of an `sep.alloc` instance of the SEP to get a requested number of cores allocated. As the clusters of cores are assumed to be fixed in powers of two, this implementation only maintains lists of available places for each size (buckets), and then select and return a place of an appropriate available size for the allocation request. First, it checks if a place with a size the smallest power of two that encompasses the number of requested cores is available, or any smaller sized places. Only if no smaller sized places are available, it checks if any larger sized places are available. As the algorithm to allocate a place is quite simple, we decided to run this SEP on an exclusive place entirely, as it almost exclusively consisted of accessing the shared data structure. The `sep.free` is similar, its instance is started with `create` and it adds the place in the appropriate list again. At a later stage we measured the performance of a similar implementation of an SEP on the Microgrid simulation platform, and this showed that we can allocate resources this way in about two to three thousand cycles.

**Experiments**

In order to have feasible experiments using the infrastructure we have just sketched, we need to define the parameters for our emulation. The interval at which records are injected into the network is 10ms, while the interval of the Observer iterations is set to 50ms. This means that every 50ms the state of the queues is inspected and a resource allocation adaptation can be made.
We have chosen the following values for modeling the box workload, so that resource requirements will be very dynamic. Box\text{A} has a $W_p$ of 100ms and a $W_s$ of 10ms, Box\text{B} has a $W_p$ of 2000ms and $W_s$ of 200ms, and Box\text{C} has a $W_p$ of 500ms and a $W_s$ of 50ms. Additionally, Box\text{A} has no random factor so that $R = 1$, but Box\text{B} has a widely varying random factor between 1 and 10, and Box\text{C} has a random factor between 1 and 4 to simulate data dependent processing time. The relative light load for Box\text{A} and Box\text{C} compared to Box\text{B} relates to the idea of a multi-media stream processing application where Box\text{A} splits the stream, a computationally fixed operation, Box\text{B} does a data dependent heavy operation, for example video stream decoding, and Box\text{C} does a data dependent light operation, for example decoding the audio stream.

We assume a 1024-core Microgrid with statically configured places: 64 single-core places, and 32 places each of 2, 4, 8 and 16 cores. We only consider the dynamically allocated resources where the application run-time executes the box instances, and not the resources used by the SEP or the run-time itself, which would require a static number of places, at least three. Our experiment runs for an input of 2500 records, and we measure the state of the system on every interval of the Observer.

The results of the first experiment is shown in Figure 6.6, which shows how the size of the queues develop over time (Figure 6.6a), and where the Observer algorithm decides to make its adaptations. Each black marker on one of the three lines indicates that the Observer decided that iteration to request more resources to processing from that queue. Whenever the resource allocation for this adaptation failed, the black marker is replaced with a red cross, but this does not occur in this first experiment. The number of allocated cores per queue is shown in Figure 6.6b, including the total number of allocated cores. Only the absolute number of cores is shown, and not how the number of cores decomposes into the mixture of places of different sizes. As we can see, the allocation peaks at just over 900 cores, which leaves enough headroom on our 1024-core configuration.

We clearly can distinguish two phases in our experiment, an initial setup phase until around the 200th Observer iteration, and a semi-stable state from there onward. The setup phase also shows that the processing of Box\text{B} requires much more resources, and it takes a while before enough resources are allocated as only a single allocation is done in each Observer iteration. Sufficient resources to start reducing the queue for Box\text{B} is reached around iteration 80, but our algorithm will continue to add resources to clean up the slack of the queue as quickly as possible. This causes the highest point in resource allocation to be reached just as the queue for Box\text{B} empties for the first time, around the 220th iteration.

A completely stable state is never reached as Box\text{B} and Box\text{C} induce a dynamic load, and resources are released as soon as a box function finds its queue empty. The fluctuation in the number of allocated cores for Box\text{B} is higher, as it has a wider range of dynamic load. We see that the fluctuation for Box\text{C} is much lower, but we also still see a small fluctuation for the processing of Box\text{A}. Even though the input rate is steady as well as the workload for instances of Box\text{A}, it is clear that the system has trouble to exactly match the required processing power, and probably no exact match is possible. And as in this experiment the system reclaims resources very aggressively, resources might be released too soon.
In our second experiment, shown in Figure 6.7 we try to make the system more stable by making the release of resources less aggressive. In this version, a resource that has been assigned to a box function will only be released when it encountered an empty queue five times in a row. Every time it encounters an empty queue it sleeps for one interval, equal to the input interval. All other parameters are left unchanged.

We see a similar setup phase for each of the three queues in Figure 6.7a, but after this completes the system fluctuates much less, and the queue sizes in the second phase rarely grow beyond 5. Of course this comes at a higher resource usage cost, which we can see in Figure 6.7b, the pattern is overall the same but the fluctuations for each box have been reduced, and the total number of allocated resources is slightly higher.
Of course if we increase the number of intervals that a resource will be kept to retry to fetch a record from a queue, we can decrease the average queue size even further against a higher resource usage cost. This is shown in Figure 6.8 which shows the development of the queues with a 20-interval retry (6.8a) and 50-interval retry (6.8b). The queue sizes rarely grow beyond 1 entry, though there are still some peaks for the heavy random load of \textit{Box}_B. Also for the experiment with a retry period of 50 intervals, we start to hit the resource limits in the system and all 1024 cores have been allocated. This is shown by the red crosses which indicate that the resource request of the Observer failed when it tried to make an adaptation. However, this does not affect the processing of the queues though, as all three queues have already been over-provisioned with plenty of resources.
6.2. RESOURCE-ORIENTED RESOURCE MANAGEMENT

If we increase the number of retries even further to 100 intervals, we effectively end up in a situation where resources are acquired and almost never released again. This is shown in Figure 6.9, which similarly to the 50-interval retry result shows the failing resource allocations at some point, as all resources have been exhausted around the 250th Observer iteration. We also can see here by comparing the resource allocation in Figure 6.9b to our earlier results, for example in Figure 6.6b, that mainly the resources for \textit{Box} A are largely over provisioned, at least a 5-fold. The number of resources that are almost constantly allocated for \textit{Box} C are comparable to the short peak values in our earlier experiment. The same holds for \textit{Box} B, though this is being limited as we reach a point where all resources are already allocated.

Figure 6.8: Streaming application behavior on an emulated 1024-core Microgrid with a 20-interval and 50-interval retry before resources are released. Only the development of the queues over time is shown.
Figure 6.9: Streaming application behavior on an emulated 1024-core Microgrid with a 100-interval retry before resources are released. Resources are almost never released and allocations start to fail around the 250th iteration.

As a final experiment using this framework, we show what happens when we decrease the number of resources to a point where they become scarce but there is just enough resources available to be able to meet the throughput required by the input rate of the system. However, the headroom is greatly reduced, and it shows an interesting effect. Reducing the resources even further is not very interesting, as it will simply result in one or more of the queues constantly growing in size until the input is finished and then they will drain again as resources free up and can be reassigned.

After adjusting the parameters carefully, we reduce the number of cores in our configuration from 1024 to 832, by removing the 64 single-core places, and reducing
the number of 16-core places from 32 to 24. Also, we revert to our original aggressive approach of releasing resources, without any retry iterations. The results are shown in Figure 6.10, where we can identify three phases.

The first phase in Figure 6.10a runs until around the 165th iteration of the Observer, and is a similar setup phase to what we have seen in our previous experiments. However, at that point all the resources have been allocated as $Box_B$ is over-provisioned as the system is trying to get rid of the large queue of records. Then, a second phase starts until around the 320th iteration of the Observer, in which the system is unstable. Here, we see the same behavior as when there are not enough resources available to keep up with the rate of the input stream. The queues for $Box_A$
and $Box_C$ increase as the Observer fails to allocate more resources to them, as any recently freed resources have already been re-allocated for $Box_B$.

Around the 320th iteration of the Observer, we can identify a third phase in the experiment, where the balance in the system is restored again. At this point, the queue of $Box_B$ has been sufficiently drained, and we can now see an interesting resource sharing pattern between the dynamic workloads of $Box_B$ and $Box_C$ from this point onward. This is clearly visible in Figure 6.10b, which effectively shows the fine-grained space-sharing and time-sharing behavior of two independent 'applications' with dynamic resource requirements, when we view the collections of components of the application that execute $Box_B$ or $Box_C$ as independent entities.

While we see in our experiments that the approach we have taken manages to reach a semi-stable state, there is room for a lot of improvements. First of all, the adaptation algorithm to decide where and how many resources should be allocated could be improved. For example by reacting on multiple queues in each iteration, or by maintaining a larger history record and applying heuristics. On the other hand, it is interesting to see how well such a simple algorithm already performs for a highly varying system.

### 6.2.4 Summary

In this section we have discussed our approach of resource-oriented resource management, in which applications request a specific amount of processing resources (cores) similar to the `malloc()` call to allocate memory. We discussed how such a service would work internally and what information it should manage. We proposed two approaches to make it scalable by employing multiple layers and exploiting locality. Then, we presented our evaluation with a dynamic streaming network application and associated run-time system that dynamically requests and releases resources to adapt to the needs of the application. While the resource manager has no model or data of the requirements of the whole application, the interaction with the run-time system which is requesting and releasing resources leads to a relatively balanced situation. The interaction between the resources assigned for the different components of the streaming application also show that multiple applications can use this to do both time and space sharing as long as they dynamically request and release resources, instead of monopolizing the resource pool.

### 6.3 Service-oriented Resource Management

The second approach to resource management that we present aims to support heterogeneous architectures and is based on expressing requirements for certain functionality and performance. Instead of requesting a number of general purpose cores, we take a service-oriented approach, i.e. where the applications asks the SEP; "Please find me a resource that can execute function X". We name it a service-oriented approach as the resources offer a set of SVP components (cf. Chapter 3) which they can execute as services and which implement different functions on this resource either in hardware or software (for example a Matrix multiplication, FFT, DES encryption). We are not
6.3. SERVICE-ORIENTED RESOURCE MANAGEMENT

considering run time code specialization for heterogeneous resources here where parts of the application are dynamically specialized to execute on these resources.

Expressing which functionality is required will not be enough, and additional constraints or wishes should be possible to be expressed as well. For example, the performance delivered by the requested component, or the amount of energy or power it uses to execute a specific task, depending on what the application or system wants to optimize for. This means the request will be more like "Please find me a resource where I can execute function X as efficient as possible", where efficiency is related to an optimization goal, evaluated by some fitness function which can be based on performance, power, energy, locality, or other parameters. As each resource offers a set of possible functions, and no arbitrary code is executed on them, some of these behavioral parameters can be determined beforehand.

The advantage of a component based approach is that it does not require support for pre-emption to share resources between applications, as long as a component is guaranteed to execute for a bounded amount of time. A resource could be assigned to an application for a certain number of component invocations, which can be specified in the resource request. After these executions on the resource have completed, it is released again and can be allocated to a different application.

The reason we look at this approach is twofold; first of all it is clear that the first approach presented in Section 6.2 cannot work for heterogeneous architectures with specialized units. Secondly, it fits well with the collaborative ÆTHER computing platform where heterogeneous nodes with specialized units co-operatively share resources, and this approach was developed by us as part of that project (cf. Chapter 1). Therefore, we present this section from the ÆTHER platform perspective, in contrast to the Microgrid perspective that we have used throughout most of this thesis.

The ÆTHER platform consists of a loosely coupled collaborative environment of SANEs, Self-Adaptive Networked Entities [CSA8], which are computing elements that support the SVP protocol and expose an SEP protocol interface. In other words, an application can delegate work to a SANE using the create action, while addressing it by a place that it acquired from the SEP. SANE implementations that have been investigated in the project consist of hardware that can adapt itself to optimize for specific functions, for example using on the fly reconfiguration of FPGA areas [CSA36] or programmable floating-point pipelines [CSA28].

In such a collaborative environment, the SEP acts as a lookup service and negotiator for services between the SANEs. The environment is dynamic, as SANEs can join and leave the network, so this information needs to be kept up to date. It should be noted that some but not necessarily all SANEs require some general purpose computing capability to execute the control code (cf. Chapter 3) of an application, while specialized SANEs only execute certain specific SVP components.

We will now discuss the protocol that we designed to support this platform, which was also published in [MWvT4]. Similarly to the resource-oriented SEP that we presented earlier, this protocol is implemented on top of SVP based delegations. Again, it also requires implementation specific local interfaces to configure the resources, for example to reconfigure an FPGA to load a specific function.
6.3.1 Protocol

The protocol consists of seven stages: announce, request, bid, accept, delegate, release and withdraw. Specific implementations may omit stages that are implicit, for example, in an on-chip environment the functionality of each component will be known beforehand, and does not need to be announced. The agents that implement this protocol are the SEP instances, where each SANE has its local SEP to govern the resources on that SANE. The design is, again, hierarchical, and a cluster of SANEs have their local root SANE running a root SEP. These roots expose the same SEP and SVP interfaces to other clusters again, representing the cluster of SANEs as a SANE itself again, which makes the system composable. The root SEP also functions as the local directory where resources within the cluster are registered, as we will see now when walking through the stages of the protocol.

Announce and Withdraw

In the first stage of the protocol, a SANE joining a cluster announces its functionality by presenting a list of the components it can provide to the root SEP. Before this can be done, a protocol will first have to be established that implements SVP. This will initialize the joining SANE with the place of the root SEP which is required to initiate the SEP protocol. Which is similar in concept to automatically receiving the address of the DNS servers in a conventional computer network.

The joining SANE announces its arrival with a create of the SEP_announce function at the root SEP. As a parameter, it passes record(s) defining its components, their types and the average throughput it can achieve for these components. Note that the components can be represented at various levels of granularity, i.e. from arithmetic operations to complex components.

When a SANE wants to withdraw its resources from the pool again, it does this with a create of the SEP_withdraw function at the root SEP. Of course, this may not always happen gracefully, which means that timeouts are required on all inter-SANE calls in case a SANE and its SEP suddenly disappear. When such a communication failure is detected, this is reported to the root SEP and the resources belonging to that SANE are removed from the resource pool.

Request

Having announced itself to its environment, a SANE may now make or receive requests for resources. It should be noted that an application running on this system is in fact a distributed application that runs on the whole network of SANEs. When it wants to execute a specific component, it asks its local SEP to find a resource that can execute it, which, unless it can satisfy the request locally, makes this request to the root SEP. The root SEP will then broadcast the requests to any SANE in its environment that it knows is possibly capable of meeting the request. This broadcast can be done concurrently and independently with a local create of a family with a thread for every remote node that needs to be contacted, which in turn then each handle one remote create to send the requests.
The request sent out by the root is defined as a required performance on the given function but also includes an estimated time for which the resources are required. A timeout is attached to each request, which is the validity of the invitation for the SEPs to send a bid in response, which are matched to the request by a unique ID that was provided for the request by the root.

**Bid**

Each bid provides a yes/no response to the request and if yes, it will provide a cost for meeting the request. This incorporates the time required to configure the resources for the requested function, a lifetime for the bid (the SEP will reserve these resources for this amount of time), and a limit on the time that it is able to provide these resources, which may be less than or greater than the time requested. The bid also includes an identifier and a place to where the potential agreement for the bid must be sent.

While we just used time as a cost measure, we can also replace this with energy which allows different optimization goals. The choice of the cost model depends on what the implementation wants to optimize for, and can also be a fitness function based on a combination of multiple factors, i.e. both time and energy.

The bids are sent back through the root SEP, which will then forward one or more bids back to the requesting SEP. It optionally filters out some bids that are considerably worse than others, and already inform the bidding SEP that it does no longer need to reserve its resources for the bid. We leave the question open if the root SEP should determine the winning bid and only return this to the requesting SEP, or that all relevant bids should be forwarded and the requesting SEP should decide depending on the parameters set by the application when making the request. It does not make a fundamental difference in the protocol, but filtering requests at the root SEP will reduce the required SEP protocol traffic.

**Accept**

After the requesting SEP has received one or more bids, one is selected which meets its requirements. It does a create of the SEP_accept function to the offering SEP, which in response will configure the resources on that SANE for the requested function. The unique ID used in the bid that was returned will be used again to identify the reserved resource at the offering SEP. Besides configuring the resource itself, also security and protection requires to be set up (cf. Chapter 5), and a place is returned to the requesting SEP with all necessary capabilities to be able to delegate work to the offering SANE. Bids that are not accepted can either time out or be explicitly rejected by the root or requesting SEPs.

**Delegate and Release**

All that is left to do when the place has been configured and returned to the requesting SEP is to return it to the requesting application. This in turn can then start to delegate work using create to the newly acquired place. When it has completed all the planned work at the place, it will ask its SEP to release it again. This means doing
a `create` of `SEP_release` to the corresponding SEP of the resource which unconfigures the place and its capabilities, which restricts access again from the application that just finished using it, and allows it to be offered again in response to new requests.

### Overview

![Figure 6.11: Overview of the SEP protocol for service-oriented resource management](image)

An overview of the protocol and the interactions we have just discussed is shown in Figure 6.11. The requesting SEP sends a request to the root SEP, which distributes it to the SANEs that are registered with that functionality. Two of them send a bid back and these are sent back to the requester, who selects the bid from the SANE at the bottom, accepts the contract and delegates the work. After this is finished, it releases the resource on the SANE again.

#### 6.3.2 The root SEP

In the discussion and description of the protocol we had so far, the root SEP is at the physical root of the cluster which is responsible for maintaining a complete picture of the capabilities of all SANEs that have announced themselves within the cluster. It also provides an interface to the next level of hierarchy, and it is trivial how each SANE can find the root at initialization. The problem with this implementation is that it relies on the root SANE being fault tolerant, as it is potentially a single point of failure in the entire system. Contention of the root SEP is less of a problem, as soon as a single root SEP becomes overloaded, its resources can easily be partitioned and allocated to two local root SEPs known by two subsets of SANEs, and one new root SEP above them. Then, the same protocols can be used again at the higher level between the root SEPs.
Conceptually, the root SEP and the protocol admit different implementations. The root SEP is first and foremost, the place to which a SANE announces itself and to which it directs requests for resources. It is assumed that directly or indirectly, all known SANEs in a cluster may be reached from this place. We will now shortly discuss an alternative distributed root SEP organization.

**Distributed root SEP**

In this approach there is no single root SEP, but every SEP at every SANE has a notion of which services are provided by the SANEs in the neighborhood. This can be done by maintaining a list of services that is filled by exchanging information about known services and their location using epidemic algorithms [47] with other SANEs in the proximity. This is a common approach to implement gossiping in peer-to-peer systems to spread information through a network, and has maximum redundancy [85, 56].

The announce step of the protocol when a SANE is joining the network will then consist of initiating the epidemic protocols and starting to publish information about the services offered by this SANE. Withdraw is no longer needed as the information will slowly disappear from the system when the origin is no longer disseminating it. The request step will be a lookup in the local list and then contacting the appropriate nodes, or potentially asking nodes in the proximity if they are aware of any location providing the requested component. The bid, accept, delegate and release steps are unchanged, assuming that it can reach, possibly through multiple hops, the node providing the service.

While this just skims the surface of such an approach, sketching out its possibilities, it sounds promising enough for further investigation in future research. However, it should be noted that while this seems very suitable for systems such as the ÆTHER platform with its dynamic ad-hoc collaborative network, it is highly unlikely to be applicable to future heterogeneous many-core architectures.

**6.3.3 Evaluation**

We now evaluate our SEP framework with an implementation running an adaptive least mean squares (ALMS) filter application. This application was developed by our partners from UTIA, Prague within the context of the ÆTHER project. It runs on top of our framework consisting of our distributed SVP implementation [MWvT13] (cf. Chapter 3) and our implementation of the service based SEP resource management protocol. It was one of the final project demonstrations, and has been published in [MWvT14]. We choose to discuss this application here as it dynamically requests different components of which we had different implementations, and it was very suitable to capture and display the behavior of the system for discussion.

**Adaptive Least Mean Squares Filter**

To demonstrate the possibilities of our framework, we use an application that implements an adaptive least mean squares (ALMS) filter, shown in Figure 6.12a, based on the well-known least mean squares (LMS) [162] filter. We send 2000 input samples
to four instances of the LMS filter using different learning rate coefficients, where the initial values $\mu_m(0)$ are given by the following vector:

$$\mu^T(0) = [0.015, 0.012, 0.01, 0.009]$$

The learning rate $\mu$ determines the step size of the LMS towards the minimum of its cost function, where $\mu$ influences the convergence rate and precision of the result. After each batch, the $\mu_m(n)$, for which the LMS reached the best score is computed as $\mu_{best}(n)$, which is used to generate new learning rates $\mu(n+1)$ for the next batch using the following equation:

$$\mu(n+1) = [1, 1.5, 1.2, 0.9]^T \mu_{best}(n)$$

By using four LMS filters we are able to improve tracking of the non-stationary model parameters. The adaptation of learning rates is shown in Figure 6.12b. Parameters of the stationary model were estimated up to iteration $n = 50$. After that, the input was switched to a different model, instantaneously changing the estimated parameters. As a result, the learning rates quickly increase after iteration $n = 50$, and as the system adapts to the new condition, the learning rates slowly descend back to their original values.

**Application Programming Model**

We have multiple ways available to implement the ALMS application, as the ALMS filter can be run as a single service, or composed by combining multiple smaller scale services. Executing one iteration of the ALMS filter can therefore be done as a single component itself, or, when this is not available, by composing it of four LMS components followed by 32 vector product (VPROD) operations.

The application acts as a client in our framework sending a request to the SEP to determine if the ALMS component is available. If it is not, it sets up four asynchronous
threads, using create, that request and execute an LMS component from the SEP, and sync is used as a barrier to wait for all four to complete. Each thread starts with a blocking resource request to the SEP. Once the resource has been acquired, the operation is delegated to that resource, and after its instance finishes, the resource is released. Then, the same sequence is done for using the VPROD service. The decision about the best filter result is obtained by the summation over VPROD results and by their comparison.

In the end, the sync operation synchronizes all outstanding activities. As these activities are run as independent asynchronous threads, they can use one to four parallel LMS filters and one up to 32 parallel VPROD operations. When there are not enough resources available to execute all instances in parallel, the resource request to the SEP will block until another instance has released the resource. Such solution hides the number of real resources involved in the computation, as the concurrent execution is constrained by the availability of resources. Therefore, the performance can scale up and down dynamically according to the run-time availability of computing places. However, if we need to meet certain real-time constraints, we could also decide to adapt the quality of our solution by changing the number of different learning rates we try each iteration.

Platform

In our case study we use a platform based on three Xilinx ML402 prototype boards [166] connected through switched Ethernet. Each has a Virtex2 SX35 FPGA that is configured to contain a MicroBlaze RISC processor [167] on the FPGA fabric. Programmable hardware accelerators [CSA27, CSA28, CSA6, CSA26] are connected to the MicroBlaze to accelerate floating-point DSP algorithms. The possibility to program the accelerators with microcode is one of the key features of our solution; the SANE implementation can provide hardware accelerated functions that can be changed on demand.

On the board we run an embedded Linux and use our POSIX threads based implementation of SVP [MWvT12], which was extended [MWvT13] with a lightweight TCP/IP message protocol to uniformly handle concurrency and communication on our distributed platform. The SEP is implemented on top of this, exposing the accelerators and the MicroBlaze core as places offering services. To implement these services the accelerator microcode for both the LMS and the complete ALMS filters was developed, and VPROD was already available. These three are encapsulated as services exposed by the SEP, where each node can support one or more of these services simultaneously. This is discovered by the SEP when a node starts, and the available services are announced to the network.

For our experiment, we can configure the nodes to support a different set and number of services. By removing a node from the network and then joining it again in a different configuration, we can simulate a larger heterogeneous environment with many different types of resources that come and go. We show how the framework and the application automatically adapt to these changes as the application uses resource neutral code that requests certain components, and these are executed on the available resources.
CHAPTER 6. RESOURCE MANAGEMENT

Figure 6.13: Adaptation of the ALMS application to the available resources

Results

In our experiment, we show the behavior of the ALMS application running on the framework when resources are entering and leaving the system at run-time. Figure 6.13 shows the presence of resources and the development of the speedup of the application compared to running the application on the bare minimum of resources it requires to successfully execute (one LMS, one VPROD).

At the start of the run, the user application cannot execute its computations because there are no available resources. It starts immediately when a SANE, providing one LMS and one VPROD instance enters the system. When another SANE providing these services enters the system at iteration \( n = 10 \), the application starts to use both in parallel and the performance is improved. Now all three nodes are in action. One with the user application and two with SANEs, each containing one place. After that, around \( n = 20 \) we withdraw one SANE from the system again, and replace it with a new SANE which contains four places able to execute four LMS or VPROD in parallel. The application starts to use four LMS filters in parallel. At the same time, the application uses all five available VPROD operations. When the second SANE with one computing place is replaced by the SANE with four places at \( n = 32 \), the number of VPROD operations in use reaches eight. However, in this case the number of used LMS filters remains at four as the application cannot use more LMS operations in parallel. At this point the performance of the system decreases because the overhead of executing 8 remote VPROD operations. Finally, the SANE capable of computing the complete ALMS filter appears at \( n = 47 \). Since the user application prefers to use this implementation, the ALMS is used instead of the composition of LMS and VPROD. This configuration shows the best performance. When the ALMS
capable place leaves the system again at \( n = 74 \), the backup solution based on using the LMS and VPROD operations is restored.

Further measurements on our framework implementation showed that the minimum overhead for calling a remote function is 16ms on a 1 Gbit Ethernet network connection from a PC to our FPGA boards, and as low as 0.3ms between two PCs on a 1 Gbit Ethernet connection. This means that the software components that are distributed across the network need to be of sufficient granularity to hide the latency of the network overhead. However, we measured that a single iteration of the ALMS software implementation takes 6.8ms on the MicroBlaze on a node, and 0.7ms when executed in the hardware accelerator. Taking into account that the requests to the SEP take multiple remote calls, the granularity of the distributed software components turned out to be too small compared to the framework overhead. This can also be observed in the result of the previously explained experiment when there were 8 VPROD operations available on the network, the overall performance of the application would decrease due to network overhead.

6.3.4 Summary

In this section we have covered our approach for service-oriented resource management. The SEP exposes resources that offer implementations of specific services, and uses a cost model to optimize the allocation of resources for a certain goal, for example energy or performance. The SEP protocol for this has been established with the distinct announce/withdraw, and request/bid/accept/release stages. We have shown in our evaluation using an example application how these services can be dynamically used and how this behaves when resources appear and disappear in such a system.

6.4 Discussion

In this chapter we have covered two approaches to resource management which are quite opposite extremes of each other. First we discussed a resource-oriented resource management where we allocate numbers of cores in a way similar to memory, which is useful in general-purpose systems which are perfectly homogeneous with many, many cores. While it is a very simple approach, it lacks support for heterogeneous systems as it only has one simple way to express the requirements for a resource.

On the other end of the spectrum, we have discussed a service-oriented resource management approach where the heterogeneity of the system is masqueraded by abstract services. The application no longer needs to deal with figuring out how much resources it actually wants to have, but asks the system to find it a resource that can execute a specific component as efficient as possible. While this is an interesting approach for the collaborative ad-hoc network of the ÆTHER platform, this is not suitable for general-purpose many-core systems. We can not expect that every possible component of every possible application will be captured somehow and that every such service will have a uniform and well-defined interface.

The question is, what is a feasible approach for future many-core systems? This will be a major challenge in future research extending out from the ideas covered in
this chapter, but we will try to sketch out some ideas. First of all, these many-cores will very unlikely be perfectly homogeneous, for example concerning memory access time and memory bandwidth. This means that from the resource-oriented approach perspective we need to include more information in our requests so that the SEP can make a better decision in where to allocate resources. This is not unlike the mechanism for NUMA-aware memory allocations where locality matters, and such an affinity metric between processes was also implemented in Helios [114].

For example, if an application gets a cluster of cores allocated at the other end of the chip, this might swamp the memory network resulting in an overall degradation of performance. In fact, it might not even be beneficial for the performance of the application, so it would be better off not to receive those resources. Another example would be a situation when the requested place will receive a lot of delegations from the application, for example when it requests an exclusive place to use for synchronizations. This means that this will receive a lot of traffic on the delegation network, and it would benefit from having this place allocated as closest as possible to the other places that will access it to reduce network traffic and latency.

A solution to this would be to include QoS support on all on-chip networks for memory, delegation, possibly I/O, and cache coherence. Such QoS on channels is one of the key points of the approach in Tessellation [104, 39]. However, there is likely more heterogeneity involved, as we have discussed in Chapter 2. For example, a many-core chip can have a combination of simple and fat cores, or differences in clock speed based on heat or energy limits. Then, the application would have to express requirements for example in terms of bandwidth, access time, locality, computing throughput and concurrency for a resource, and the system could reserve channels accordingly that enforce the requested QoS. However, this puts a large burden on an application developer to express such requirements, which in turn are not independent of many other factors such as the current set of resources allocated to an application and the properties of the machine it is running on. Tackling this problem is one of the most difficult challenges for future many-core systems.

It could be that we move even further towards heterogeneous systems by including specialized components, perhaps reprogrammable, but only supporting a limited set of functions. As we have seen we can not allocate these in the general-purpose way of our resource-oriented resource management, not even when we extend it with additional requirements for our resources that we just discussed. In this case, a simplified version of our service-oriented approach can be used. Simplified, as we do not expect resources on chip to come and go, so omitting the announce/withdraw stages. A combination of approaches where we use the resource-oriented approach for interchangeable general-purpose computing resources and the service-oriented approach for specific functions on specialized execution units could perhaps be interesting.
7.1 Introduction

In this chapter we will discuss our approach for the last missing point of our list of operating system requirements from Chapter 2, which is how to deal with I/O. While this would in theory not be radically different from contemporary systems, or from any other service based approach that we have discussed so far in this thesis. However, when we manage concurrency in hardware we know that implementing interrupts is not so trivial [MWvT8], as what will we interrupt? Which of the potentially many concurrent running contexts?

Furthermore, with the integration of a large number of cores on a chip, we can expect that not all of them might be connected to an I/O network, and perhaps even a few certain cores are dedicated to taking care of I/O. We explore this by presenting our design for an I/O interface for the Microgrid architecture which uses specialized I/O cores, and we will discuss the software stack on top of this as an example of an SVP based I/O service. We only concentrate on the communication with a device, as the creation of higher level services on top of I/O devices, for example a file-system service, can be trivially constructed on top of this following from this study and the other mechanisms described in this thesis.

We will first discuss a short overview of related work in this area followed by the description of a generic SVP I/O Service, which assumes that only certain I/O places in the system are capable of physically accessing I/O devices. The internals of this service is discussed in detail, also to give an example of how a system service can be structured and implemented on an SVP system. Then, we will discuss our I/O Interface design for the Microgrid which implements an I/O place, and show how this works with the I/O service. We conclude the chapter with results of measurements of an initial I/O Service implementation for the Microgrid.

The work on the initial design of the I/O system described in this chapter has been jointly developed with M. A. Hicks as part of the Apple-CORE project, and has been published in [MWvT3, MWvT11]. The Microgrid implementation of the
I/O Service used for the experiments in this chapter has been (re)-developed by the author, with some assistance of R. C. Poss.

7.2 Related Work

The use of dedicated programmable processors to handle I/O is not something new, having been first introduced in 1957 when it was implemented in the IBM 709 system [83]. Following this development, the IBM System/360, System/370 and the architectures that superseded them have featured channel processors for high performance I/O [31]. Another system from this period that had even more similarities to the Microgrid implementation, that we will present in this chapter, was the Control Data CDC 6600 [147], which had a dedicated I/O processor that shared 10 distinct I/O processor contexts.

These I/O processors were very limited and only supported a simplified instruction set for handling I/O, and they are not very different from the programmable DMA controllers found in contemporary computer systems. Both approaches serve the same purpose: to prevent the CPU from being frequently interrupted during I/O operations, and to allow the overlap between I/O and computation. This is also a problem in real-time embedded systems [133], where it is common that state-of-the-art micro-controllers are equipped with a peripheral control processor which can be used to handle interrupts while the main processor is still guaranteed to meet its real-time obligations.

The Helios OS [114], that we have mentioned several times before, also takes an approach of distributing small satellite kernels to programmable I/O devices in order to offload the execution of programs and system services. As we mentioned in Chapter 6, it is important to control the location where certain parts execute, for which Helios uses affinity meta-data to hint about efficient placement of such programs to put data processing close to its source. We believe that this would be the task of the SEP to intelligently place components to exploit the locality of data, which requires resource requests to express their affinity with I/O services. Others have also observed this as a problem, and suggest [28] that in order to achieve high throughput I/O, tightly coupled communication between the components, with no global interactions, is highly desirable. This suggestion is in line with the service based, distributed, operating system design that we are covering in this thesis.

7.3 I/O System Service

When implementing device I/O on any SVP system, including the Microgrid architecture, there are two distinct levels at which we need to tackle it. First, we will consider the implementation of I/O at the software and SVP level, where the traditional requirements of device I/O must be combined with the advantages, facilities and restrictions of the SVP programming model. In practical terms, this is an I/O service that is part of the operating system API. It provides I/O as a service which can be used by other Operating System components, or by user programs.
The I/O model that we describe here is designed to be very general, in which we assume that we read and write chunks of data from/to a device. This is represented by providing a familiar software interface to client SVP programs: the standard `read` and `write` system/library calls are implemented and both synchronous and asynchronous modes of operation for I/O are supported. The example that we will discuss in detail focuses largely on devices that communicate in a ‘request-response’ form (e.g. a block device), since this is the most typical kind of device use-case, but the model is not limited to those types of devices. It should be noted that the described design forms the generic basis of the I/O system. As a result, it is possible to encapsulate calls to this service with further APIs to higher level services, for instance a file system service which provides an abstraction over the structure of data on a device and would thus issue potentially multiple I/O actions for a single request to the system.

Figure 7.1: An overview of the Generic I/O Service software stack which implements generalized I/O events using SVP. All processes inside the two light gray areas are part of the system’s I/O service. Processes in the low level driver enclosure are grouped separately, as they are executed at the designated SVP I/O place. Objects in square-cornered boxes represent threads; round-cornered boxes represent state or interfaces with which these processes interact. Dashed round-cornered boxes are conceptual groupings and privilege domains (where Client A and Client B are external applications or higher level system services).

7.3.1 I/O Model (Synchronous and Asynchronous)

Figure 7.1 shows an overview of how the I/O service works, including the interface with synchronous and asynchronous read and write calls. This approach is best explained by stepping through each component in the diagram individually.

Client Requests

Two client processes are shown, A and B, that respectively want to issue a synchronous and asynchronous read or write request to a device. Each process as well as the I/O service are shown as dashed rounded boxes, which indicate their protection domains.
and encompasses all the components of which the processes consist. Using the I/O API, the read or write requests are prepared, and the corresponding device descriptor node for the device they want to access is looked up. Then, they trigger entry into the generic I/O service and a switch of protection domain using the special indirect create for system calls that we presented in Chapter 5. This then takes care of creating the appropriate thread of the I/O service to which parameters such as a size and pointer to a buffer for sending/receiving data are provided.

Client process A issues a synchronous read or write action, and also sets up one of the synchronization mechanisms that we discussed in Chapter 4. A reference to this synchronizer is passed on with the request that enters the I/O service, so that it can be signaled by the I/O service when the request has been serviced and has completed. The library code that implements the I/O API within Client A then waits on the synchronizer until it returns, in order to deliver the semantics of synchronous I/O calls. In our prototype implementation of the I/O service we have used the sear thread mechanism to implement this synchronization.

Client process B issues an asynchronous read or write action, and provides a reference for a call-back thread that needs to be created by the I/O service when the read or write request completes. Needless to say, this means that privileges need be set up appropriately so that the I/O service has the rights to create a new thread in the protection domain of Client B. Calling the I/O service is made asynchronous by implementing it with a detached create so that execution in Client B can continue while the I/O operation takes place. It should be noted that Client B should take care of the concurrent execution of its code by the asynchronous callback – the burden lies within the client process to ensure synchronization of its data-flow around this event.

Device Descriptor Node

The device descriptor node, which is created and set up when initializing a device driver, is a data-structure in the memory of the I/O service which contains the persistent state and necessary information for the I/O service for that device. It has references to the relevant data structures (queues, buffers, device I/O place) and is located in the memory of the I/O service so that it can be only accessed by privileged code that is part of the I/O service itself. To guard the safe access and updates to this data-structure, all accesses to it are delegated through an exclusive place, which is not shown in our diagram.

Request Queuing

When the service is entered with a read or write request, the Queue Request component is executed. This validates the request, and reads the device descriptor node to determine which device queue the request should go to. It then adds the request to the appropriate queue of pending I/O requests for that device, an update that also uses an exclusive place. From the device descriptor node it determines if there is already an instance of the Dispatch Controller active, and if it is not it invokes a Dispatch Controller instance to start processing requests from the queue. The Queue Request can also handle any priority I/O scheduling by taking this into account when updating the request queue.
Request Dispatching

The Dispatch Controller examines the request queue to see if any I/O operation is currently pending. If no I/O operation is pending, it updates the device descriptor node to the state that no Dispatch Controller is active for this device and it terminates. Otherwise, it updates the state to active and it takes a request and performs a sequence of one or more create actions of the appropriate Low-level driver threads to carry out the I/O request. These threads are delegated to the I/O place at which the communication with that device can take place, and they are created with the appropriate parameters to carry out the particular read/write operation; for example the size, target address, and a channel number. One request can correspond to multiple I/O operations, for example if this is on a multiple channel device, and one low-level driver thread is created for every channel.

After the I/O request has been serviced by the low-level drivers, they acknowledge their completion or return the values read from the device. The Dispatch Controller waits on this using the sync action on the created threads. The returned values are then passed on to a newly created instance of a Completion Handler, while the Dispatch Controller starts over again and examines the request queue if there are any other I/O requests for this device pending.

Low-level driver

In the Low-level driver thread, a single I/O operation is handled and the actual communication with the device takes place through a local I/O interface. This driver thread is the component that needs to know how to control the low level interactions with the device to have it perform the required operations. The low-level driver thread uses a special mechanism in the I/O place to suspend, waiting on an event from the External Device while the operation is serviced by the device. When the low-level driver wakes up again, the resulting data is read from or written to the appropriate memory location using the I/O interface. Upon completion of the operation, the low-level driver terminates and returns the result to the Dispatch Controller.

Completion Handler

The Completion Handler thread, created by the completed r/w action, is responsible for ‘waking up’ the client. Based on the information in the request for the particular I/O operation, the completion signaler will either: signal the appropriate synchronizer (cf. Chapter 4) of the client (allowing execution to continue after a synchronous I/O request) or perform an create action on the specified ‘call-back’ thread in the client, for asynchronous I/O.

In the case of signaling the completion of a ‘read’ operation, the data needs to be transferred to the application. Ideally, this would use a zero-copy operation where the memory from the buffer is remapped to the application’s protection domain, but for a discussion on the issues of such IPC, and ways to implement this, we refer back to our discussion on this in Chapter 5. Alternatively, the application could ask the Protection Domain Manager to allow the I/O service and its driver to directly write to the buffer location given for the ‘read’ operation.
7.3.2 General Notes

All device controllers in this model have to be initialized with a number of desired behavioral parameters. An important parameter is the location of the buffers for the I/O data. These need to be allocated as a part of the I/O service if no zero-copy operations are possible to directly deliver the data to the requesting applications.

The previous example describes the request-dispatch device controller behavior, where I/O requests trigger the dispatch of a low-level operation as required. However, in order to handle externally triggered devices that would normally work with interrupts, the I/O service can also repeatedly issue a continuously listening low-level driver thread which fills an internal buffer in the I/O controller stack. This can be trivially implemented in the Dispatch Controller, of which one permanent instance needs to be active. Subsequent I/O read requests for this device can then be served the internal buffer. Alternatively, a call-back thread (similar as with the asynchronous I/O example) can be registered as part of the device descriptor node, which is started every time an event comes in from the I/O interface, effectively mimicking an interrupt handler in a highly concurrent environment.

7.3.3 I/O Places

An I/O place (shown at the right-bottom in Figure 7.1) is a place at which a particular device is physically accessible, in the sense that communication with that device can be locally achieved through some I/O interface. An I/O place does not necessarily have to be remote, however in the Microgrid implementation it will map to an I/O core, that we will discuss later on. In a more generic, for example software, implementation of SVP, this I/O place could be a remote environment or machine that provides access to a particular resource. Importantly, more than one device may be associated with a particular I/O place, which is captured by the channel identifier.

7.4 Microgrid I/O Implementation

The I/O service that we just described requires support from the specific SVP implementation; it must provide the special I/O places on which instances of the low-level driver can communicate through some interface to the device. We will now describe an implementation of such an I/O interface in the hardware of the Microgrid by introducing special dedicated I/O cores. An overview of the hardware scheme that implements this is shown in Figure 7.2, where external devices are connected via a HyperTransport-like bus to dedicated I/O cores. The scheme is based around a model of packet based communication via a bus interface, similar to message signaled interrupts, or MSI, in the PCI specification [118], or the newer PCI-Express bus. We will highlight and describe the components that make up our I/O interface.
7.4. MICROGRID I/O IMPLEMENTATION

Figure 7.2: A schematic overview of the architectural implementation of I/O in the Microgrid. An enlarged view of individual I/O core is shown to the right. The L2 Caches shown are part of the on-chip distributed cache network and are shared between multiple cores. Individual external devices are shown to the left, including a potential legacy DMA controller. Components are not drawn to scale considering chip real-estate.

7.4.1 I/O Cores

On the Microgrid, an I/O place is implemented by special dedicated I/O cores that are a derivative of the regular SVP cores. They can be distinguished from other general purpose cores in the Microgrid by the following characteristics:

*Bus Interface* – the I/O core contains a Bus Interface device which connects the I/O core to a bus for high-speed communication with external devices. This hardware is only present in I/O cores.

*Simplified logic* – given the specialization of the I/O core as a place only for performing I/O, it does not need to contain additional logic for performing computations such as floating point support. This removes the need for a floating point unit, as well as a floating point register file. The size of the integer register file can also be reduced compared to general purpose cores, given the simplicity of the threads the I/O core will be executing.

*I/O Instruction* – the pipeline of an I/O core supports an additional I/O instruction which allows threads to issue and wait for events from the high-speed Bus Interface.

The purpose of the distinct I/O core is to relieve fully-fledged Microgrid cores of the burden of I/O operations, and the additional complexity of providing every core with an I/O interface and connection to I/O buses. It also allows for a higher level of parallelism when I/O takes place in the Microgrid as the I/O core allows the general purpose cores of the Microgrid to continue executing in parallel while I/O operations are serviced. The reduced complexity of I/O cores, needing only to handle simple atomic I/O operations, means that their footprint in chip real-estate is relatively small, potentially allowing for more parallel I/O places or general-purpose cores in a fixed transistor budget.
Alternatively, I/O cores can be made just as complex as general purpose cores while additionally having the I/O interface. This allows for more flexibility where the I/O cores can also be used for general computations when not much I/O activity is required. However, we still assume that not every single core will be connected to I/O, and there are still specific I/O places where we need to delegate our special low-level driver threads that can talk to the I/O interface.

### 7.4.2 High-Speed Bus

Each I/O core interfaces with one or more high-speed packet/message based buses by means of an on-chip bus controller. While in principle a variety of packet-based buses could be used, such as HyperTransport, QPI or PCI-Express, we have based our ideas on the specs of the HyperTransport specification [82], and therefore refer to our bus as a HyperTransport-like bus. We have selected the HyperTransport bus based on its ubiquity in various high-speed I/O applications, the ability to interface with a variety of existing high performance devices, and its proven reputation for on and off chip processor interconnects.

We specify the bus as HyperTransport-like because it is unlikely that it would be necessary to implement all of the features of the HyperTransport specification. The hardware specifications of HyperTransport rather serve as a basis for what is achievable in terms of transfer rates and implementation in future simulation. The current HyperTransport specification (3.1) stipulates a maximum bus width of 32 bits with a unidirectional transfer rate of \(25.6 \text{ GB/s}\). However, there is no need why it needs to be HyperTransport, and it could also be changed for another packet/message based bus, such as QPI or PCI-Express.

As can be seen in Figure 7.2, the bus need not be connected directly to a single device, but can itself be connected to another interface or bridge which multiplexes between several devices using virtual channel identifiers, including, for example, legacy bus implementations.

### 7.4.3 Bus Interface

The bus interface is a small piece of logic (comparable to a simple interrupt controller) present at I/O cores which is responsible providing the processor pipeline with an interface to the bus events and messages. The bus interface performs the following actions:

1. Compose and decompose bus messages appropriately, based on the associated channel information.

2. Deliver channel events into appropriate I/O control registers.

3. Reading or writing the data payload of a bus message to/from a specified location – which can either be the register file or directly to/from the memory subsystem.
### 7.4.4 Device Communication and Interrupt Handling

The pipeline of I/O cores contain additional support for a special I/O Control Instruction, `ioctl`, which is used to read/write information to the I/O control registers which control the operation of the I/O Bus Interface. It can be used to perform an individual read or write operation on the bus in order to communicate with an external device and it provides the associated synchronization with the performed operation.

The low-level driver that we described in our section on the general I/O system service uses this instruction to send and/or receive bus messages. By specifying a channel identifier it can select a device, and additional options control the source/destination of data, to either send/receive data from/to the distributed cache interface or the register file. These two modes of operation, Register \(\leftrightarrow\) Bus and Cache \(\leftrightarrow\) Bus, allow for the differences in performance trade-offs; Register \(\leftrightarrow\) Bus communication is useful for small low-level device control messages, which fit in a few registers. High-volume transfers can, in principle, be carried out through registers, however the implementation of the Bus Interface also permits a directly reading or writing the data from/to the cache interface of the I/O core, directly accessing the on-chip memory network (see Figure 7.2).

### 7.4.5 Synchronizing and Interrupts

Synchronization is achieved using the synchronizing behavior of Microgrid registers at the I/O cores, and the data-flow scheduling that the pipeline implements. It uses exactly the same mechanisms as other long-latency operations in the general-purpose cores on the Microgrid, as described in Chapter 3. When the `ioctl` instruction is issued in the pipeline, the synchronizing state of the associated control register is set to ’pending’. The low-level driver thread can then perform a read on this control register, at which point its execution will be suspended until a bus message of the appropriate channel arrives or completes. Then, the I/O interface will write to the control register which will wake up and resume the execution of the suspended driver thread. Suspending a thread on an I/O channel this way is similar to installing an interrupt handler; as soon as the event arrives, the context is activated and the thread resumes execution.

### 7.4.6 Memory Interface

All cores in a Microgrid are connected to an on-chip hierarchical distributed cache memory system [CSA22, CSA45]. Data written to memory will migrate through the memory hierarchy to the location at which it is used. All I/O operations, whether lightweight operations through registers or bulk transfers, will eventually read/write their data from/to a buffer in memory. However, such I/O is not bound by the traditional limitations of external memory bandwidth, as we can directly read it into the cache network, unlike conventional existing DMA architectures [72].

The organization of the distributed cache memory system into rings means that not only can I/O be extremely fast, but also extremely parallel. I/O operations can take place in parallel, at different I/O cores, with different clients, using only a local part of the on-chip memory subsystem, avoiding thrashing of the memory hierarchy.
between conflicting operations. This introduces the property of *I/O locality*, where, for the highest performance, the SEP resource manager will have to allocate a client of an I/O service to a place on the same cache level as the associated I/O device's I/O core; i.e. they will share the same L2 cache. A 'smart' placement algorithm would ensure that jobs on Microgrid are created at a place appropriate for the I/O dependencies of that particular job.

### 7.5 Evaluation

In this section we will present an evaluation of the proposed I/O system on the Microgrid with an implementation of the I/O service using the structure described in Section 7.3. We investigate the overhead, throughput and contention of this I/O service implementation using the cycle-accurate Microgrid simulator [CSA29].

#### 7.5.1 Experimental Setup and Implementation

We run our experiments on the default Microgrid configuration, which corresponds with the overview shown in Chapter 3 in Figure 3.4 on page 42. It contains 128 cores that are clocked on 1 GHz, each with a 2 KB I-Cache and 4 KB D-Cache, 32 family entries, 256 thread entries, and 1024 registers. Two cores share an FPU and four cores share a unified 128 KB L2 cache. The L2 caches are grouped into four rings of 8 caches which connect them to the four corresponding directories on the top level ring network, using the core numbering on a single ring as shown in Figure 3.7 on page 45. All memory rings operate at 1 GHz and are 64-bytes (a single cacheline) wide. The top level ring also contains four memory controllers which are configured to meet the DDR3-1600 specification.

We simulate an external ROM device connected to a 1 GHz, 64 byte wide I/O bus connected to core 0 which supports both transferring single register reads or injecting data directly into the memory network. For the latter method we can request the device to read arbitrary sizes as long as they are a multiple of cachelines. To express the relation to DMA accesses, we refer to this access method as DCA, or *Direct Cache Access*.

The implementation of the I/O service consists of the components discussed in Section 7.3, though no protection or memory translation is supported. It supports both the synchronous and asynchronous interface, and employs a zero copy approach where the driver, with register transfers, or the device, using DCA, directly copies the data into the buffer of the client. Also only a single word can be read from the device where the data is passed through shared communication channels instead of through memory. This corresponds with the modes of operation we discussed in Section 7.4.4.

The Completion Handler component shown in Figure 7.1 on page 123 is in this implementation made part of the Dispatch Controller, as it has no other task in this implementation then to signal a synchronizer or to create a callback thread. The Queue Request will queue requests in the same order as they arrive, and every request made to the I/O stack translates to a single request to the device. We developed different low-level driver implementations to support controlling the DCA interface,
reading to a buffer through the register interface and support for direct single word reads through registers. In our experiments, we have placed the I/O service on core 1, sharing the L2 cache with core 0 to which the low-level drivers are delegated to interact with the I/O interface.

### 7.5.2 Overhead and DCA performance

We start our experiments with the aforementioned I/O service implementation and Microgrid setup with determining the latencies of the I/O stack, and then we will determine the performance characteristics of the DCA interface. The result of the initial measurements are shown in Figure 7.3.

<table>
<thead>
<tr>
<th>Access mode</th>
<th>Driver</th>
<th>Synch</th>
<th>Asynch</th>
<th>Callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block DCA</td>
<td>54</td>
<td>665</td>
<td>355</td>
<td>751</td>
</tr>
<tr>
<td>Block Register</td>
<td>23</td>
<td>619</td>
<td>355</td>
<td>661</td>
</tr>
<tr>
<td>Word Register</td>
<td>16</td>
<td>623</td>
<td>340</td>
<td>640</td>
</tr>
</tbody>
</table>

Figure 7.3: Latencies to complete I/O service accesses of the smallest possible size using three different access modes.

The first column of values represents the minimum latencies (in cycles) for communication with the device for a 64-byte DCA read, an 8-byte read to a buffer, and an 8-byte read to a register. These latencies determine the minimum execution time of the low-level driver when interacting with the device. This includes waiting for the device to signal back that the DCA transfer has completed. The second column, Synch, is the time it takes for a client (executing on core 2) to complete a synchronous read operation using one of the three access modes. This includes queuing it in the I/O service, getting it dispatched to the low-level driver, and waiting for the device to complete the request. It should be noted that for reading a single word through the register interface makes the value available right away through the use of shared channels, while a block read to a buffer (using either the DCA or register interface) still requires the data to move from the cache at the I/O core to the cache of the core running the client. This data migration cost is not included in this measurement, and here the client, I/O service and I/O core are all connected to the same L2 cache.

The third column in Figure 7.3 represents the cost for dispatching an asynchronous request to the I/O stack. This includes passing a handle for a callback thread that the Completion Handler will create as soon as the request has been serviced by the Dispatch Controller of the I/O service. However, the dispatch only consists of queueing the request in the I/O service, and then completes. This is why the latency for queuing a block read using either DCA or register transfer is the same, while the single word read through registers is a little bit faster as no address for the buffer needs to be passed. The fourth column represents the latency between dispatching an asynchronous request and when it is returned and the callback thread is started. The callback thread can receive only a single parameter which is either a pointer to the buffer containing the request, or the value in case of the word through register read.
After we have determined the latencies for the smallest possible operations inside, and using, the I/O stack, we now investigate the performance of the DCA interface itself. We only further investigate the block DCA reads, as the block register reads and single word through registers modes will scale linearly with the number of words read, based on the values we saw in Figure 7.3, as well as the efficiency of the low-level driver that transfers the data. We have measured the throughput of DCA based block reads on core 0 locally, directly talking to the device as the low-level driver would. The results are shown in Figure 7.4 for both reading data to a buffer that is cold in the cache as well as a buffer that is warm in the cache, for block sizes ranging from the minimum 64 bytes to 2 MB.

At its peak performance, the DCA interface can achieve almost 50% of its theoretical bandwidth when reading into a buffer that is warm in the cache. As soon as we read larger than the size of the L2 cache (beyond 128 KB), we see that the throughput drops to around 8 GB/s. For cold caches we see the same behavior but also for smaller read sizes, which is expected as the cache will fetch the cachelines before writing to them and they have to come from main memory. We have yet to understand where the peak value at a 16 KB block read to a cold cache comes from.

### 7.5.3 Throughput of I/O requests

For our first performance measurements of I/O requests we have a client program that sends synchronous requests for a block DCA read of different sizes ranging from a single 64 byte cacheline to 2 MB. For each size we make an average over 10 measurements, and for each individual measurement a new buffer is allocated at a unique memory address so that it is not yet present in the cache. After the synchronous request completes, the client program reads a single byte from each cacheline to pull the data to its local L2 cache. The time to move the data to the local core so that it can be used is part of the measurement interval. We then placed the client on core 3
on the same L2 cache, on core 8 which is on the same L2 cache ring, and on core 32 which is on the next ring. We could not measure a difference in performance between a client on the second, third or fourth ring, which makes sense as cache request/responses will always make a full round-trip over the top level ring. The results of this experiment are shown in Figure 7.5.

![Graph showing throughput of block DCA reads through the I/O system using different placements for the client, showing locality effects.](image)

Figure 7.5: Throughput of block DCA reads through the I/O system using different placements for the client, showing locality effects.

In our first experiment we clearly see the result of a locality effect on the performance. As the client only reads in the requested data cacheline by cacheline, the resulting throughput is dominated by the latency to fetch each individual line. This explains the difference between the three different placements in the results that we show in Figure 7.5. The throughput is not limited by the rate at which DCA injects data into the memory system (Figure 7.4), as this is the same regardless of the placement of the client. Unexpectedly, we observe that the throughput from accessing the cache on an other ring is slower than reading from main memory, which is caused by a currently unknown and unresolved issue with the Microgrid memory system. We also observe in this experiment that the throughput achieved, even on the local L2 cache, is much lower than the speed at which the DCA interface delivers the data. This is caused by the client being a single thread that accesses the data cacheline by cacheline, which is not the typical use case scenario on the Microgrid. In general, it is safe to assume that after reading a large chunk of data, an application on the Microgrid would spawn a lot of concurrent threads to perform an operation on the data that was read. This allows more concurrent requests for cachelines, which hides some of the latency, which we show in our next experiment.

The results of our second experiment are shown in Figure 7.6. Here, we replaced the client with a client that creates a family of threads on the local core where each thread takes care of reading a single cacheline of the buffer after the synchronous I/O request returned. At a request size of 16 KB, this results in a family of 256 threads which all concurrently issue a single cacheline request. This simulates the same access
pattern as a concurrent operation on the retrieved data, and allows for a much greater throughput as cacheline requests are pipelined in the memory system.

Please note the different scale on the vertical axis compared to Figure 7.5; local throughput on the same L2 cache is increased from a maximum of 1.6 GB/s to 5.0 GB/s. Also the difference in performance between the different placements of the client is greatly reduced, as the pipelining of many concurrent requests for cachelines hides the individual access latency. We can conclude that a single Microgrid core cannot match the bandwidth offered by the block DCA transfers, even if we only consider DCA transfers to a cold region in the cache.

Using the Register Interface

Our second set of throughput experiments focus on using the register transfer interface to the device, where we investigate synchronous request for both a block buffered register access and the direct word register transfers. The difference between these two access modes is that in the block register access the low level driver will perform multiple register based reads from the device to satisfy the requested block length and place the result in a buffer. The direct word register transfer does not use a memory buffer but uses the shared communication channel to pass the value back through shared registers from the low-level driver through the Dispatch Controller to the requesting client.

In Figure 7.7 we see the results of an experiment similar to those of Figure 7.5 and Figure 7.6, but which now uses the block register read mode. While there is a small locality effect that can be observed, we should note that the overall throughput of the low-level driver reading the data word by word into the buffer is more than an order of magnitude lower than DCA transfers. When we use a client with a lot of microthreads to read from the buffer (not shown in a figure), the difference between the three placements is further reduced and the maximum throughput tops out at
116 MB/s. This is not a significant improvement over the 111 MB/s maximum in Figure 7.7, which makes clear that copying the data word by word in the low-level driver is the bottleneck here. However, it has the advantage that it can be used to copy small blocks of data that are smaller than a single cacheline.

When we use the word register interface where we pass the data through shared channels instead of memory, we can only transfer 8 bytes for each request to the I/O service. Transferring larger blocks of data this way requires the client to send multiple requests, which then scales linearly with the number of requests sent, based on the minimum value that we observed earlier in Figure 7.3. Also in this experiment we could not observe a locality effect, as only global and shared channel communication is made between the client core and the I/O service core. The current simulation of the Microgrid does not yet have an accurate modeling of the delegation and register access networks, so we can not measure a difference in latency induced by the distance between cores on these networks. However, we assume that the difference will be minimal compared to the effects we saw in our DCA based experiment where many cachelines had to be moved around. For each access the measured latency corresponds with a throughput of 11.8 MB/s.

We compare the latencies for small read sizes using the three different access modes in Figure 7.8, where the vertical axis is a log scale of the number of core cycles it takes to complete the (synchronous) I/O request. We see that the word register transfer mode indeed scales up linearly as expected, and is a little bit faster on average than the block register mode for a single word read. Another important observation that we can make in this comparison is that as soon as you want to transfer at least a single cacheline, the DCA mode already delivers the best performance.
### Contention of I/O requests

In this section we discuss a set of experiments which attempt to measure how the I/O service can cope with contention. As an initial experiment we start a number of clients spread across the whole Microgrid except the first four cores on an L2 cache where the I/O service runs. Each client then reads 1 KB using the word register access mode, therefore making 128 requests to the I/O service. This is then repeated 10 times and the average and minimum and maximum request response times are measured.

The results are shown in Figure 7.9, where we divide the average, minimum and maximum through the number of clients as this is expected to grow linearly with the number of clients, simply because accesses to the device are serialized. However, when
7.5. EVALUATION

there would be additional inefficiency in handling the contention, we would see this divided average increase with the number of clients. The minimum and maximum values are computed over all measurements over all clients, but their deviation from the average is negligible and therefore they are not visible in this graph. We also plot the corresponding aggregated throughput (note it is expressed in KB/s in this graph to make it visible) which can be computed from this average.

We can not observe any contention using this setup, as the measurements result in an average latency that is proportional to the number of simultaneous clients as shown in Figure 7.9. When we look carefully, we see that the throughput increases when going from 1 to 2 clients before it stays steady. This can be explained by a pipelining effect. While the request of one client is handled by the Dispatch Controller and low-level driver, a second incoming request can be queued at the same time. As the Dispatch Controller can then continuously process requests, we observe an aggregated throughput that is twice the throughput we measured before.

It should be noted that there are two reasons why we can not observe contention with this many requests being concurrently submitted to the I/O service. First of all, as the delegation network is not simulated in detail we do not observe contention on communication with the I/O service. However, as these delegation messages are relatively small and of bounded size, a proper delegation network design would not impact contention heavily.

The second reason is the extremely latency tolerant nature of the Microgrid and the support for many execution contexts in a single core. As multiple requests are handled inside the I/O service on core 1, multiple requests can be set up for the queue at the same time while the pipeline of the core executes efficiently switching between working on multiple requests when one is stalled for a memory access. These requests are then only serialized at the point in the Queue Request component where they are actually put into the queue using an exclusive place. At the same time the Dispatcher continues its processing which only needs to access the exclusive place to pop the next request from the queue. The property that all execution contexts remain in the core at the same time and do not need to be spilled to external memory with context switches makes this possible. When the contexts in the core are full, this is back-propagated through the delegation network, delaying the context on the requesting side which does not affect the performance of the I/O service as it runs on its own core.

Contention with DCA

As a second part in our contention experiments we experiment with DCA transfer mode accesses as these could result in contention on the memory network. We have seen that the other two transfer modes do either not depend on the memory network at all or do not consume sufficient bandwidth to be able to observe contention imposed by the memory network.

In our first DCA based experiment we again run 10 measurements of reading 16K through in block DCA transfer mode using a different number of clients spread over the Microgrid, where each client is assigned its own core. However, in contrast with the previous contention experiment, only a single request needs to be sent for
the 16K transfer, so any contention we observe can only be caused by the memory system. Again, we similarly collect minimum, maximum and average latencies and compute the aggregated throughput. The result is shown in Figure 7.10.

In this experiment we can observe some variation between the minimum and maximum latency, which is to be expected as some clients will run on cores on the same memory ring as the I/O service and driver, and some will run on other rings with the behavior as we saw before in Figure 7.5. We use the same method for consuming the data at the client. Here, we see the same pipelining effect as with our previous contention experiment. While single 16K DCA mode requests could only achieve 1.5 GB/s in our earlier experiment (Figure 7.5), the system delivers an aggregate throughput of 5.2 GB/s which corresponds with the values we saw in Figure 7.6. There is a small peak in performance at 11 clients at almost 5.4 GB/s, but it is not clear why this is slightly higher. This could be a very mild contention effect on the memory system.

We repeated the last experiment but reduced the block size to a single cacheline. The result of this is shown in Figure 7.11. We still did not observe any contention, neither on the accesses to the I/O service nor on the memory network. However, as our clients are again spread across the whole Microgrid, we see more fluctuation in the minimum and maximum latencies. At about 70 clients and more this fluctuation becomes larger, but we suspect that this is caused by the number of clients that are no longer running synchronized as the read operations are all very short. The aggregated bandwidth at around 140 MB/s is again better than the single request bandwidth for 64-byte requests in Figure 7.5, due to the request pipelining effect. However, it is not close to the theoretical peak of 64 byte DCA requests as shown in Figure 7.4, as there is still the overhead that the Dispatch Controller needs to fetch a new request from the queue between the transfers.
Figure 7.11: Concurrent clients spread over a whole Microgrid, reading a single cache-line of 64 bytes using DCA transfers. We observe no contention, but more varying behavior.

As a final experiment and attempt to produce a contention effect we increase the block size to read 1 KB per request. However, we use the microthreaded approach to consume the data returned by the request to increase the number of concurrent messages on the memory network. The result of this experiment is shown in Figure 7.12, where we can observe a very slight contention effect.

In contrast to our other contention experiments, in this experiment we indeed see the aggregated throughput decrease when more then 3 clients access the service at the same time. The average latency for a request normalized to the number of requests will also slightly increase when more clients are attempting to access the service. This contention is caused by the number of cacheline requests that are concurrently sent to the memory system, as every client will issue up to 16 requests at the same time. However, while this data is produced by the I/O service, it is in a sense a benchmark showing properties of the Microgrid memory system.

7.5.5 Discussion

The experiments that we have performed with our I/O service implementation of the Microgrid has shown us some interesting properties. First of all, due to the latency hiding effect and the cheap scheduling and suspension of threads, the I/O service can work very efficiently with a large number of clients. This gives us confidence that also the other suggested operating system services can be made scalable.

One of the effects that was less obvious than we had anticipated was the effect of locality; as the Microgrid can efficiently hide latency, this mitigates this effect. However, we have only run experiments where only our I/O benchmark was running in the system. In a real user scenario making large I/O requests to the other side of the chip will consume bandwidth on the top level memory network and two lower level
Figure 7.12: Concurrent clients spread over a whole Microgrid, reading 1 KB using DCA transfers, consuming the data with multiple threads. Some slight contention can be observed.

rings, which might impact performance of other applications. Therefore we would still suggest using smart placement of I/O intensive application components.
CHAPTER 8

Run-time system

8.1 Introduction

We have shown in the previous chapters which components are required to make up the basis of an operating system on an SVP platform. However, we have not yet covered how we would write applications for such a system. As SVP platforms such as the Microgrid offer support for a massive amount of concurrency, it is clear that most programmers would not want to express this all explicitly at the programming level, for example by using $\mu$TC. From the perspective of programming productivity there is a need for higher level languages and run-time systems that take care of this, together with the resource management that we discussed in Chapter 6. Multiple languages and tools have been developed or modified to target SVP based systems. This includes a parallelizing C compiler [CSA39, CSA40], a compiler and run-time for the functional array language SAC [CSA13, CSA10, CSA41, CSA15], and the Graphwalker [CSA25] run-time for the S-Net coordination language [CSA14, CSA35] which was specifically designed to exploit the amount of concurrency offered by the Microgrid.

In this chapter we propose a novel alternative design for an S-Net run-time for SVP systems, inspired by the Graphwalker design, that we named the Resource Walker. The major difference with the Graphwalker is that it tries to achieve a balanced resource allocation, and is tailored to use the resource management systems described in Chapter 6 to employ a space-sharing strategy. Before introducing our approach, we first introduce some more details on S-Net in Section 8.2 including details on existing and proposed run-times such as the reference implementation and the Graphwalker. We discuss our proposed Resource Walker run-time in Section 8.3, including details of a proof-of-concept implementation that we built and which in fact drove the experiment with the resource-oriented resource management of Chapter 6. We conclude this chapter with a discussion of this approach.
CHAPTER 8. RUN-TIME SYSTEM

8.2 Introduction to S-Net

S-Net is a declarative coordination language [CSA14, CSA35] that aims to ease the programming [CSA33, CSA34] of multi- and many-core systems by separating the concerns of expressing computation and managing concurrency. It is only concerned with coordination and therefore is not suitable to express computation; it can be used to specify stream processing networks of asynchronous components that are written in conventional languages. These networks can be expressed in algebraic form using four different network combinators. Data flows over these networks in the form of records, which consist of a set of named fields represented as opaque label/value pairs, and can contain special tags that can be interpreted by the network.

Each component in the network has a type signature which specifies which fields it accepts as input, and which fields it will produce in records on its output stream. The type system is used to determine the routing of records in the network, so that a record is delivered down stream to the next component that has a matching type signature. The type system supports record sub-typing where the type signature of a component that matches one or more fields in a record also matches records which contain additional fields. Flow inheritance is used when such a record is consumed by a component; the additional fields that are not matched are appended to every resulting record that appears at the output of that component.

A component can either be one of the two built-in kinds, a filter box or synchrocell, or a box which is written in an external language and can perform computation. Filter boxes and synchrocells take care of housekeeping tasks with lightweight modifications of records on streams; a filter box can for example discard or duplicate fields in a record, or add tags to it, while a synchrocell is the only available method of synchronization [CSA9]. A synchrocell captures two or more records according to its signature, and combines them into a single output record. A box is a stateless component which consumes a single record on its input stream, and in return produces zero or more records on its output. This stateless property implies that multiple instances of a box can execute independently on different input records, and that boxes can not act as synchronizers by consuming multiple records.

Boxes are implemented using an external box language, which currently can be either standard C or SAC [CSA13]. Traditionally, a box is seen as a sequential piece of computation, but using the aforementioned tools to transform and run C or SAC code for SVP systems we also assume we can exploit fine grained concurrency within a box, and that they can be expressed in an SVP native language such as $\mu$TC. A box is implemented as a function in the specific language that receives the fields of a record as its arguments according to its signature, and that during its execution can call an output function to emit a new record. Another key aspect of this isolation of computation and both communication and concurrency management is the reusability of such components, and not surprisingly, this idea closely matches our SVP components from Chapter 3, and the service oriented resource management from Chapter 6.

Networks are composed of components strung together with network combinators. Intuitively, there is a serial composition (\ldots) and a parallel composition (\mid). The other two combinators are the complementing serial replication (\ast) and parallel replication.
8.2. **INTRODUCTION TO S-NET**

The serial replication combinator \( A^* \text{type} \) constructs an infinite chain of replicas of \( A \) connected in serial. Only records that match the specified type signature can exit the infinite chain. Parallel replication expressed as \( A!<\text{tag}> \) replicates network \( A \) using parallel composition, where the value of the specific \( <\text{tag}> \) in a record is used to determine to which replica the record is forwarded.

The replication and parallel combinators can introduce non-determinism in the ordering of records. For example, a record taking a different branch could overtake the previous record, or one record can exit within less iterations of the serial replication compared to the previous record. Therefore, deterministic variants of these three combinators (\( |\mid \), \( \times \) and \( !! \)) exist that preserve the stream order. For more details on the semantics of these combinators and the type system of S-Net we refer to the official documentation [CSA35].

### 8.2.1 S-Net Example

In order to give a concrete example of what an S-Net definition of an application would look like, we apply it to our example application of Chapter 6, which was shown in Figure 6.5 on page 103. The corresponding S-Net definition is shown in in Figure 8.1, and we will now discuss in some more detail what it means. It should be noted that this is a very limited and simple example, and for more complex examples using synchrocells and the other network combinators we refer to the S-Net documentation [CSA35].

```plaintext
1 net example ({a} -> {x} | {y}) {  
2   box BoxA ((a) -> (b) | (c));  
3   box BoxB ((b) -> (x));  
4   box BoxC ((c) -> (y));  
5 }  
6 connect BoxA ... (BoxB || BoxC);
```

Figure 8.1: Corresponding S-Net definition of our streaming example application

The network consists of three box components, that we named BoxA, BoxB and BoxC. The box signature on line 2 tells us that BoxA accepts records which contain field \( a \) and then emits one or more records with either a field \( b \) or field \( c \). Similarly, BoxB accepts records which contain field \( b \) and produces one or more records with field \( x \), and BoxC accepts field \( c \) and produces field \( y \). Then, the connect definition tells us how they are connected; BoxB and BoxC are composed in parallel, which are together composed serially after BoxA, just as our example application was defined. This connection results in a type signature for this network, which is shown on line 1. This network accepts records with field \( a \) and produces records with field \( x \) or field \( y \).

The fields inside a record emitted by BoxA after computing on an input record determines if it is then routed to BoxB or BoxC for processing. If BoxA would only emit one kind of record (for example, \( (a) -> (k) \)) and BoxB and BoxC would both accept this field in their signature \( (k) -> (x) \) and \( (k) -> (y) \) respectively, the
choice for the output of BoxA between BoxB and BoxC becomes non-deterministic from the perspective of the network specification, and can be decided by the run-time system. However, as the deterministic parallel composition (||) is used, it is still guaranteed that the interleaving of records with fields $x$ or $y$ on the output of the network is in the same order as the records with field $a$ or $k$ that went through BoxA.

8.2.2 Existing S-Net implementations

The reference implementation of the S-Net run-time [CSA12] resembles a Communicating Sequential Processes [79] (CSP) approach where a box in the network is the unit of concurrency, executing in its own thread, connected to other components with buffered streams. The behavior of this approach is not fully compatible with the definition of CSP as the network can dynamically expand, but it is similar during its execution phase. Therefore we refer to it as the CSP style implementation. In contrast to the CSP style approach, as boxes do not maintain state, an implementation can be allowed to run multiple instances of a box concurrently, each processing records independently, allowing for more concurrency. To develop an SVP based S-Net implementation for the Microgrid which supports a high degree of concurrency, the Graphwalker [CSA25] approach was proposed.

In the Graphwalker approach, the records flowing through the network are the unit of concurrency and are scheduled using dynamic data-flow principles. Records that need to be processed are kept on a bulletin board which holds information on where they are in the network. The active agents, Graphwalkers, take a record from the bulletin board, infer where the record is in the network and which processing is required next and invokes an instance of the component to consume this record. This box instance will then emit zero or more new records that are posted back on the bulletin board, though it can be optimized by making sure that the Graphwalker instance continues with one of the records that its previous invocation produced. The Graphwalker approach is quite similar to Scoreboarding [146] which was used to schedule instructions in early out of order processors, though a record can never have any dependencies.

Effectively, this means that as long as we have enough resources, we can have an active agent, or thread, for every record that exists in the system. This potentially exploits much more concurrency than in the CSP style implementation as it is only limited by the current number of records present in the network, and not by the number of components in the network. However, the output of the network and the synchrocells, the only stateful component in the network, need to observe records in stream order. Therefore, at some points in the network a re-ordering of records is required for which an indexing scheme has been described in [CSA43].

An issue with the CSP style reference implementation is that it does not fit naturally to the concurrency patterns of SVP, though it is clear to us that an implementation can be made using the synchronization and communication methods described in Chapter 4 and Chapter 5. However, it is not very suitable for our dynamic resource management of Chapter 6 as components are statically assigned to threads. In contrast, the Graphwalker approach can dynamically request and release resources for each instance, but this can put a severe pressure on the SEP and introduce a
lot of unnecessary overhead. This would be especially the case in a system where not all resources are suitable for general purpose computation. On a platform with run-time reconfigurable resources such as presented in the Service-oriented resource management approach, the Graphwalker has no incentive to reuse a resource for the same operation besides the cost mechanism in the SEP protocol. The lack of a global resource strategy in the Graphwalker would cause such a system to reconfigure unnecessarily often, with the associated reconfiguration overhead.

Another issue with the Graphwalker, as it was proposed in [CSA25], is that it does not attempt to achieve any fairness in which records get priority in processing. This means it can not guarantee progress or guarantee to produce any output as long as there is input records available on the network. Considering our application example, it is perfectly valid for the Graphwalker to only instantiate instances of BoxA, as there is no monitoring of overall progress. The CSP style implementation does not have this problem assuming a fairness in scheduling of the threads that implement the components of the network, but it does suffer from load imbalance problems. The design that we will propose in the next section tries to find a strategy in the middle between these two extremes of the implementations that we have just described. It tries to keep the dynamic and massively concurrent approach of the Graphwalker to cope with load imbalance, while being more resource aware and attempting to achieve fairness similar to the CSP style implementation.

8.3 Resource Walker

We named our approach for an S-Net run-time the Resource Walker to emphasize the dynamic use of resources and the similarities to the Graphwalker. We take the notion of buffered streams from the CSP style reference implementation [CSA12], but dynamically start instances of the boxes to process these streams similar to the Graphwalker [CSA25]. To follow up on the Scoreboarding analogy of out of order processors that we used in our description of the Graphwalker, we have buffered streams that are similar to the reservation stations from Tomasulo’s algorithm [148]; they queue up records that need to be processed by an instance of a box. In that sense, it is a decentralized form of the bulletin board used by the Graphwalker, divided over all boxes in the network.

The Resource Walker is similar to the Graphwalker, as it is an agent that takes an input record from its assigned queue and executes an instance of the corresponding box to consume the record. After the box completes, it takes another input record for the same box, and so on. Whenever the box produces an output record, it will call a routing function, similar as in the Graphwalker, to determine the destination for this record and put it in the appropriate queue. A single instance of the Resource Walker is responsible for only one queue and one instance of the corresponding box, and it has its own set of resources assigned to execute these instances on.

Having these queues gives us information of the state of the network and a handle for load balancing. If the queue in front of a certain box becomes very large or grows rapidly, this is an indication that an adaptation needs to be made to improve the throughput of this component. We do this by allocating additional resources
to execute more instances of the component, so that it can process records from this queue concurrently. This results in the same stream ordering issues that the Graphwalker has, but we can use the same indexing scheme \cite{CSA43} to reorder the stream whenever required. Conversely, when a queue is often found empty, this can indicate that too many resources are allocated to the corresponding component.

We introduce a second entity executing in our run-time named the Observer, which makes resource management decisions and corresponding requests. It periodically inspects the state of the queues and decides if more resources need to get allocated for a box in the network. After successfully acquiring resources from the system’s resource manager, the SEP in our case, it starts an instance of a Resource Walker on those resources. What a good general adaptation algorithm would be is left as interesting future work, so far we have only experimented with the simple adaptation algorithm described in Chapter 6, with the corresponding results. Another important part of this algorithm is the decision when resources should be released again, which is generally done when a Resource Walker starts encountering an empty queue. This way, the over provisioned resources allocated to a box are released. However, as we have seen in our experiments in Chapter 6, it is desirable not to give up resources too soon.

![Figure 8.2: Schematic overview of the Resource Walker S-Net run-time](image)

An overview of the Resource Walker run-time and its processes is shown in Figure 8.2. On the left it shows the Observer which inspects the state of the queues and that negotiates resource requests with the SEP. It spawns additional Resource Walker instances (RW) on the acquired resources. The Resource Walkers take records from the queues and instantiate the box function on them, which then can output new records by calling the Routing function. The Routing function has a notion of the topology of the network and determines the destination for the record, moving it to the appropriate queue.
8.3. RESOURCE WALKER

8.3.1 Implementation

We should make clear that we have not developed a full implementation of a complete S-Net run-time system which contains all of the mechanisms that we have just described. However, many components of the Graphwalker approach can be re-used such as the routing and re-ordering mechanisms, and box language interfaces. Therefore, we ignore these and focus on investigating how the run-time structure of the Observer, queues and Resource Walkers can be implemented on an SVP system. The proof-of-concept run-time implementation that we built statically implements the topology of the example application from Chapter 6 (Figure 6.5 on page 103). Another goal of our proof-of-concept implementation was to investigate if we could build an S-Net run-time on SVP without requiring the use of the detached form of create, which is heavily used by the Graphwalker to start concurrent continuations.

The implementation of the queues in the network is trivial. Each queue has a data-structure that is guarded by its own exclusive place, to which update functions are delegated to add or retrieve a record from a queue. An input process takes care of reading the input of the S-net program and inserting this as records into the network using the Routing function. In our specific case this is a family of threads that periodically generates a record to simulate it appearing on the input stream and then adds it to the queue for BoxA.

The Observer is implemented as a family of threads, where each thread is responsible for a single Observer iteration. The indexing parameters of the family are set to create an infinite sequence of threads, but the actual number of threads that can exist at the same time are either bounded by a block size parameter, or the number of thread contexts supported by the SVP implementation. The number of Observer threads that can exist at the same time limits the number of resources the Observer can manage in its current design, as we will now show.

A thread in our Observer implementation consists of two parts, the first part observes the system state and makes the decision if it is required to request more resources, while the second part executes this decision. The first part of the thread is guarded by a token passed through a shared channel so that only one thread in the Observer family takes this action at the same time. Also this part incorporates taking care of the delay needed to periodically trigger the inspection of the system state. We show the corresponding pseudo $\mu$TC code in Figure 8.3.

The inspect_state() step takes care of querying the state of each queue. It can be implemented as a set of concurrent create actions to all exclusive places of the queues to read out the current sizes of the queues. As this is a read-only action, this might not always require a delegation to an exclusive place, for example on SVP implementations with sufficiently coherent shared memory the value can possibly be read directly. Reading the exact value at a certain point in time is not important, the Observer only requires regular snapshots to observe trends in the flow of records. Therefore, it also saves a copy of the size of the queues for the next iteration so that we can determine the changes in queue sizes.

Using the information that has been gathered, the make_decision() step executes an algorithm to decide if and where resources should be added in the system. This algorithm depends also on how resources are managed in the system, i.e. the resource-
thread observer(shared int token)
  int tmp_token = token;
  delay();

  inspect_state();
  make_decision();
  token = tmp_token;

  if (adaptation_required)
    request_resource(resource);
    create(fid;resource) resource_walker(target);
    sync(fid);
    release_resource(resource);

Figure 8.3: Pseudo µTC code for the Observer entity of the Resource Walker implementation

oriented or service-oriented resource management of Chapter 6. When a number of
general-purpose resources can be requested, deciding how many should be requested
is also a task of this algorithm. For our proof-of-concept we used the algorithm that
was outlined in Section 6.2.3.

At this point, the Observer has made a decision if more resources need to be
requested, what kind, and how much. It then passes on the token to the next in-
stance which continues the periodical monitoring, while this instance further handles
the required adaptation if needed. Assuming that it decided to make an adaptation,
it proceeds to the request_resource() step, which will consist of a create of an
SEP instance to get the requested number and/or type of resources allocated. When
resources are successfully allocated, the Observer starts an instance of the Resource
Walker on these resources. It tells it which S-Net Box component it should exe-
cute, and the location of its associated queue. This Observer thread then suspends
until the Resource Walker terminates, in order to release the resources again in the
release_resource() step which consists of another create to the SEP.

An example implementation of the Resource Walker is shown in Figure 8.4. First
it takes a record from the queue at get_record(queue), which will consist of a del-egation to the exclusive place of the queue to safely retrieve the record. Then, if a
record has been successfully retrieved, it invokes the associated box function on the
retrieved record. It does not need to take care of the output of the box function, as
this is handled by the box function itself which calls an snet_out function internally
to emit a record, which is then handled by the Router component. When fetching
a new record from the queue has failed, it executes test_abort_walker() which is
the part of the algorithm that determines if the resource should be released again. If
this is the case, it will terminate this instance of the Resource Walker and the parent
Observer thread will take care of releasing the resources as we saw in Figure 8.3.
8.4 DISCUSSION

In principle, the Resource Walker would execute on the same resource as the box function, but this is only possible if the resource is a general-purpose computing resource, capable of actually executing it. When boxes are executed on specialized resources such as with the service-oriented resource management approach of Chapter 6, the Resource Walker will have to run elsewhere and also receive the place to execute the box as a parameter. The create of the box function will then be a delegation to this place.

An alternative for keeping an Observer thread around for each existing Resource Walker is to have the Resource Walker release its resource itself after it decides to terminate. If this is possible depends on the SEP implementation, as the release request would originate from a different place than the allocate request, possibly even the same place as the one that is requested for release. However, with this approach, the Observer does not have to be an infinite family of threads but can use a simple loop and use a detached create to start Resource Walker instances, requiring much less thread context resources.

8.4 Discussion

We assert that our Resource Walker approach is fully implementable, assuming that the Graphwalker can be implemented as it was specified. While an implementation of the Graphwalker on SVP has been constructed\(^1\), there is no available documentation about the details of this implementation. Besides the parts that we borrowed from the Graphwalker, we showed and discussed how the other parts of the Resource Walker can be implemented. One of the most tricky parts to get right is the implementation of the synchrocell which would be part of the Router component. This requires a matching store, which can be implemented with critical sections using an exclusive place.

An interesting part of future work will be investigating the possible adaptation algorithms for the Resource Walker run-time. The Observer component could keep more historical data and employ more complex heuristics than the simple algorithm

\(^1\)K. Bousias, University of Amsterdam, personal communications, 2009

```
thread resourcewalker(queue_t queue)
    while (true)
        record = get_record(queue);
        if (!record)
            test_abort_walker();
        else
            create(fid) queue->boxfunc(record);
            sync(fid);
```

Figure 8.4: Pseudo µ/TC code for a Resource Walker
that we used in our Chapter 6 experiments. Also the Observer could attempt to make larger adaptation steps for faster convergence, perhaps starting multiple Resource Walker instances for different network components in the same iteration. Ideally, the Observer forms a model of the behavior of the network using its measurements, making it a truly self-adaptive system. However, it is also interesting to observe that in our experiment of Chapter 6, our very simple algorithm already managed to reach a semi-stable state with the dynamic resource requirements of the application.

There are some implicit assumptions in the approach of the Resource Walker. For example, it assumes that a box performs sufficient computation compared to the communication cost of the records, as data is moved to the computation and not vice versa. This is the assumption in most S-Net implementations, even the implementation on distributed memory [CSA11] or that on the Intel SCC [MWvT15], though these implementations also support on demand fetching of data so that only the fields of a record that are required for the box computation are fetched. When the cost of computation is smaller than the communication cost, it can be generally argued that the granularity of the box functions has become too small and that parts of the network should be aggregated into larger boxes of computation. On the other hand, the Graphwalker approach would be able to easily move computation to the data.

Another assumption that we made in our approach is that the box function can exploit concurrency and can fully utilize the resource it has been assigned, or even gain performance from running on place of multiple cores in a Microgrid setting. It is important that the Observer component is aware of the properties of a box function, such as how much concurrency it can exploit. The Observer should not request a multi-core place for a box function that can only execute sequentially or exposes very little concurrency.

One issue with the Resource Walker approach that should not be left un-discussed is scalability. Though we have discussed before how we can scale the SEP resource manager, and the organization of the queues and their processing is distributed, the main central component that will likely not scale very well in the way we have currently implemented it is the Observer. There are two related issues; first of all, on larger networks with many components, it will take an increasing amount of time to check the state of all queues, and to determine where to make adaptations. The second problem is to orchestrate all the adaptations and resource requests, and starting of new Resource Walkers, though this can partially be made concurrent. As for the first problem, one solution that we suggest is to split the network into sub-networks which each have their individual Observer. There might even be room for a hierarchical solution as with the SEP resource manager; higher level Observers that monitor the throughput of networks controlled by the lower level Observers and that make coarser grained resource decisions.
CHAPTER 9

Conclusion

9.1 Overview & Discussion

In this thesis we have investigated strategies and mechanisms to construct a general-purpose operating system for the Microgrid many-core processor. In this process we used SVP, the programming and concurrency model for the Microgrid (Chapter 3), as a general abstraction which allows the ideas and patterns expressed in this thesis to be re-used for other future many-core platforms based on SVP or similar concurrency models. In this process we have attempted to find the least intrusive way to construct operating system functionality based on SVP or the Microgrid, so that the efficient fine grained concurrency support of the original design can be preserved.

In order to claim that we can build a general purpose operating system for the Microgrid we make an analysis based on the following five required core functionalities that we should be able to implement:

- Process handling, synchronization and scheduling (Chapters 4 and 6)
- Protection (on both memory and resources) (Chapter 5)
- Address space and memory management (Chapter 5)
- IPC (Chapters 4 and 5)
- I/O (Chapter 7)

For each of these topics we have shown either solutions based on SVP or how the model or Microgrid design can be extended to support them without significant changes to the programming model. In terms of structure, it is clear that a distributed operating system design is best suited to scale to many cores, and this also suits the patterns of SVP very well. We suggest a distributed design based on services that are located on dedicated cores, which reduces the interference with applications, also as the Microgrid hardware does not require the presence of a kernel with a scheduler on every individual core.
9.1.1 Processes

In order to have efficient low level hardware concurrency management, the distinction between processes should be made at a higher level than the individual hardware managed threads. In Chapter 4 we proposed to do this by introducing process boundaries on the basis of resources, where we view a process as a collection of allocated processing resources, i.e. a set of places. The implication of this definition on scheduling is twofold; fine grained scheduling of threads within a process are left up to the SVP implementation, while coarse grained scheduling of processes becomes a resource allocation problem. As the amount of concurrency an application exposes can vary over time, we introduced a dynamic resource allocation mechanism (Chapters 4 and 6) where resources can be requested and released on demand. This mechanism then forms the basis for our coarse grained scheduling by controlling how resources are assigned.

9.1.2 Scheduling

It is clear that on a platform with many computing resources and multiple applications a space-sharing approach to scheduling should be used where multiple applications are scheduled at the same time on different resources. However, the challenge lies in how to deal with a situation in which the applications request more resources than the system has available. The classic approach is to pre-empt execution and to time-share the resources which has its inefficiencies due the cost of context switches. Such time-sharing can be trivially implemented in our SVP based systems with an approach similar to Tessellation [104] where we time-share places, as long as we can pre-empt and resume the execution on these resources. However, it was part of the strategy of the original Microgrid design to not support pre-emption in order to keep the design of the individual cores as simple as possible. We discussed the complexity of implementing pre-emption in Chapter 4 in Section 4.2.4 and Section 4.2.5, after which we investigated alternative approaches using pure space-sharing.

An important component in our proposed space-sharing strategies is the resource manager which facilitates the dynamic resource management for which two approaches were presented in Chapter 6. The policies it uses to hand out resources is of great importance to maintain fair resource usage between applications making sure that a single application can not monopolize all computing resources. We have outlined several possible approaches for this in Section 4.2.6; first of all, the resource manager can always allocate less resources and force the application to throttle its amount of exposed concurrency. Secondly, one possible policy is to only allocate a set of resources for a specified period of time, or a limited number of invocations of a component to guarantee that the resource becomes available again. Thirdly, we suggested that application run-times should support an upcall mechanism to revoke resources, only resorting to killing applications in the worst scenario.

We have demonstrated in Chapter 6 in Section 6.2.3 how a simple space-sharing approach with well behaving applications that dynamically allocate and release resources can work well. However, further research and evaluation is required to determine the best policies for the space-sharing strategies outlined in this thesis using realistic application scenarios. Additionally, the effect of implementing pre-emption
in the Microgrid should be investigated in terms of architectural complexity and its
effect on performance, as to determine the trade-off between the pure space-sharing
and hybrid space and time-sharing approaches. Only then we can make an educated
decision which approach is best suited for a Microgrid operating system.

9.1.3 Protection & Memory management

In a system with hardware managed concurrency where an application can start new
threads on a resource using hardware mechanisms, we require a protection mechanism
to constrain this to the resources that have been allocated to the application. We
showed in Chapter 5 how this can be achieved by adding an access key to the identifiers
which effectively turns them into capabilities. As the same mechanism is used to
delegate execution to a system controlled resource for accessing system services, this
special case required a more rigorous form of protection to also be able to control
which code entry points could be accessed. For this we proposed a second capability
based mechanism in the form of a hardware supported table in which such system
entry points are registered, where only a delegation to a system resource can be made
using one of the entries registered in the table which can be configured for each process
individually.

We have also addressed the issues of memory protection, address space translation
and memory management in Chapter 5. We defined a memory protection scheme
based on execution resources (places), and discussed the advantages and disadvantages
of single address space approaches which seem to most naturally fit the SVP model
and the Microgrid design. Here, we introduced the Protection Domain Manager as
a system service that can configure the protection for a resource, and the Memory
Allocation Manager service which can allocate large pieces of memory to facilitate
stacks and heaps for processes.

9.1.4 Synchronizations & IPC

In order to support IPC in SVP based systems we had to tackle two challenges, first
the lack of arbitrary synchronization support (Chapter 4, Section 4.3), and the issues
with the weak memory consistency model (Chapter 5, Section 5.3.4). In Chapter 4
we presented solutions based on the use of exclusive places with different resource
trade-offs. More specifically, we could even omit the use of an exclusive place using
a property specific to the Microgrid architecture. Similarly, we showed in Chapter 5
using different approaches how we can guarantee that data communicated through
IPC is accessible to the receiver due to the consistency model. As the current im-
plementation of the memory system of the Microgrid has a more strict consistency
than mandated by the SVP model, the most efficient IPC method presented is actu-
ally feasible on the Microgrid, while it is not on an SVP system that implements the
original weak memory consistency model.

9.1.5 Resource Management & Run-times

Besides memory, we also need to manage our computational resources which we do
with the SEP resource management service described in Chapter 6. We proposed two
different approaches to resource management, one based on requesting a number of cores for a (relatively) homogeneous architecture, and one based on requesting specific functionality in a heterogeneous system. Both approaches have been demonstrated with experiments in Section 6.2.3 and Section 6.3.3. In Chapter 8 we further proposed a run-time system for the S-Net coordination language that is able to exploit both these resource management approaches. This run-time also gives insight in how higher level language run-times can be implemented on top of an SVP based system, and can be used to offer easier programming methods for future many-core architectures.

9.1.6 I/O

The final component that we identified as a requirement to construct a general purpose operating system was the support for I/O. In Chapter 7 we proposed an I/O interface for the Microgrid, and have shown in more detail how a system service for I/O can be designed using SVP and the synchronization primitives described in Chapter 4. This proposed structure was then implemented as a service on the Microgrid and evaluated, in which we investigated latency, performance and contention on this service. The results showed that this approach scaled well due to the latency hiding properties of the Microgrid architecture, which worked properly together with the way the service is designed.

9.1.7 Discussion

We have presented mechanisms and solutions in this thesis that provide the required functionality that we identified in Chapter 2. Using these we can facilitate the design and implementation of an operating system for the Microgrid. For example, using the synchronization mechanisms of Chapter 4 and the I/O interface and service design of Chapter 7, other higher level services can be developed that are required to form a complete operating system such as a filesystem or network interface. Therefore, we can claim that we have succeeded in designing operating system components for the Microgrid based on the abstractions offered by the SVP model. Implementation and evaluation of such an operating system now depend on the implementation of the missing features and interfaces in the Microgrid architecture.

There are many technical details left undefined that go beyond the interface that SVP exposes, leaving it up to the SVP implementation, and that can not be expressed using SVP itself. This includes interfaces for managing place configuration (cf. Chapters 5 and 6), memory protection and translation (Chapter 5) and I/O (Chapter 7), though we have proposed and discussed a design for the latter at the Microgrid level. This makes SVP incomplete as a machine model to develop systems on. However, the use of SVP did provide us with a framework which allowed us to reason about and design operating system components at the time when the Microgrid architecture had not been fully specified yet and was still subject to change. Furthermore, it allowed us to construct other systems using the same mechanisms based on SVP such as on the ÆTHER platform.

After having designed our operating system components based on SVP, we can reflect on the use of SVP for such designs. One of the issues that we encountered
with SVP is the lack of general synchronized communication, a core feature required for distributed operating systems in order to communicate between system services and/or applications. Without such synchronizations, we can not implement IPC. However, we have introduced this based on the mechanisms discussed in Chapter 4 (Section 4.3), which allows us to use SVP to effectively support CSP style [79] communications. These approaches heavily relied on using exclusive places and their special synchronizing and consistency behavior.

As the overview at the end of Section 4.3 shows, any of these approaches have a considerable resource cost where either family or place slots are occupied for a synchronizer. While the non intuitive use of these synchronizers can be hidden with an abstraction, it would be desirable to have a more direct support for such a mechanism in SVP. Interestingly enough, the Microgrid in fact supports a more direct mechanism by being able to write into the shared communication channel of an unrelated family as long as the right identifier and capability is presented, as we discussed in the search based synchronization. Therefore we can conclude that on an SVP level these approaches are relatively cumbersome and not necessarily efficient, and we would benefit in our systems from more direct synchronizations that still can efficiently exploit the suspend and scheduling mechanisms of the Microgrid. One example can be to implement hardware support for the Plan 9 style rendezvous synchronization [122] where a synchronizer is defined, the first thread writes to and suspends on the synchronizer and is woken up again as soon as the second thread arrives and exchanges the data.

The reason that arbitrary synchronization is not supported in SVP is that it was attempted to offer a 'safe' programming model in two different ways; first of all by only allowing forward communication and synchronization, the original model could be proven to be deterministic and deadlock free [CSA44], even under composition. This was under the assumption that there is no resource deadlock, which is tackled by the second property that SVP relies on. The second property is that any part of an SVP program can be trivially reduced to a sequential schedule, which is what solves the problem of resource deadlock, and which actually follows from the first property. It should be noted that this however assumes sufficient memory for the required recursion depth [CSA38].

The introduction of exclusive places in the model, which were introduced in order to support critical sections, breaks these two aforementioned properties. Intuitively, this immediately introduces the possibility of communication deadlock in our system, as shown with an example in Chapter 4. Furthermore, we also showed that it restricts the use of the sequentialization property of SVP, as we can only guarantee that a part of a program can be safely sequentialized when it does not make any delegations to exclusive places. While exclusive places introduce these complications, they are required in order to be able to implement operating system structures. In fact, while designing and implementing system services such as the SEP and the I/O service (Chapters 6, 7 and 8), we found them very useful and intuitive to use where we delegated critical sections to exclusive places to guarantee safe updates.

It is clear that the two original properties of SVP still hold under the restricted use of exclusive places; for example it can still be applied to a part of the concurrency tree that is guaranteed not to contain delegations to an exclusive place. However, even with
such a weakened version of the model we still would require additional communication patterns to support operating systems efficiently. While the concurrency patterns of SVP match well with operating systems and their interactions, the associated communication pattern does not. It is more suitable for computational kernels such as data parallel operations or reductions, that can be spawned with a sequence of one or more create actions.

9.2 Conclusion

In this section we will more specifically discuss the answers to the research questions that we have posed in Section 1.3. First we cover our four research questions concerning the Microgrid, before we continue with the two more general questions about the SVP model.

We answered the first question about the Microgrid fairly early on in Chapters 2 and 4, as it was clear that it was not possible to port a contemporary operating system to the architecture in its current design. However, as it does support thread scheduling and management in hardware, this removed the requirement for implementing a software scheduler on every core. This naturally leads to a micro-kernel style approach where the other operating system functionality is offered as services, running as collections of threads on top of this mechanism, which is a similar design to many other many-core and distributed operating system approaches. As we explored further in Chapter 4, this leads to a different view on what a process is. As the process context is no longer managed by a software scheduler, it instead becomes a set of allocated resources. The definition of a process and the way they are scheduled is where our approach differs the most from the other many-core operating system approaches. These findings also dictated the answer to the first half of our fourth research question considering the Microgrid, where we can reiterate that it will be a service based distributed design.

While we have not discussed the second research question in depth throughout this thesis about how and if concurrency can be managed at all levels using the same construct, we have discussed the different mechanisms that should taken into consideration throughout Chapters 4, 5 and 6. Fine grained concurrency can be managed with solely a create action, though when we look at task level concurrency or component level concurrency, some other mechanisms are involved. For example, in order to have a new concurrent executing component, one might first want to request an additional resource to run it on from the SEP resource manager. Similarly, in order to start a concurrent process, besides the interaction with the SEP, also the PDM (protection) and MAM (memory) services (cf. Chapter 5) need to be invoked to configure the resources for the process. Therefore, in principle the answer to the question is yes, the same create action can be used in the end, though potentially additional interactions with services are required to acquire and set up a resource.

The third research question that we tried to answer about the Microgrid was to investigate which features are missing from the hardware implementation that are required for implementing a general-purpose Microgrid operating system. It was already clear that basic features such as memory protection and address translation
9.2. CONCLUSION

are not yet present, but we also came to the conclusion that we required protection on access to resources, for which we proposed a capability based mechanism (Chapter 5). Another important feature that we have proposed in the past was support for exceptions [MWvT8], which we will require for doing fault handling, for example when a process attempts an illegal access that is stopped by one of the aforementioned protection mechanisms.

One question that still remains is if we require support for pre-emption and virtualization of family and thread resources in the Microgrid. We have discussed strategies in Chapter 4 and 6 that can avoid this requirement with a space-sharing approach where resources are partitioned between applications and time-sharing is no longer necessary assuming we have a sufficient number of cores. However, this requires a situation where the number of cores is high enough to be considered ubiquitous, just as memory, which might not be the case for the foreseeable future. The number of cores to achieve this is also very subjective to the type and number of applications the platform needs to support simultaneously. A second need for pre-emption that was identified is to be able to revoke resources from applications, but this can be covered by the proposed upcall mechanism and with the kill action of SVP.

While we can assume that we find solutions and implementations for the other missing features we identified for our third research question, we are still not comfortable about the answer on our fourth research question considering the Microgrid; determining its suitability as a general-purpose computing platform. This answer can only be decided when it can be shown that we have a good solution for the scheduling issue. Either by showing that our proposed space-sharing approach is sufficient or by exploring implementations of pre-emption for the Microgrid, and both these two explorations requires more future research.

The two research questions concerning SVP have largely been answered in our discussion in Section 9.1.7. As SVP does not provide us with the low level machine interfaces that we need to work with in some of our operating system services, it does not seem to be suitable for the development of operating systems. However, we have shown in Chapter 8 how a run-time system can be developed completely based on SVP, and when we look at the I/O service in Chapter 7, a component in our operating system, we see that everything except the special low-level driver component which interacts with the low level machine interface can be defined at the level of SVP. In this service we hit the limitation of arbitrary synchronization though, which is also the problem which does not make SVP suitable for general-purpose computing use. As mentioned before, it can be used efficiently to express certain, even resource agnostic, functional computation kernels. However, it requires complicated constructions to cope with the synchronization and consistency in more control code oriented programs, therefore we can conclude that it is not suitable for general-purpose computing, but rather for specific algorithmic or computational tasks.

While the use of SVP had its downsides requiring non trivial workarounds, such as the lack of arbitrary synchronization, we want to make clear that SVP was still very useful in the course of our research as it provided a reasoning framework for the Microgrid architecture. SVP, and the software based implementations that we developed of it (cf. Chapter 2), could be used to explore the design space to discover
which mechanisms should be implemented in the Microgrid architecture. It enabled
the work described in this thesis, which has now identified how and which mechanisms
are further required to develop the Microgrid towards a true general-purpose com-
puting platform. The next step would be to investigate how to improve the model,
while retaining its simplicity which allows for an efficient hardware implementation.

9.3 Future Work

During the work on this thesis we have already identified several essential mechanisms
that are still lacking from the current implementation of the Microgrid, and that
should be covered in future research. This includes support for memory protection
and address translation, and interfaces for setting up the security features based on
capabilities. Another feature that requires future evaluation is an implementation
to pre-empt and virtualize the thread and family tables which allows the system to
have more flexibility in handling its resources; in a way this can be used to expose
an ‘infinite’ machine to its programmers, similar to what contemporary operating
systems tend to do to a certain extent. The cost of such an implementation should
then be compared with the effectiveness of a pure space-sharing approach without pre-
emption. This requires an extensive exploration and evaluation of the space-sharing
strategies that we have described in Chapter 4, where policies should be investigated
to manage resources without requiring pre-emption.

Related to further investigation of the space-sharing approach is the required fu-
ture work on resource management. We expect resource management to be one of the
key challenges in future many-core operating systems, especially when more hetero-
genecity is introduced in hardware. While we have discussed two extreme approaches,
more hybrid approaches should be investigated. One of the key questions will be what
the interface should be to the resource management system; how does the program
specify what it requires?

In terms of programming the Microgrid, it would be interesting to investigate
how other popular concurrent programming models would map onto the mechanisms
of the architecture, which can perhaps also deliver inspiration on how to improve
SVP. Interesting candidates would be concurrent programming languages such as
Cilk [21], Chapel [32] or X10 [34]. It is important to keep concurrent programming
patterns (both in terms of computation and communication) in mind so that they
can be efficiently supported by both the programming model and the architecture. A
good guide for this is the report from Berkeley [10], which identifies 13 ‘Dwarfs’ that
capture different patterns of communication and computation.

The final most obvious part in future work is to design and implement a complete
general-purpose operating system for the Microgrid, given that the interfaces for the
missing features have been defined for the architecture. This work will then include
investigating the optimal parameters, strategies and policies for the services proposed
in this thesis. We believe that this thesis will be a useful contribution to future many-
core operating system implementation work, and that it can function as a handbook
on how to design a Microgrid operating system.
Een omwenteling is gaande in de computer industrie. Jarenlang zijn de ontwerpen voor de centrale verwerkingseenheid van de computer, de processor, steeds sneller geworden bij elke nieuwe generatie producten. Ondanks de nog steeds beter wordende productietechnologie lopen we nu tegen een grens aan waarbij deze snelheidstoename voor een enkele processorkern aan het stagneren is, veroorzaakt door limieten zoals hitte en energieverbruik. De algemene trend om alsnog meer reken capaciteit met iedere volgende generatie processoren te bieden, is het integreren van meerdere processor rekenkernen in een enkele processor, waarbij meerdere taken tegelijkertijd uitgevoerd kunnen worden. Dit soort multi-core processoren zijn op dit moment gemeengoed in alledaagse computer systemen met bijvoorbeeld tussen de twee en zestien rekenkernen. In de nabije toekomst met het oog op de verdere verkleining van productietechnologie voorzien we processor ontwerpen met honderden tot duizenden rekenkernen, zogenaamde many-core architecturen. Het is een grote uitdaging om dit soort systemen op een schaalbare manier te ontwerpen, beheren en programmeren.

Bij de Computer System Architecture onderzoeksgroep aan de Universiteit van Amsterdam wordt gewerkt aan een nieuwe processor architectuur, de Microgrid. Deze processor heeft als doel een architectuur te ontwikkelen die kan opschalen naar honderden tot duizenden rekenkernen. In deze architectuur bestaan programma’s uit kleine deeltaken (microthreads) die verdeeld kunnen worden over meerdere rekenkernen. Om dit efficiënt te kunnen doen, worden deze deeltaken beheerd door mechanismen in de hardware van de processor. Dit is in tegenstelling tot hedendaagse computer systemen waar dit beheer in het algemeen door de software van het besturingssysteem wordt gedaan. Deze verschuiving van verantwoordelijkheid tussen software en hardware vereist een ander ontwerp voor een besturingssysteem, naast de benodigde veranderingen om het te laten opschalen naar het beheren van duizenden rekenkernen.

Dit proefschrift maakt een verkenning van hoe een besturingssysteem gebouwd zou kunnen worden voor de Microgrid processor architectuur. Deze verkenning wordt gemaakt aan de hand van vijf basis-mechanismen die vereist zijn om een algemeen besturingssysteem te kunnen implementeren. De vijf mechanismen die wij identifieren in Hoofdstuk 2, omvatten het laden en plannen van processen, geheugen en adresruimte beheer, beveiliging tussen processen, inter-proces communicatie/synchronisatie en het beheren van in- en uitvoer communicatie met de buitenwereld. Tevens verkennen wij de structuur van besturingssystemen in het algemeen, en leiden hier de strategie uit af dat besturingssysteem ontwerpen voor toekomstige many-core archi-
tecturen en de Microgrid het beste gedistribueerd kunnen zijn voor schaalbaarheid. In Hoofdstuk 3 beschrijven wij het ontwerp van de Microgrid in meer detail en introduceren wij SVP, wat een rol heeft als programmeermodel voor de Microgrid, maar ook als abstract model van de Microgrid hardware geïmplementeerd als een emulatie in software.

De effecten van het hardwarematige beheer van microthreads op de definitie van een proces en op de manier van het beheren en plannen van processen beschrijven wij in Hoofdstuk 4. Hier stellen wij een twee-laagse oplossing voor waarbij het fijnmazige beheer van de kleine microthreads door de hardware gebeurd terwijl het besturingsysteem op hoger niveau gehele rekenkernen aan applicaties toewijst. Verder wordt een analyse gemaakt van de verschillende manieren waarop twee processen kunnen synchroniseren in SVP gebaseerde systemen om inter-proces communicatie te kunnen implementeren. Hoofdstuk 5 bouwt hier op verder en kijkt naar de beveiliging en het beheer van geheugen en adresruimte gekoppeld aan onze definitie van een proces en de beveiligingsaspecten van het beheren van microthreads door de hardware. Dit laatste aspect is afwijkend van hedendaagse systemen, maar is belangrijk om bijvoorbeeld te voorkomen dat een proces de hardware mechanismen kan gebruiken om nieuwe microthreads te starten op een rekenkern welke het niet toegewezen heeft gekregen van het besturingsysteem.

Hoofdstuk 6 en Hoofdstuk 7 gaan verder in op de details van specifieke diensten in een Microgrid besturingsysteem. In het eerste hoofdstuk beschrijven we de SEP beheersdienst die de toewijzing van rekenkernen aan processen verzorgt, waarbij wij twee verschillende aanpakken verkennen voor de interactie tussen processen en deze dienst. In Hoofdstuk 7 presenteren we een systeemdienst voor in- en uitvoer, alsnog een ontwerp voor een in- en uitvoer mechanisme in de Microgrid architectuur. Van deze diensten worden implementaties gepresenteerd met enkele experimenten om deze aanpak te valideren en het gedrag van een dergelijk systeem te laten zien. In Hoofdstuk 8 behandelen we een programma-omgeving die gespecialiseerd is om gebruik te maken van de SEP beheersdienst. Deze omgeving is ontworpen om paralele programma’s die gebaseerd zijn op de S-Net coördinatie taal uit te voeren, welke gericht zijn op het verwerken van datastromen.

Ten slotte maken we in Hoofdstuk 9 de uiteindelijke analyse en discussie over de haalbaarheid van het bouwen van een besturingsysteem voor de Microgrid. Terwijl er duidelijk nog mechanismen ontbreken aan het huidige Microgrid ontwerp hebben wij kunnen laten zien welke oplossingen er mogelijk zijn om de eerder genoemde vijf basis mechanismen te implementeren. Hieruit kunnen wij afleiden dat wij voldoende basis hebben gecreëerd om een algemeen besturingsysteem voor de Microgrid te kunnen implementeren. Dit proefschrift heeft geen concreet ontwerp gepresenteerd; het kan gebruikt worden als handboek voor het ontwerpen en bouwen van een besturingsysteem voor een processor architectuur zoals de Microgrid.
First of all I would like to thank my father, as I am sure that he would have been extremely proud on what I have achieved. It was a difficult period while starting my PhD research at the same time, but all I can say is that I love you and miss you. This period also really strengthened the bond with my mother, and I am happy that she supports me in what I try to do, and sometimes pushes me a bit when I need it.

I would like to thank my supervisor and promotor, Chris Jesshope, for offering me the position to do a PhD after completing my Master’s project with the group. I really appreciated the amount of freedom that you gave me to pursue my own ideas, and allowed me to take responsibility in projects as well as in student supervision. While we did not always agree and I was sometimes worried about the focus, productivity and direction of our research, I am happy with the end result that we achieved together.

I want to thank Andy Pimentel first of all for his inspiring Computer Architecture lectures back in the days when I was still a student, as well as for his patience and later introducing me to Chris when I was searching for a graduation project. I really valued your accessibility and support, I could always knock on your office door and have a quick chat to exchange thoughts (or vent frustration) about anything. I wish you good luck with managing and steering the group through the University’s stormy waters, I’m sure they are in good and capable hands.

Similarly, I would like to thank Clemens Grelck for the many interesting conversations we had, where the topics ranged from European history to public transport, or from (University) bureaucracy to research problems. You often gave very strong feedback to my work, perhaps sometimes a bit too strong, but it was always useful.

The CSA group has always had a very pleasant and friendly atmosphere, including our legendary 3 o’clock tea breaks, movie nights and group outings. I want to thank all my colleagues from the group for this period. Kostas for all the fun we had working on the ÆTHER project and on our foreign trips to our project partners, the humor with Jony and Tom, and the many tea breaks, lunches or drinks with Thomas (and the rest of the gang at the time). Simon for the many brainstorming and debugging sessions we had, as well as our shared love for snooker and ThinkPads which are -of course- the best laptops ever made. Mark for his friendship and help, and for always being open to listen or talk about anything.

I had the pleasure to share my office with several people at the group over time; Duong, you brought me the extremely strong Vietnamese green tea that I still enjoy
Sometimes, and would always tell me the latest gossip from Vietnam. Mike, a great help as interactive C/C++ manual and troubleshooter, our shared love for the Command & Conquer games and especially their music, and the many fun distractions often involving South Park episodes. Joe, to point out the subtle differences between American and British language and culture. The Chinese background chatter from Qiang and Jian and Fangyong, and the interruptions with random statements from Merijn. Roy and Roeland for overloading me with cookies, which only started out as a joke. Roy, I really had a lot of fun together with you hacking on the SCC, as well as on our trips, and the same holds for Merijn. Irfan, I am happy that you were interested to pick up the high level simulation work, and to see how you have developed and what you achieved over the years.

Without a doubt, my most special colleague with whom I shared an office has been Raphael. I still fondly think back of the time that we shared the office with just the two of us in the old building, where we had so many interesting (but sometimes not so useful) discussions about where we should take our research. You were an incredible support for me whenever I felt a bit down or had problems seeing the value of my work, always trying to inspire me again.

As for the other members of our group I would like to thank Peter for our discussions on the SPARC T-series, Roberta for bringing some female flavor to the office as well as our struggles with the Xilinx tools, the ever friendly and smiling Toktam, and Michael for working out the I/O system together and the nice paper that we produced about it. A special thanks should also go out to Erik, our group secretary, who would always do his best to answer any question or to help me out with any problem I might have.

Outside our group I had the pleasure to meet and work with many people from our European project partners. Frank, Alex and Sven-Bodo (then) at UH, Martin, Zdeněk and Lukáš at UTIA, Jean-Marc and Benoît at CEA-LIST and Juha K. and Juha P. at VTT. I am really proud that we managed to work together to integrate our technologies in the final ÆTHER demo. I enjoyed meeting and/or working with Ulrich, Michael K., Michael R., Werner, Ted and Jim at Intel Labs when we were studying the Intel SCC. Ruud at Oracle, it was nice to hear your enthusiastic feedback on my SPARC T3 report and you opened many doors for me, I hope we will be able to work together in the future.

Outside the people I directly worked with there are many people from other groups (mostly SCS and SNE) that I have met at the office and interacted with throughout the years. There are too many things to mention, chatting at the coffee corners, having lunch, parties, playing squash, or simply friendly greetings in the hallways. Thanks go out to Gökhan, Kamana, Rudolf, Ralph, Cosmin, Emiliano, Carlos, Fabio, Eric, Rick, Jaap (thanks for helping me back on track in 2005), Nol, Lotte, Adam, Dick, Narges J., Narges Z., Shan Mei, Carolina, Marit, Sofia, Gohar, Alfredo, Yves, Jordi, Bianca, Alexis, Pam, Dorien and anyone else whom I forgot to mention.

I would like to thank the members of my committee; I am proud to have prof. Herbert Bos and prof. Sape Mullender on board as thorough experts in the field. Alex, I have always enjoyed meeting and talking to you, and Rob thanks for the times we sat down for feedback on my research. Cees, I am sure there will be cake. Outside of my committee I would also like to thank prof. Timothy Roscoe for inviting
me for a talk in Zürich and for his feedback, which made me realize that I had to strengthen my story.

Of course there is also life outside of the academic research context of my PhD work. During the first three years I was still working part-time at the Duitsland Instituut Amsterdam where I have worked for nine years. I would like to thank all my former colleagues there for the nice and friendly atmosphere, and the interest that they have shown in the progress of my PhD.

Music has been a passion of mine for a long time, and is a great way to relax and escape from the troubles of doing a PhD. Either by creating my own music, alone or together with others, by playing or by attending events. Therefore I would like to thank the Dutch Drum & Bass scene and the many people I know there as a whole for its parties and performances, and the people from Jungletrain.net in particular where I am still hosting a regular radio show.

Thanks go out to Ruben, Martijn and the rest of the ex-Kern-posse for the relaxing and fun VrijMiBo’s after a hard week of work. Gerald, or cavey, the person with whom I can daily share and rant about anything, and who always tries to support me and make me see the bright side of things. You have been a great friend throughout the years, even though we very rarely see each other in real life. However, I would be nowhere without my group of closest friends, who I all know for at least 10 to 15 years or even longer. Olivier, Loes, Peter Paul, Marius, Koos and Mathilde, and of course little Vince and Douwe, thanks for being my support. I am not sure how I would have made it without you! Many games of squash, Agricola or snooker or just general ‘gezelligheid’ have helped me to relax, not to mention our awesome cycling holidays!

Thank you!

Michiel.
General Bibliography


Publications by CSA and affiliated project partners


Publications by the author


Michiel W. van Tol received his MSc degree in Computer Science from the University of Amsterdam in 2006. He completed this PhD thesis at the University of Amsterdam in 2012.

The current trend in the computer industry is to integrate more and more processor cores onto a chip as power and heat limitations have stagnated the increase of single core speed. In the future we expect to see many-core chips with 100s to 1000s of computing cores. The CSA group at the University of Amsterdam has been working on a scalable many-core design, the Microgrid. A special feature of this design is that it manages concurrent tasks in hardware, which allows computations to be efficiently spread over the large number of cores.

This thesis investigates how to construct an operating system for the Microgrid, and identifies which features are missing from the current hardware design to support a general-purpose operating system. A set of features that is required to implement an operating system is identified, and an analysis is made how these can be constructed based on the mechanisms of the Microgrid. The primary contribution of this thesis is that it provides a blueprint of how a Microgrid operating system can be built. Other contributions include the development of tools to emulate or simulate the behavior of the Microgrid, and a proposal for a concurrent software runtime system.