DISS. ETH NO. 20930

TACKLING OS COMPLEXITY WITH DECLARATIVE TECHNIQUES

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

ADRIAN LAURENT SCHÜPBACH

Master of Science ETH in Computer Science, ETH Zurich

19. April 1981

citizen of Landiswil, BE

accepted on the recommendation of

Prof. Dr. Timothy Roscoe
Prof. Dr. Gustavo Alonso
Prof. Dr. Hermann Härtig

2012

Kurzfassung

Diese Dissertation zeigt, dass die erhöhte Betriebssystemkomplexität, die durch die Notwendigkeit entsteht, sich an eine grosse Anzahl unterschiedlicher Rechnersysteme anzupassen, mittels deklarativer Techniken signifikant reduziert werden kann.

Moderne Hardware ist zunehmend unterschiedlich und komplex. Es ist wahrscheinlich, dass sich diese Entwicklung in Zukunft fortsetzt. Diese Entwicklung erschwert den Betriebssystembau. Betriebssysteme müssen sich der Rechnerarchitektur optimal anpassen. Sie müssen den gesamten Funktionsumfang des Rechners ausschöpfen, um die volle Leistung des kompletten Systems zu gewährleisten. Vom Betriebssystem ungenützte, suboptimal genützte oder gar falsch genützte Rechnerfunktionalität führt zu geringerer Leistung des Gesamtsystems. Traditionelle Betriebssysteme passen sich durch vorgefertigte Regeln, die im ganzen Betriebssystem verteilt und mit der eigentlichen Betriebssystemfunktionalität vermischt sind, der Rechnerarchitektur an.

In dieser Arbeit argumentiere ich, dass es aus zwei Gründen nicht mehr möglich ist, vorgefertigte Regeln für eine Anzahl bekannter Rechnerarchitekturen mit der Betriebssystemfunktionalität zu vermischen. Erstens garantieren vorgefertigte Regeln nicht, dass alle Rechnerfunktionen vollständig ausgeschöpft werden. Zweitens bedeutet das, dass die Regeln für jede neue Rechnerarchitektur angepasst werden müssen, wobei die Regeln für bisherige Rechnerarchitekturen beibehalten werden müssen. Dies führt zu erheblicher Betriebssystemkomplexität und schliesslich zu einem enormen Anpassungsaufwand. Um dies zu vermeiden, muss das Betriebs-

system während der Laufzeit Wissen über die Rechnerarchitektur aufbauen und daraus, durch logische Schlussfolgerungen, die bestmöglichen Anpassungsregeln ableiten. Dies führt zu einfacheren, verständlicheren, pflegeleichteren und leichter anpassbaren Betriebssystemfunktionen und stellt sicher, dass die Rechnerfunktionalität vollständig ausgeschöpft wird.

Der Wissensaufbau und das Ableiten von Anpassungsregeln durch logische Schlussfolgerungen sind mit hoher Programmierkomplexität verbunden. Dies gilt insbesondere, wenn dafür maschinennahe Programmiersprachen, wie zum Beispiel C, verwendet werden. Deklarative Techniken erlauben hingegen, angestrebte Regeln durch eine einfache und verständliche Beschreibung der gewünschten Art der Anpassung, basierend auf Wissen über die Rechnerarchitektur, abzuleiten. Durch die natürliche Beschreibung in höheren Programmiersprachen wird die Programmierkomplexität stark verringert.

Um den Vorteil deklarativer Techniken im Zusammenhang mit Komplexität und Anpassungsfähigkeit in Betriebssystemen zu beweisen, stelle ich in dieser Dissertation verschiedene Fallstudien vor, die, basierend auf deklarativen Techniken, Regeln für die Anpassung an die Rechnerarchitektur, mittels logischer Schlussfolgerungen, ableiten. Die Fallstudien setzen kein Wissen über die Rechnerarchitektur voraus, sondern eignen sich dies während der Laufzeit an.

Das Wissen wird in einem zentralen Wissensdienst des Betriebssystems aufgebaut. Regeln werden in diesem Wissensdienst durch logische Schlussfolgerung abgeleitet. Dadurch, dass die Fallstudien, und somit die verschiedenen Betriebssystemkomponenten, diesen Wissensdienst benützen können, wird ihre Komplexität nochmals deutlich verringert. Es ist somit nicht nötig, dass sich jede einzelne Betriebssystemkomponente mit der Wissensgewinnung und der Ableitung von Regeln beschäftigt. Mit dieser Implementation beweise ich die praktische Anwendbarkeit deklarativer Techniken in Betriebssystemen.

Abstract

This thesis argues that tackling the increased operating systems complexity with declarative techniques significantly reduces code complexity involved in adapting to a wide range of modern hardware.

Modern hardware is increasingly diverse and complex. It is likely that this trend continues further. This trend complicates the operating system's construction. Operating systems have to adapt to the hardware architecture and exploit all features to guarantee the best possible overall system performance. Not exploiting all hardware features, or using them in a suboptimal or even wrong way, results in lower overall system performance. Traditionally, operating systems adapt to the underlying architecture by predefined policies, which are intermingled with the core operating system's functionality.

In this thesis I argue that for two reasons it is no longer possible to encode predefined policies for a set of known hardware architectures into the operating system. First, predefined policies do not automatically guarantee that hardware features are fully exploited on all hardware platforms. Second, for this reason, predefined policies would need to be ported to many different hardware platforms, while, at the same time, it would be necessary to keep the policies suitable for older platforms. This leads to a significant complexity of operating systems and finally to a high engineering effort, when porting the operating system to new hardware platforms. To avoid this problem, the operating systems must gain hardware knowledge at runtime and derive policies suitable for the current architecture through online reasoning about the hardware. This leads to operating sys-

tems code that is simpler, better understandable, more maintainable and easier to port, while ensuring that the operating system exploits the hardware features as best as possible.

Reasoning about hardware and deriving policies is a complex task. This is especially the case, if low-level languages like C are used. Instead, declarative techniques allow deriving policies through a simple description of how to adapt to the hardware based on hardware knowledge gathered at runtime. The natural description in a high-level declarative language reduces code complexity significantly.

To prove the usefulness of declarative techniques in the context of adaptability of operating systems and handling of complexity, I present several case studies in this thesis. The case studies are based on declarative techniques. They reason about hardware and derive policies based on hardware knowledge. The case studies do not assume any a priori knowledge about the current hardware platform. Instead, they gain knowledge at runtime by online reasoning about the hardware.

A central knowledge service stores hardware knowledge and allows the operating system and applications to derive policies according to declarative rules. Because the case studies, and therefore the operating system components, can use the central service, their complexity is again reduced significantly. It is not necessary, that every single component deals with knowledge gathering and deriving policies by itself. It pushes this part to the knowledge service. With this implementation I prove the practical feasibility of applying declarative techniques in real operating systems.

Acknowledgments

First of all, I would like to thank my advisor, Prof. Dr. Timothy Roscoe, for all his help, for interesting technical discussions and for his invaluable advises and feedback on this thesis. I would also like to thank the entire Barrelfish team for the interesting discussions and nice time. Furthermore I would like to thank Simonetta Zysset for proofreading my thesis and for giving me valuable feedback on the language.

Fortunately, I had the chance of discussing problems with several members of the Systems Group, either in the office, or during lunch. The discussions helped me to look at a problem from a different angle. Special thanks go to Michael Duller, Jan S. Rellermeyer, René Müller, Jens Teubner, Louis Woods, Qin Yin, Philip Frey and Ionut Subasu.

During my time at ETH I also had to do many months of “Zivildienst”, causing me a complicated time management. I would like to thank Spital Davos, especially the IT department and the Spitalleitung for their flexibility and also for allowing me to go to conferences during this time. Special thanks go to Florian Steiger, Peter Driedijk, Luzius Valär, Markus Hehli and Monika May. Furthermore, I would like to thank Andi Roveretto and Nadine Krättli for the joyful and funny time at Spital Davos.

Most importantly, I would like to thank my mother Anna, my father Laurent, my sister Letizia and my girlfriend Nadine Duivenstijn for their continuous support and motivation and for always believing in me.

Finally, I would like to thank Dr. Arnold Spescha for always listening to me and for “pushing” me through the Gymnasium, which was the basic step towards this thesis.

Contents

1 Introduction **1**

1.1 Motivation . . . 3

1.1.1 Diversity . . . 6

1.1.2 The interconnect network . . . 9

1.1.3 Managing Hardware . . . 12

1.1.4 Managing Applications . . . 14

1.2 Problem Statement and Hypothesis . . . 14

1.3 Goals . . . 15

1.4 Contributions . . . 15

1.5 Structure . . . 17

2 Background **19**

2.1 Declarative Techniques . . . 19

2.1.1 What is declarative programming? . . . 20

2.1.2 Declarative languages . . . 21

2.1.3 Constraint logic programming . . . 22

2.1.4 CLP programming in ECL^{i}PS^{e} . . . 24

2.2 Barrelfish . . . 25

2.2.1 The Multikernel . . . 26

2.2.2 A Barrelfish "node" . . . 28

2.2.3 Explicit access to physical resources . . . 30

2.2.4 Messaging . . . 31

2.2.5 Drivers and services . . . 32

2.3 Reasoning in operating systems 33
2.3.1 Hardware representation 33
2.3.2 Declarative hardware access and configuration . . 34
2.3.3 Resource allocation 35
2.4 Declarative reasoning in networks 37
2.5 Summary 38

3 The system knowledge base 39
3.1 Introduction 40
3.2 Background 41
3.2.1 Knowledge 41
3.2.2 Knowledge bases 43
3.3 How does the SKB help the operating system? 45
3.3.1 Purpose 45
3.3.2 Examples 46
3.3.3 Common patterns of resource allocation descriptions 47
3.3.4 When to use the SKB 49
3.4 Design 50
3.4.1 Design principles 50
3.4.2 Overall architecture 53
3.4.3 Core 55
3.4.4 Interface 56
3.4.5 Facts, schema and queries 58
3.4.6 Data gathering 59
3.4.7 Algorithms 62
3.4.8 A note on security 64
3.5 Implementation 65
3.5.1 Implementation of the SKB server 66
3.5.2 Facts and schema 67
3.5.3 Datagatherer 69
3.5.4 Common queries 70
3.5.5 Startup 70
3.6 Client library 71
3.6.1 Using and initializing the library 71
3.6.2 Interacting with the SKB 72

3.7 Evaluation 77
3.7.1 Code complexity 77
3.7.2 Memory overhead 78
3.7.3 Performance 79
3.8 Discussion 80
3.8.1 Advantages 80
3.8.2 Disadvantages 83
3.8.3 Approaching a configuration problem in CLP . . . 88
3.9 Summary 89

4 Coordination **91**
4.1 Introduction 92
4.2 Background 93
4.3 Approach 94
4.3.1 Design principles 95
4.3.2 Octopus 96
4.3.3 Records and Record Queries 97
4.3.4 Record Store 99
4.3.5 Publish-subscribe 100
4.3.6 Implementation 100
4.4 Use-cases 101
4.4.1 Synchronization primitives 101
4.4.2 Name service 102
4.4.3 Application coordination 103
4.4.4 Device management and system bootstrap 104
4.5 Evaluation 104
4.5.1 Code complexity 105
4.5.2 Performance 105
4.6 Summary 108

5 Device management **109**
5.1 Kaluga 110
5.1.1 Architecture 110
5.1.2 Driver mapping files 111
5.1.3 Hardware records 113

5.2 Hardware discovery 114
5.2.1 Hardware discovery life-cycle in Barrelfish 114
5.2.2 View hotplugging as the default case 117
5.2.3 Minimize basic architecture and platform information 117
5.2.4 Device information 118
5.3 System Bootstrap 119
5.4 Evaluation 119
5.4.1 Correctness 120
5.4.2 Code complexity 120
5.5 Related work 121
5.6 Summary 122

6 Declarative PCI configuration 125
6.1 Introduction 126
6.2 Background: PCI allocation 128
6.2.1 PCI background 128
6.2.2 Basic PCI configuration requirements 130
6.2.3 Non-PCIe devices 132
6.2.4 Fixed-location PCIe devices 133
6.2.5 Quirks 133
6.2.6 Device hotplug 135
6.2.7 Discussion 136
6.3 PCIe resource allocation 138
6.3.1 Approach 139
6.3.2 Formulation in CLP 142
6.3.3 Quirks 148
6.3.4 Device hotplug 149
6.4 Interrupt allocation 152
6.4.1 Problem overview 152
6.4.2 Solution in CLP 153
6.5 Evaluation 156
6.5.1 Test platforms 157
6.5.2 Performance 158
6.5.3 Code size 158

6.5.4 Handling quirks 160
6.5.5 Postorder traversal comparison 162
6.6 Summary 164

7 Efficient Multicast Messaging 167
7.1 Introduction 168
7.2 Background 170
7.2.1 Multicast messaging 170
7.2.2 TLB shootdown 172
7.2.3 Summary 173
7.3 Design 174
7.3.1 Design principles 174
7.3.2 Hardware-aware multicast tree 175
7.4 Implementation 178
7.5 Evaluation 183
7.5.1 Adaptability 184
7.5.2 Code complexity 184
7.5.3 Execution time 184
7.5.4 Effective multicast performance 185
7.6 Summary 187

8 Global Resource Management 189
8.1 Introduction 190
8.2 Background and related work 193
8.3 Model hardware and global allocation 195
8.3.1 Hardware model 195
8.3.2 Application model 196
8.3.3 Application requirements 197
8.3.4 Translating requirements to constraints 199
8.3.5 Decision variables and concrete topology-aware allocation 201
8.4 Resource manager 201
8.5 Framework to register parallel functions 202
8.5.1 Using the framework 204
8.5.2 Terminating threads 207

8.5.3 Overall architecture 208
8.5.4 Use-cases 209
8.6 Use case 1: pbzip2 209
8.6.1 Architecture 210
8.6.2 Evaluation 211
8.6.3 Summary 216
8.7 Use case 2: Column store 216
8.7.1 Problem 217
8.7.2 Internal knowledge 218
8.7.3 Registering scanning function 219
8.7.4 Evaluation 219
8.7.5 Summary 222
8.8 Evaluation of the allocation policy code 223
8.8.1 Code complexity 224
8.8.2 Execution time 225
8.9 Summary and future work 226

9 Conclusion **229**
9.1 Summary 229
9.2 Directions for future work 230

Bibliography **233**

Chapter 1

Introduction

This thesis argues that operating systems face a significant challenge to adapt to a wide range of diverse hardware found already today. As the diversity and heterogeneity of hardware is likely to increase, the complexity involved in adaptability and smart decision taking is growing. From a portability and software engineering aspect, it is therefore not possible anymore, to intermangle policies throughout the operating system code. Further, generic policies are not an option, because they do not automatically yield to optimal hardware usage on every platform.

This thesis further argues that the operating system needs to reason online about the current underlying hardware to adapt as best as possible to every platform. Reasoning involves deep knowledge of hardware and can quickly lead to high complexity. Typically, there is a lot of data about hardware. According to Niederliński[97], *data* in a specific context provides *information* about it. The ability to use the information to achieve a specific goal, like, for example, adapting to hardware, leads to *knowledge* about hardware, which can be used to derive informed policies (see also section 3.2.1 for the complete definition).

The complexity involved in reasoning and decision taking has to be taken out of the operating system code to enable adaptability and portability to a large set of diverse hardware. This thesis presents the design

and implementation of the system knowledge base (SKB), the reasoning facility of the operating system with the goal to reduce code complexity in both, the operating system's mechanism code and the policy code. It is the central place to store knowledge and derive policy parameters online, based on hardware information of the current underlying platform. A clear policy/mechanism separation throughout the complete system enables the programmer to implement policy code and mechanisms separately leading to a much lower complexity and higher portability. Reasoning algorithms rely on high-level knowledge in a machine-independent format and mechanism code is simple, because it does not need to take decisions based on hardware information. For this thesis I chose the ECL^{i}PS^{e} constraint logic programming system to implement reasoning algorithms, because it is an expressive high-level declarative language and it is easy to port.

The thesis presents concrete use-cases for the SKB, showing what type of data is needed to derive policies and how algorithms transform this data into context-specific knowledge. I introduce and motivate the specific use-cases in the respective chapters while also providing the necessary background and a short use-case-specific evaluation.

The thesis is part of the Barrelfish project[15], a joint work between ETH Zürich, Microsoft Research Cambridge, Microsoft Research Redmond and Microsoft Research Silicon Valley. Parts of the thesis have been published in several papers[17, 18, 48, 49, 106, 107, 116, 117, 118, 144] and I refer to the concrete ones on a per chapter basis. Together with Andrew Baumann and Simon Peter, we created the basic system and formed it into a solid and stable basis for doing operating systems research on top of it. Andrew Baumann contributed mainly on the distributed nature of Barrelfish, including the Multikernel and the capabilities[17] and on Barrelfish's message passing[18]. Simon Peter's PhD thesis[105] is mainly about scheduling in a Multikernel and he also contributed to the capability system, the Multikernel and Barrelfish's message passing. Timothy Roscoe contributed to the Multikernel, the capability system and the message passing while specifically working on Mackerel[114], a device description language and Hake[113], a build system for heterogeneity support. Akhilesh Singhania worked on routing of messages and contributed to the Multkernel and capability system. Jan S. Rellermeyer contributed

to the message passing and worked on a name service for named communication endpoints lookups. Pierre-Evariste Dagand contributed an interface description language for message passing[32, 33]. Tim Harris worked on language constructs to facilitate using the asynchronous message passing interface for programmers by avoiding the necessity of “stack-ripped” code[55, 56]. Paul Barham and Rebecca Isaacs mainly worked on message passing and the Multikernel. Pravin Shinde is currently working on high-performance networking based on low-level demultiplexing, new hardware features provided by NICs and user-space network stacks. Kornilios Kourtis is mainly working on scalable file systems. Stefan Kästle is working on possible hardware designs for hardware-based message passing with demultiplexing facilities in hardware providing isolation between message channels on the same core.

1.1 Motivation

This section introduces the main reasons for the increased diversity and heterogeneity of current and future hardware. It motivates the need for adaptability to hardware at the operating system's level by showing hardware diversity already found in today's machines.

To improve execution performance of applications on desktop machines and servers, in the past few years the clock frequency of processors could be raised, while keeping the architecture mostly the same. As a consequence, applications ran faster without having had to change them. Now, a critical point has been reached, where it is not possible anymore to simply raise the clock frequency due to physical limitations like, for instance, the heat produced[128, 129]. Instead, a higher degree of parallelism in terms of multiple cores is offered by the hardware to improve performance of applications[22]. This, however, has consequences in the whole hardware design (which I show below), such as specialization of computing units and increased heterogeneity, interconnect topologies and memory hierarchies. This again has implications on the operating system's design.

Nowadays, we have commodity machines with up to 128 hardware

execution contexts (for example four Intel Xeon E7-4870 CPU packages with a total of 80 hardware execution contexts or a SPARC T3 processor with 128 hardware execution contexts) and they are mostly homogeneous in terms of CPU type per system. Current operating systems can deal with this number of cores, even if they originally were not designed for many-core machines. In the future, machines with hundreds of cores are expected to improve the performance even more by providing a high degree of parallelism[22, 58]. This allows desktop machines to run a wide range of compute-intensive applications like, for example, RMS workloads[58]. This trend has implications on the hardware construction side.

Cores are expected to be more specialized to certain functions. Computations will need to be placed on the right core to execute on by the operating system. Not only processor cores, but also special devices like offloading hardware of smart NICs[96], cryptographic accelerators (for example in Sparc T3 processors), FPGAs and GPUs[51] will participate in the computation. The hardware is becoming increasingly heterogeneous and this trend is likely to continue in this direction.

The interconnect between different cores, caches, memory and devices are much more complicated even nowadays and looking at recent trends, the complexity is likely to grow even further. Current interconnects look more like a network than like a bus[18]. Some devices will be near together while the communication between others might be routed over several bridges and switches of the interconnect. Different paths have different characteristics in terms of latency, bandwidth and throughput. Section 1.1.1 shows that there are already clear latency differences in the network-like interconnect.

It is the task of the operating system to assign hardware resources such as CPU cores, memory, accelerators, offloading hardware, devices, disk, interconnect bandwidth or network connections, to applications. The operating system not only needs to multiplex hardware safely, but it also needs to derive smart allocations to exploit hardware features, meet applications' expectations on hardware and finally improve hardware utilization. This is an increasingly difficult task on heterogeneous hardware.

Smart *policies* decide *which* parts of the resources are to be assigned to which computation. The policies become more complex, because, as I

show in chapter 8, they need to include hardware topology knowledge, hardware feature information, and application requirements in order to achieve optimal performance for the overall system.

The operating system is an important part of the software stack. As the thesis argues in the next paragraph and in section 1.2, it is impossible to manually tune software to a set of known hardware. Instead, software needs to adapt automatically to a wide range of divers hardware, where the hardware configuration is not known in advance. Because the operating system is itself an important part of the complete software stack, it also has to adapt itself automatically to the underlying hardware. The operating system must never be the bottleneck in terms of scalability, because its scalability directly affects applications' scalability. Chapter 7 clearly shows how application scalability is affected by the scalability of performing a globally coordinated operating system operation. A poor, non-hardware-aware implementation of an operating system operation, which executes on behalf of the application, limits the application's scalability. Further, it must not prevent applications to extensively use the complete available hardware, even if it is highly heterogeneous. Instead, the operating system's task is to actively support applications to use all hardware as much as possible.

Nowadays, it is not practical anymore to manually tune operating systems and applications for specific many-core systems when deploying them. In a mass-market deployment scenario, there are too many different kinds of hardware types available. Generic policies are not suitable for all kinds of hardware types and do not automatically lead to optimal hardware utilization in all the cases. Instead, operating systems, language runtimes and applications, with help from the operating system, have to automatically adapt to the current hardware in a sensible way. Additional resources should improve performance or at least not decrease it. Additional cores should not cause contentions in the memory system such that performance decreases. Sensible allocations of cores and memory by the operating system is important, independently of the type of hardware the system is currently running on. Furthermore, hardware is shared by many applications. The system has to manage a dynamic set of different long running and interactive applications and cannot statically partition the machine to a fixed

set of applications. As I argue in chapter 8, the operating system has to decide how many and which cores to allocate to which application. Similarly it has to decide on memory region allocations per application. The set of hardware systems, on which an operating system and applications might run, increases over time and therefore the topology, available features and characteristics are not known in advance. Applications, and more importantly the operating system and language runtimes, have to adapt at boot-up and runtime to the underlying hardware. Adapting to hardware means deriving the best allocation policies per application according to application provided requirements. Therefore, the operating system needs a smart way to *reason* about the hardware and derive policies at runtime based on online discovery of hardware features. These challenges require a smart and general way of incorporating online hardware discovery information and application requirements to derive allocation and hardware usage policies.

So far, there has been little work on commodity operating systems to support heterogeneity from the ground up. This thesis explores techniques to support heterogeneity and, furthermore, to deal with the increased complexity caused by heterogeneity.

The following sections describe the various dimensions of diversity and heterogeneity already found in current commodity systems.

1.1.1 Diversity

In this section I define the three dimensions of “diversity” used in this thesis. The classification is important, because in this thesis I explore to what extend the SKB can help the operating system to adapt to the hardware in each dimension. I term the three dimensions *non-uniformity*, *core diversity* and *system diversity*.

Non-uniformity

Non-uniformity traditionally refers to non-uniform memory access (NUMA) for scalable multiprocessing. The classical definition means that memory

regions are grouped into NUMA nodes and a group of cores belongs to one NUMA node. It is still possible to access memory of a different NUMA node, but at a higher latency. The latency of performing an operation on memory depends on the core performing it. The latency is therefore non-uniform and depends on the combination of core and memory address. Table 1.1 in section 1.1.2 shows that the latency differences are significant.

Nowadays the concept of non-uniformity becomes wider. A hierarchy of cache levels where some cores share a certain cache leads to a non-uniform cache architecture and non-uniform access latencies to cached values.
Multiple cores in a system generate an increased number of memory transactions. Obviously, the memory system has to scale with the number of cores. Therefore, most of today's mutli-socket systems are NUMA systems, where a separate memory controller per socket, or even per core, handles memory transactions to a specific NUMA-domain. This leads to fast local memory access, if OS and applications only access local memory.

Caches reduce the number of memory transactions and significantly reduce access latencies. Typical systems today have three levels of caches where the third level is shared by some or all cores of a socket. The cache-sharing property is important when the OS has to decide which threads to place on which cores, especially if the operating system would know from the application, whether threads would benefit from a shared cache or not. Also, as I show in chapter 7, some operations are significantly faster, if the fact, that some cores share a cache, is exploited.

Overall, the memory hierarchy including NUMA-domains, caches and cache-sharing is becoming more complex to handle properly by the OS. Only detailed knowledge allows the full benefit of the memory system's design by the OS and applications to be exploited.

Core diversity

Core diversity refers to the different types of cores within a single system on which a single-image OS will run. Nowadays, most systems still have

uniform cores, but trends are towards having different cores in a single system in terms of power and performance tradeoffs. The ARM big.LITTLE architecture[52] provides four cores of the same instruction set. Two of them are high performance cores and the other two consume low power. The operating system can choose to execute a computation on a high-performance core or on a low-power core. In order to do so, it needs detailed knowledge about the cores, but also about the type of computation. Further, it needs to know, whether the goal at any given moment, is to save power or provide high performance. Instruction set extensions are a another step towards heterogeneity in terms of performance and power tradeoffs[58]. The IBM Cell processor[53] has radically heterogeneous cores. Projects such as HeraJVM[85] and CellVM[99] show how difficult it is to use such a heterogeneous processor. The Intel SCC[62] is itself a homogeneous system in terms of cores, but not in terms of its memory system. An SCC connected to a x86_64 host provides an additional set of x86_32 cores on which the OS can run. Barrelfish runs as a single-image OS on an x86_64 host and the attached SCC[86, 107]. GPGPUs (General-Purpose computation on Graphics Processing Units) is becoming an increasingly hot topic[51]. GPUs are becoming more general-purpose and participate in the computation. CUDA[100] and OpenCL[71] are frameworks which allow the offloading of general-purpose computations to the GPU. The GPU is treated as a device and can only be used by one application at a time. To handle multiple GPUs in the same system, a more advanced task scheduling on GPUs is necessary[125]. The netronom netowrk interface card provides an ARM core on which the OS can run[96]. A while ago, the SunPCi cards[127] provided an x86 based system on a PCI card plugged into a Sun SPARC system. Windows applications run together with the Windows operating system on the SunPCi card. The user interacts with Windows application through a window of the common desktop environment (CDE) or through a separate monitor connected to the SunPCi card. This form of heterogeneity allows users to run applications with different ISA requirements on the same machine, but a single application cannot run partly on both types of processors.

The work in this thesis does currently not place computations according to CPU core features, but it attempts to provide CPU core features (like

floating point capabilities, streaming extensions, power-saving modes) in a high-level abstract and CPU core-independent format, such that future extensions can reason about them at a high-level, without needing to first query every core separately.

System diversity

In contrast to non-uniformity and core diversity, the term *system diversity* refers to the fact that two completely separate systems are diverse, even if they are of the same base architecture, like, for example, x86_64 systems. The number of cores and NUMA regions, the cache hierarchy including cache sharing and the interconnect topology potentially differ significantly between any two systems. Additionally, the available set of devices and accelerators might be significantly different. Therefore, it is impossible to manually tune code for specific machines. Instead, the software, including the operating system, has to adapt to the hardware features in an automated way, such that even future hardware types will be automatically supported. This removes the engineering effort of porting software to future hardware. As I show in chapter 7 and chapter 8, the high-level languages approach allows reasoning about hardware in an abstracted way, such that software automatically adapts to the underlying hardware.

This type of diversity is already present in today's systems. Figure 1.1 shows three different commodity systems available nowadays with completely different interconnects and memory hierarchies.

1.1.2 The interconnect network

Many-core systems often consist of multiple CPU packages which contain multiple cores per package where cores might provide simultaneous multithreading (SMT)[1]. Cores within and between CPU packages communicate over point-to-point links[2], where local communication within the package is much faster than communication between packages. Additionally, the interconnect graph is not a full mesh and therefore multihop-

[1] HyperThreads in AMD

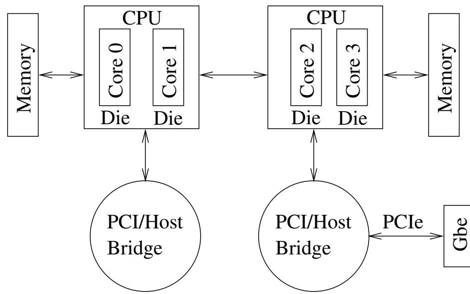
[2] QPI for Intel, HyperTransport for AMD

communication is necessary for non-neighbor cores (see figure 1.1(b) for example). Different latencies and the non-fully connected nature of the interconnect graph form a sort of diversity in terms of latency. With these characteristics, the interonnect becomes a network between cores, caches, memory and devices and is not a bus anymore[18]. In this thesis I argue that consequently, the interconnect has to be treated as such and network-type characteristics have to be part of the knowledge about the system. As chapter 7 shows, treating it as a network and performing the right communication optimizations has significant performance impacts. Of course, to derive the right optimizations, not only network characteristics, but also communication pattern knowledge is important.

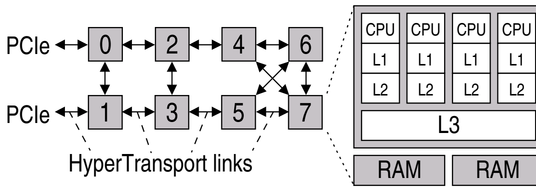
Figure 1.1 shows three examples of commodity systems with different interconnect topologies and different types of cache hierarchies. Figure 1.1(a) is a Tyan Thunder n6650W board with two dual-core AMD Opteron 2220 processors and 8GB RAM across 2 NUMA nodes. They are interconnected by HyperTransport[64] point-to-point links. Figure 1.1(b) is a TyanThunder S4985 board with M4985 daughtercard and 8 quad-core 2GHz AMD Opteron 8350 processors and 16GB RAM across 8 NUMA nodes. The cores are interconnected by HyperTransport[64] point-to-point links. Figure 1.1(c) is an Intel s5000XVN workstation board with two Intel Xeon X5355 quad-core processors with 8GB RAM on 1 NUMA node. The cores and the memory are connected by the front side bus (FSB) which is a traditional non-NUMA topology. Point-to-point communication over HyperTransport[64] or QuickPath[146] links mean that the inside of a general-purpose computer resembles a network with non-uniform messaging latencies due to different numbers of hops and the different routing depending on the source and destination of messages.

As earlier measurements show[118], there is a significant difference in terms of memory access latencies between access of the local and the remote NUMA nodes (see table 1.1). The increased latency on a remote access is a result of crossing the interconnect to reach the remote memory.

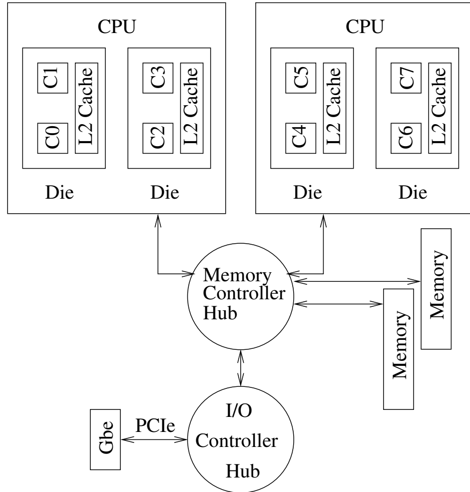
A similar experiment in[18] shows cache access latencies for the sys-



(a) 2x AMD Santa Rosa dual-core processors.



(b) 8x AMD Barcelona quad-core processors.



(c) 2x Intel Clovertown quad-core processors.

Figure 1.1: Different x86-based commodity systems.

Memory region	Core 0	Core 1	Core 2	Core 3
0–2GB	192	192	319	323
2–4GB	192	192	319	323
4–6GB	323	323	191	192
6–8GB	323	323	191	192

Table 1.1: Memory access latencies (in cycles)

Access	cycles	normalized to L1	per-hop cost
L1 cache	2	1	-
L2 cache	15	7.5	-
L3 cache	75	37.5	-
Other L1/L2	130	65	-
1-hop cache	190	95	60
2-hop cache	260	130	70

Table 1.2: Latency of cache access for the PC in Figure 1.1(b).

tem in figure 1.1(b)[3]. The results in table 1.2) show that accessing caches at deeper levels, or even remote caches, have significantly higher latencies. Boyd-Wickizer *et al.* report similar numbers for a 16-core machine[23].

1.1.3 Managing Hardware

Starting at the bottom, the operating system has to discover, enumerate and initialize hardware resources in such a way that it can best be used by the operating system and finally by applications.

Designing hardware with scalability at the hardware level in mind, increases its complexity. Apart from having many cores and NUMA-domains in systems, other hardware components are replicated for similar reasons (scalability with the size of the system). Nowadays, hardware sys-

[3]This experiment has been conducted by Simon Peter

tems have multiple PCIe buses where lots of, increasingly address-space hungry, cards can be plugged-in, even at runtime. Handling address space allocation in PCIe requires a deep understanding of the bus and plugged-in cards, including quirks to apply. Chapter 6 shows how much knowledge about PCIe buses s required in order to correctly configure them. This knowledge has to be created at bootup time, since every system might be different and the bus configuration is not known in advance. Fortunately, the high-level language algorithm, to configure PCIe buses, presented in chapter 6, handles the high complexity of this configuration process well. The increased interrupt load is handled by distributing it through different configurable IOAPIC controllers. Typical systems today have multiple IOAPICs which again have to be initialized and controlled by the operating system. IOAPICs deliver interrupts to specific cores. The destination is configured by the operating system. The destination has to be the core, where the receiver of the interrupt (typically a driver) runs. How to correctly route interrupts is shown in chapter 6. While this chapter discusses how to configure the interrupt hardware with few lines of high-level code, it does not talk about where to run the driver. Deciding where to run a driver depends on its associated device and on which PCIe bus the device is. Each PCIe root bridge is attached via a point-to-point link at the interconnect and therefore is closest to a specific CPU socket. Ideally, the device manager starts the driver on a core close to its associated device. Although the current implementation does not do that, chapter 5 discusses how to decide, where to run the driver based on topology knowledge, in more detail.

The operating system not only needs to adapt to CPUs and memory systems, but also to varying PCIe bus configurations, interrupt controller configurations and many more hardware specifics. Again, this stresses the fact that the operating system needs a reasoning facility to derive correct hardware initializations, which are automatically adapted to the hardware configuration found on the platform. The goal is to reduce the code complexity as much as possible.

1.1.4 Managing Applications

Obviously applications need to be implemented in a multithreaded fashion to benefit from the available hardware parallelism. While this is a challenge by itself, it also imposes requirements on the operating system. First of all, the operating system has to direct applications to create a useful number of threads. I argue in chapter 8 that this requires the operating system to have a global knowledge of the number of cores available, the number of applications and how many threads they wish to run at the same time. Second, the operating system has to place threads on available cores, taking different properties of threads, like, for example, communication between threads, into account.

1.2 Problem Statement and Hypothesis

Hardware is changing fast and getting increasingly diverse. The operating system needs to adapt to the underlying hardware, correctly initialize it and exploit it effectively, even if the hardware's architecture is not known at the operating system's implementation time. Traditional operating systems face significant challenges in adapting to the underlying hardware, because often policies are encoded throughout the operating system's code. This is, however, increasingly problematic. First, it is impossible to encode suitable policies for future, not yet known, hardware platforms. Second, encoding policies throughout the operating system's code increases complexity and makes it harder to port the operating system to future hardware platforms, which, however, will be necessary in order to support them.

This thesis investigates how an operating system can adapt to current and future diverse hardware while keeping the complexity low and portability high. This thesis is guided by the following hypothesis:

> If the operating system had a facility for reasoning about hardware and software, it could better adapt to a large set of diverse hardware, exploit hardware features and configure software modules to improve overall system utilization, while re-

ducing code complexity in both, the operating system and application components.

1.3 Goals

This section defines the main goals of the thesis and lists the main enabling factors to build a facility which allows the operating system to adapt to a wide range of diverse hardware. These goals are important, because they guide design decisions and implementations of the SKB and the use cases. Consequently, the remaining chapters refer to the goals and discuss to what extend they could be achieved. The thesis tries to achieve the goals by building a reasoning facility with the system knowledge base, which can be used by the operating system and by applications.

The main goals are as follows:

- Enable the operating system to adapt to the current underlying hardware
- Reduce code complexity involved in decision taking
- Increase portability to current and future hardware platforms

The main enabling factors are:

- Clear policy/mechanism separation
- High-level declarative language to derive policies
- Central global knowledge processing

1.4 Contributions

The thesis investigates how complexity can be handled by applying high-level declarative language techniques to reason about the underlying hardware. It investigates how a reasoning facility, based on constraint logic

programming, is useful to build an adaptive operating system, which automatically adapts to diverse hardware. Further, the thesis investigates how a reasoning facility helps to built services on top of it, which themselves reduce complexity.

The thesis presents several use cases to prove that reasoning in a high-level declarative language greatly reduces code complexity, both, in policy code and also in the mechanisms. Reasoning about hardware makes the operating system adaptable to the underlying hardware with few lines of code.

The contributions of the thesis are the following:

High-level reasoning facility helps to build adaptive OSs The thesis proves that a high-level reasoning facility using a constraint logic programming language (CLP) is useful to build an operating system, which automatically adapts to the underlying hardware. Such a reasoning facility allows deriving hardware knowledge at runtime and deriving policies with low code complexity, such that the operating system adapts to the underlying hardware. The SKB presented in chapter 3 is the software module used to prove that high-level languages are useful to build an operating system which is adaptive to hardware.

Services benefit from reasoning and further reduce complexity The thesis shows that services built on top of a high-level declarative reasoning facility directly benefit from its logical unification and constraint satisfaction techniques. Services become much simpler to build. They exploit the high-level language, store and process knowledge, and further reduce code complexity in the operating system by taking over functionality which otherwise would be intermingled in individual software modules. Chapter 4 presents a name service and synchronization and coordination services built in this way. Chapter 8 explains a resource allocation framework which builds on logical unification and constraint satisfaction.

Declarative languages reduce hardware configuration complexity The thesis argues and shows, that correct and complete hardware configurations can be derived with few lines of constraint logic programming code. As such, it argues, that this approach is preferable compared to an imperative approach using a low-level language like C. Chapter 6 proves that with the example of PCIe configuration.

Declarative reasoning facilitates adapting to hardware Declarative languages allow deriving policies such that the operating system and applications easily adapt to the underlying hardware. The hardware can be exploited much better, which leads to higher performance. Chapter 7 shows how a hardware-aware algorithm adapts communication within the operating system to the underlying hardware topology in such a way, that communication performance is high and scales well with the number of participants.

1.5 Structure

The rest of the thesis is structured the following way. The background for the thesis is given in chapter 2. The system knowledge base is presented in chapter 3. First, the chapter discusses the design principles and then the implementation. Also, the client interface is explained and examples show how to interact with the SKB. Finally, the SKB is evaluated in terms of code complexity and resource usage. Octopus, the coordination service presented in chapter 4, is an extension of the SKB providing distributed coordination facilities within the operating system.

After presenting the SKB and Octopus, four use-cases demonstrate their usefulness. Chapter 5 shows how to build a device manager on top of the SKB and Octopus. Declarative PCI configuration is explained in chapter 6. Chapter 7 shows how to derive a hardware-aware multicast messaging tree declaratively. Chapter 8 presents a framework and a declarative way to allocate CPU cores to a set of running applications. Finally, the thesis concludes in chapter 9.

Chapter 2

Background

The first part of this chapter discusses declarative techniques and, in particular, provides some deeper background about constraint logic programming, because this thesis builds on this technique. A better understanding of the basic concepts helps understanding design decisions of the SKB, the policy code shown in the use case sections and the interaction with the mechanism code of the operating system.

The second part of this chapter gives an overview of Barrelfish, because its structure and mechanisms are enabling factors to build a reasoning facility to derive policies outside the operating system's mechanisms. Only the parts relevant for this thesis are discussed to provide the necessary operating system background.

Finally, the chapter surveys successful applications of declarative techniques in the operating systems and networks fields.

2.1 Declarative Techniques

As the work of this thesis heavily relies on declarative techniques, this section provides the necessary related background. After an overview of declarative programming in general, the ECL^{i}PS^{e} [10, 30] constraint logic

programming (CLP) system, in which the work in this thesis is implemented, is explained in more details.

2.1.1 What is declarative programming?

Declarative programming is a programming paradigm where the programmer describes *what* he wants, but not *how* to get there[54, 81, 97]. The program describes potential solutions by logic rules without defining the control flow[54, 81]. Complete problems can be described in terms of variables, relations between variables and logic transformation rules to finally achieve a state the programmer would like to get, i.e. the solution to the problem. Typically, problems are described in terms of values, ranges and their dependencies.

Typically, declarative programming eliminates side effects, since the problem can only be described, but no concrete steps to be taken can be defined by the programmer. The concrete steps to be taken are typically defined by the implementation. This allows the implementation to use different techniques like loops or backtracking to search for a solution without exposing the actual technique used to the programmer. It also allows the implementation to change the internal technique, as long as the final solution meets the programmer's expectations. In some cases, the implementation can automatically parallelize the search for solutions, because the control flow is not specified by the programmer and, due to the description, there are no side effects possible[1].

High-level declarative programming allows expressing complex problems in a descriptive way with few lines of code. It reduces the code complexity significantly, while being extremely expressive. This is one of the main reasons to chose declarative programming techniques when dealing with complexity. As long as a complex problem can be described in terms of rules, a declarative language is a good choice.

There are different classes of declarative programming techniques and corresponding languages, which are explained in more detail in the next

[1] In practice, most of the languages allow explicit side effects by providing permanent variables on a heap. This, however, is not the common use of declarative languages.

section.

2.1.2 Declarative languages

This section surveys common declarative programming techniques, which may be suitable in the context of an operating system. As such, it is not a complete list of all declarative programming techniques and languages available, but it helps to understand the reasons why this work builds on constraint logic programming.

The paradigm "declarative programming" includes a range of sub-paradigms where each of which has a number of languages or programming systems.

Logic programming is a well-known form of declarative programming. Prolog[24, 36, 120, 130], as the programming system for logic programming, allows the programmer to describe a problem in terms of information, variables and logical unification rules. The goal is to reason about information and derive knowledge in a specific context. Facts store known pieces of information inside the Prolog runtime. The facts can be accessed by each rule during its complete execution. Variables are unified to constants or facts and possibly to other variables until a solution can be found such that all requirements on all variables are met, or the system recognizes that there is no solution to the problem. Internally, Prolog makes extensive use of backtracking to search the complete search space. Whenever it has to chose a value to assign to a specific variable, it creates a choice point on the stack and follows down a branch of the search tree. It will either output a solution, if there is one, and then backtrack to the choice point or, if there is no solution in the subtree, it will backtrack to the choice point immediately. At every choice point it assigns a new value to the variable and tries another subtree of the search space[2]. Obviously, when the search space is huge, Prolog needs to create a large number of choice points, which will make the search time consuming (or in fact slow). Datalog has its roots in logic programming and is similar to Prolog. Compared to Prolog, there are a number of restrictions in terms of allowed argument complexity and binding of variables, for example. Datalog was designed originally for declarative databases[28].

Constraint programming allows the description of a problem in terms of variables and constraints. Constraints relate variables to a range of possible values which the system is allowed to assign to them. This includes restricting a variable to a given set of constant values, but also applying constraints which relate two (or more) variables to each other. Constraints between two (or more) variables create dependencies between them. The dependencies can be created even before any of the variables has a concrete value assigned, which is an important feature on which the algorithms in this thesis rely. An example is a variable whose value has to be greater than the value of some other variable. The constraint is applied to the two variables, before concrete values are known. The solver takes all variables and all constraints into account and only assigns values such that all constraints are met. If there is a solution, it outputs all possible assignments to all variables. Otherwise, it outputs, that there is no solution to the problem.

In *functional programming* the programmer defines *how* a goal should be reached, by defining a sequence of functions to apply to a given input. Functional programming typically has no side-effects. Functions purely operate on input parameters and return the function's output. There is not global state which gets modified by any function during execution[2]. Because there are typically no side effects, the compiler has freedom for radical optimizations, including parallelizing parts of the execution without the programmer having to know about it. In fact, many tools in Barrelfish are implemented in Haskell[57], a purely functional language. For this thesis, functional languages are less appropriate, because the thesis' goal is to reason about information and deriving knowledge, rather than applying functions on information in a well-known order.

2.1.3 Constraint logic programming

This section explains why constraint logic programming is suitable to hardware configuration and allocation problems. It lays out why the work

[2] In practice, functional languages offer means to store global variables, if really necessary.

in this thesis is based on this technique.

Constraint logic programming (CLP) unifies constraint programming and logic programming. A CLP system allows the use of logical unification in combination with constraints applied to some of the variables. The logic unification rules prepare the necessary knowledge about the problem to be solved, and the constraints define ranges of valid solutions to the problem.

Programs in CLP are formalized in terms of free variables, facts, logical rules, and constraints. Free variables can be unified to other variables or to stored facts. In CLP, free variables can also be constrained to certain ranges of values assignable to a variable. Constraints indirectly influence the unification process, because only some values can be assigned to a constrained variable. If the unification process tries to assign a value outside the range, it fails and triggers a backtrack. To search a valid solution, the solver enumerates possible values and temporarily unifies a variable to them, until it finds one. Backtracking is expensive, as for regular Prolog programs. The programmer might chose to reduce the number of choice points to limit the number of backtracks performed by the system. This can lead to a much lower execution time.

As for regular constraint programming, variables in CLP can be used and constrained even before concrete values are assigned to them. It is possible to express calculations based on variables with no values yet assigned and constraining the result of the calculation to a range of values. The solver will then search concrete values such that the calculation leads to a result in the given constrained range. Of course, also the result range might come from a calculation of other variables.

When implementing a CLP program, the programmer defines a set of variables and logical unification rules which unify them to stored facts. This step provides the necessary knowledge about the problem. Second, he constrains the variables (possibly just by relating them somehow) to define valid solutions of the problem. Then, the programmer causes the program to invoke the solver which produces valid solutions in the requested ranges and based on knowledge coming from stored facts and unification.

CLP programming forms an ideal basis for hardware resource allocations. Logical unification rules derive hardware knowledge and con-

straints relate dependencies between pieces of hardware. By understanding hardware properties through the logical reasoning and by relating several pieces of hardware in terms of constraints in variables representing the hardware, the solver is able to find an overall valid resource allocation. The PCIe bus driver in chapter 6 makes extensive use of this technique. The multicast tree construction in chapter 7 and the global resource allocation in chapter 8 also make use of logical reasoning and constraint solving.

2.1.4 CLP programming in ECLiPSe

This section introduces the ECLiPSe CLP programming system, as the work in this thesis is based on it. The three phases described here need to be followed exactly, otherwise ECLiPSe might behave in an unexpected way. The technical report on developing applications with ECLiPSe explains in detail how to correctly develop applications[121].

ECLiPSe is a Prolog-based CLP system with constraints extensions. It implements an extended version of the "Warren abstract machine" (WAM)[2]. Facts are stored on the heap. Code can be uploaded to the system in source form or can be precompiled byte code. On the first execution, the source code or byte code gets compiled to machine code.

CLP programs in ECLiPSe follow three phases. First of all, the appropriate data structure with necessary variables should be constructed in such a way, that the data structure models the problem in a natural way. This includes rules to match variables with stored facts. In a second step, constraints should be applied to the variables. During this step, no backtracking should be performed, as that would cause the system to create new variables. While this is fine for pure Prolog, where facts are unified to the new variables, it does not work for ECLiPSe programs. Constraints are attached to concrete variables and do not get attached automatically to newly created variables by the system, even if the new variable logically holds the same value. Therefore, a programmer has to be careful about that. In the last step, a rule has to invoke the solver and tell it which variables should be instantiated with concrete values. This means, that the solver enumerates the passed set of variables with the valid range of val-

ues according to the constraints. Potentially, this trigger backtracks until a solution can be found. During that phase, the logical unification rules provide the necessary knowledge such that the solver can relate variables to each other and to stored facts.

As soon as the system finds a solution, it outputs it and stops searching for further ones. The choice points, however, remain and the caller has the option to trigger the solver to search for further solutions. Alternatively, all solutions can be searched by using the goal `findall/3`. This produces a list of all possible solutions. The risk is, that this takes a lot of time as the search space might be large and the number of valid solutions might be huge.

2.2 Barrelfish

Barrelfish has the right operating system structure to run on a large, possibly non-coherent, heterogeneous hardware system. Its structure further allows deriving policy parameters outside the core operating system code, a property which is important for the work in this thesis. The thesis therefore uses Barrelfish to evaluate the declarative language approach to make an operating system adaptable to the underlying hardware. The declarative reasoning algorithms in this thesis derive policy parameters which can directly be used in Barrelfish's mechanisms. The techniques presented in this thesis are however not bound to Barrelfish. They can be used in other operating systems in a similar way if the mechanisms of the operating system allow using policy parameters derived outside the mechanism code.

Because the thesis uses Barrelfish to evaluate the declarative language approach for reasoning about hardware and for adapting to it, this section explains Barrelfish's structure and its most important properties relied on by this thesis.

Barrelfish is a new operating system for heterogeneous many-core systems written from scratch. It implements a new OS architecture, the *multikernel*[17], presented in the next section.

2.2.1 The Multikernel

The multikernel[17] is a new OS architecture designed for modern and future heterogeneous many-core systems. It is structured as a distributed system where one operating system node runs on one specific CPU core. This structure naturally matches the underlying hardware, which increasingly resembles a network[18]. Furthermore, it naturally supports hardware heterogeneity, as every core runs a separate operating system node (see section 2.2.2).

The multkernel is guided by the following three design principles:

- Make all inter-core communication explicit
- Make OS structure hardware-neutral
- View state as replicated instead of shared

These three design principles allow structuring an OS in a way that it naturally supports hardware heterogeneity, scalability and the ability to adopt distributed systems principles to improve performance and interconnect usage. The multikernel builds the foundation of running on heterogeneous hardware. It provides mechanisms to execute tasks on the available CPU cores, but it does not decide by itself, on which core a task should be executed. The decision has to be made outside the operating system. This is one feature on which the work in this thesis relies.

Figure 2.1 shows the multikernel model. The three design principles are explained in more detail in the following sections.

Make all inter-core communication explicit

The multikernel makes all communication explicit. State is kept completely local and no memory is shared between code executed on different cores. Explicit messaging facilitates reasoning about the interconnect usage. The knowledge of *who* is accessing *what* parts of states and *when* it is accessing the state, is exposed. In contrast, a shared-memory-based

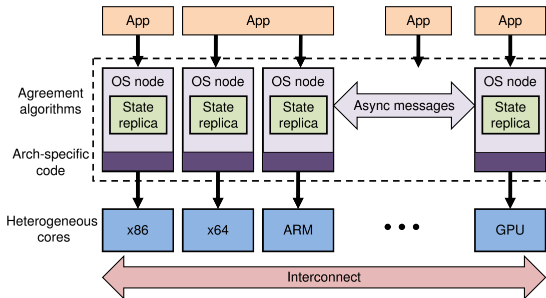


Figure 2.1: The multikernel model

system with implicit messaging, such as the shared memory itself or the cache coherence messages, does not have this explicit knowledge.

The explicit knowledge of when messages are sent over the interconnect allows creating efficient communication primitives by deriving policies defining from which source to which destination messages should be sent. Because it is explicit, messages are only sent as requested by the policies. Chapter 7 makes use of this explicit communication between cores.

Make OS structure hardware-neutral

The multikernel is structured such that most of the operating system is separated from the hardware. Only two aspects are specific to a target architecture. The first aspect is the message transport mechanism and the second aspect is the interface to devices and CPUs. Having only these two aspects hardware dependent, has a number of advantages.

Running the OS on a different architecture with different characteristics in terms of performance or hardware interface, including message transport mechanism, there is no radical code change necessary to make the system work well. Messaging can be implemented as user-level RPC or on hardware message facilities, in case the hardware supports this feature. The higher-level interface to the messaging system does not need to

change, and especially the OS structure does not change, if a new message transport is used.

Moving to a completely new architecture requires modifying drivers according to the new hardware interface. In the multikernel model, CPUs are treated as devices and their device driver is a small kernel, called the *CPU driver* (see section 2.2.2). Therefore, it is sufficient to exchange all drivers and ensure that the appropriate message transport is used to make the system run on a new target architecture.

In the future, these benefits are increasingly important as diversity in hardware is likely to grow, which makes it impossible to radically restructure the OS on every deployment. A multikernel is prepared to easily adapt to diverse hardware.

View state as replicated instead of shared

By keeping all state local to every core, there is no shared state at all. Decisions are based on local data structures and updates are performed on the same local data structures. In case that multiple cores have to coordinate and maintain a global view, messages are exchanged between them to update the same piece of information in the respective local data structures. By making replication a part of the multikernel, heterogeneity support comes naturally. Cores with different endiannesses, for example, can communicate by messages and do not need to take care of the different endianness, as no sharing occurs. A second advantage is that cores can be hotplugged or shut down to save power without complicating the maintenance of shared data structures. A limited amount of sharing, for example between cores on the same package, could be seen as a local optimization of replication.

2.2.2 A Barrelfish "node"

This section briefly describes Barrelfish's implementation of the multikernel, because the thesis is about heterogeneity support and builds on the concrete structure of Barrelfish. Further, the hardware discovery process,

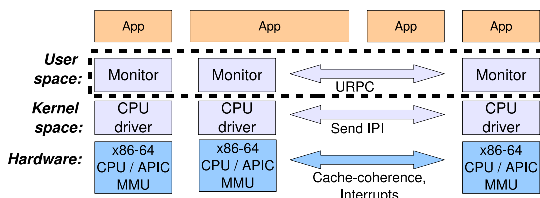


Figure 2.2: The structure of Barrelfish

described in section 5, requires a basic understanding of the actual implementation of a Barrelfish node.

Barrelfish is an implementation of the multikernel model. As such, it runs one separate operating systems node per core without sharing any memory between nodes. The implementation of an operating system consists of two parts. There is the *CPU driver* running in supervisor mode and on top of it, there is the *monitor* running in user mode. Figure 2.2 shows the structure of Barrelfish.

The *CPU driver* is capable of executing privileged instructions. Traditionally, it would be called kernel, but in Barrelfish, a CPU core is treated as any other device and therefore the kernel is the driver of a CPU core. The CPU driver offers a small number of system calls, for example to map a physical memory page into an application's virtual address space. The CPU driver checks first, whether the right to perform the operation has previously been granted to the application. All state is kept purely local to the core, meaning that no memory is being shared with other CPU drivers. Also, the CPU driver does not perform any communication to remote cores.

The *monitor* is the user-level part of a Barrelfish node. It is responsible for maintaining a consistent view of the whole operating system. It does not share any memory with other cores, but instead uses messages to synchronize state among the cores. The monitor offers additional functionality to applications. Applications can ask monitors to forward messages to applications on other cores. Finally, the monitor is able to send capabilities to a remote core.

Because neither the CPU driver nor the monitor share memory with other cores, Barrelfish does not rely on cache coherence. It only needs a way to send messages from one core to another. This may be implemented on shared memory, but hardware messaging features may also be used. As such, Barrelfish is ready to run on future hardware, which is potentially not fully cache-coherent.

2.2.3 Explicit access to physical resources

This section defines the meaning of explicit access to physical resources as used in the thesis. It explains the physical resources for which the thesis implemented policy code and to which the use-cases need explicit access.

Explicit access to physical resources means that the operating system does not enforce any policies and does not hide the resource behind any layer. There is no translation of any form and no interpretation of the resource by the operating system's mechanisms. Policies are pushed into application domains like in the exokernel approach[40]. This however does not mean, that there is no protection. Protection is guaranteed through mechanisms at the operating system's level and through hardware support (for example in the case of memory). The resource requester should decide which resource, or which part of it, would best suit its needs. Once access to a physical resource has been granted by the controlling mechanism, it is the responsibility of the current resource holder to decide how the resource should be used. The following paragraphs describe each physical resource directly relevant for this thesis.

The multikernel, in some sense, gives explicit access to CPU cores. The local CPU driver invokes an upcall interface in user-space, which decides which computation to execute. This is the mechanism used in Psyche[84] and scheduler activations[7]. It allows executing code in user-mode on the specific requested core. Computations are not migrated automatically and there is no automatic decision process about placing computations on cores. The only policy enforced is basic time-partitioned scheduling. This makes the operating system code much simpler, as all the complexity involved in taking placing decisions are removed and pushed to a separate reasoning facility.

Access to physical address spaces (including memory, non-volatile platform data and memory-mapped devices) is controlled by a *capability system* similar to the one of seL4[39, 72]. The *memory server* in the operating system manages capabilities. Applications can ask for certain capabilities (used to allocate memory). Managing capabilities is a mechanism which does not decide by itself which ones to return on a request. External policy code needs to decide which capabilities would best suit the applications' needs.

An application, which holds a capability to a range of memory, can ask the CPU driver (see section 2.2.2) to map it at a chosen free virtual address in its address space, providing it access to the memory. NUMA-aware allocation means getting a capability pointing to a page within a given physical address range. It is a policy parameter which instructs the memory server to return capabilities for a given range. The policy needs to be derived outside the memory server, making the memory server code much simpler.

Likewise, an application, which holds a capability to a memory-mapped device, can also ask the CPU driver to map it at any free virtual address in its address space. This gives it access to the device. Device drivers get safe access to devices using this mechanism.

If an application holds a capability to a device bus (such as the PCIe bus), it can get access to the configuration registers of all devices. As chapter 6 will show, this allows the PCIe bus driver to configure base addresses for PCIe devices and bridges.

Only implementing mechanisms and relying on policy code outside the mechanisms is the right step towards simpler and cleaner operating systems code and fits well with the goals of this thesis.

2.2.4 Messaging

This section discusses how communication is done in Barrelfish. While most of the code in this thesis just relies on having messaging, the multicast tree construction in chapter 7 directly interacts with the messaging mechanism and therefore requires some high-level knowledge about messaging.

Barrelfish provides mechanisms to create message channels between every pair of cores. Two ways of creating a message channel are possible. First, an application can create a message channel and listen on it for incoming messages. This type of channel allows building operating system services. Second, an application can connect to a message channel offered by another application. It can start sending commands to the other application. Message channels are bidirectional. Once they are set up, both ends can send messages to each other.

The mechanism of creating message channels does not impose any restrictions on the number of channels and the source and destinations. Consequently, it is possible to create several message channels between every pair of cores. The number of channels grows quickly, which is not desired. Chapter 7 argues that there is a routing problem within the machine. With the example of multicast messaging, it shows how to decide on the number and sources and destinations of message channels to decrease latency and also the number of channels necessary.

As with other resources, Barrelfish gives explicit access to the messaging mechanisms. This provides a lot of freedom in creating message channels and makes the message mechanisms much simpler, as no policy code is intermingled with it. It builds the basis of reasoning about channel creation and routing of messages in a separate facility.

2.2.5 Drivers and services

This section explains how operating system services and drivers are being built in Barrelfish and how they export their service to the rest of the system. This section presents only the mechanisms of exporting a service, but does not talk about coordination. However, chapter 4 argues, that a clean coordination is necessary and that the complexity should be taken out of the actual service and driver code.

All services and drivers in Barrelfish run in user-mode, as in a traditional microkernel. Services create a message channel on which they listen for incoming commands. They register the message channel by a name and with a name server. The name server has a well-known message channel which can directly be used for registration and lookups of other

services.

Basic operating system services have well-known names. Applications use these names to look up the service's message channel to start using the operating system's services.

Drivers are services which export a device's functionality to the rest of the system. They get physical access to a device and are responsible for initializing and operating it correctly. Drivers are able to send and receive data from devices. Every driver creates a message channel (as every other service) on which it waits for commands to be executed on the device. Every driver registers the channel by a name with the name server.

Bus drivers get access to the configuration space of the bus. They manage resource allocations for all devices within the bus and are responsible for granting safe access to a specific device. Bus drivers export a message channel by name on which they listen for incoming commands. Device drivers look up the bus driver's message channel, connect to it and ask for access to a specific device.

Starting services and drivers in the right order and resolving dependencies is important for the correct functioning of the system. Section 5 explains, how services and drivers are coordinated in the distributed nature of Barrelfish.

2.3 Reasoning in operating systems

The section summarizes related work in terms of hardware representation, hardware configuration, resource allocation and reasoning about resource.

2.3.1 Hardware representation

Operating systems need to manage hardware and have to deal with the increased complexity. A representation of hardware is therefore an inevitable requirement. The representation of hardware is done at different layers of the complete software stack, depending on the actual system architecture. Also, the extent of how much information (hardware and policies) is exported to user space depends on the system.

Traditionally, operating systems tried to abstract resources. Concrete low-level details were kept in the OS and an abstract API was exported to applications. Together with abstracting resources, the OS applied resource allocation policies without negotiating with applications. Within the OS there is however a long history of policy/mechanism separation. For example, Hydra[79] applied policy/mechanism separation as an important design principle.

Commodity OSs and platform firmware increasingly export at least parts of the resource knowledge to user-space.

Linux exports hardware knowledge through the `sysfs` file system[94], and the `proc` file system. The SKB allows simple hardware queries, similar to reading the text files in the `sysfs` or the `proc` file system. Additionally to reading out information, the CLP-based approach provides a much more powerful interface allowing its clients to reason about the hardware in a single query.

Windows exports hardware knowledge in its registry[122]. The registry is a key-value store which can be queried (and updated) by services and applications. The query language is however not designed to unify different pieces of hardware knowledge in the same query as CLP would allow.

There are a few examples of rich, high-level descriptions of heterogeneous hardware resources at the platform level. In particular, the ACPI and EFI standards have an explicit representation of many board-level resources, and the CIM standard[37] defines a schema for a description of higher-level resources. It is easy and convenient to inject such representations into the SKB with the goal of having all information in one place in a uniform way.

2.3.2 Declarative hardware access and configuration

At the lowest software level, drivers access registers with typically complex bit field patterns. Declarative languages reduce the complexity of these accesses significantly and reduce errors due to wrong accesses in drivers.

Devil[87], an IDL for hardware programming, uses a declarative specification of device ports (base addresses), registers, and their interpretation to generate low-level code for device access. This leads to simpler and more understandable code for device drivers, in an attempt to improve driver reliability. ATARE[69] uses a series of regular expressions to extract IRQ routing information from ACPI, without the need for the usual complex byte code interpreter.

Singularity[123] uses XML manifests to reason about the resources used by a device driver. These manifests may be analyzed at driver install time to check for resource conflicts. They also ensure the correctness of a driver's interaction with the OS through contracts on message channels.

Prolog has been used in commercial systems such as Windows NT[61] to derive network configurations: a backtrack-based binding algorithm takes facts about interfaces of network modules and derives valid configurations, including the correct load order of modules, which it then stores to the registry. DEC developed a series of expert systems to ensure that selected component configurations, that include CPUs and other hardware as well as software, are valid and components are compatible to each other[14]. Hippodrome uses a solver to automatically configure minimal and still performant storage systems by analyzing workloads and iteratively searching a global minimum[6].

2.3.3 Resource allocation

Resource allocation is a core functionality of every operating system. While traditionally the policies have been hidden from applications, there were some attempts to provide information about internal state to applications. Further, some systems have extensions such that applications can reserve resources. More recently, systems started to use declarative reasoning to allocate resources according to applications needs. This section gives an overview of some related systems.

Infokernel[11] stresses the importance of providing detailed information to user space. An Infokernel exports general abstractions describing internal kernel state to user-space applications to allow them to reason online about the system's state and internal policies used and to build more

sophisticated policies on-top of kernel policies to direct those kernel policies in various ways.

The Resource Kernel[102] is a loadable kernel module which interacts with the host kernel and allows the applications to reserve system resources which are then guaranteed. The module runs completely in kernel mode and is designed to run together with the host kernel. The main goal of this work is to satisfy the reservations made by applications on system resources.

The Q-RAM[109] project is designed to satisfy minimum resource constraints and furthermore to optimize a utility function to allocate more resources to applications than minimally required, if available. If an application gets more resources than the minimum specified, it adapts itself to provide better QoS. Searching for optimal solutions is a hard problem[110] in practice, and quickly becomes infeasible with many applications. Furthermore, the utility function must be statically specified by the programmer.

Declarative techniques have also been used successfully to specify resources and resource requirements. Condor[80, 131] allocates resources in a distributed system from several nodes to distributed computations. Resources and resource requirements are specified using a declarative approach. A matching algorithm matches the resource requirement descriptions to actual available resources and derives an allocation from resources to tasks.

Helios[98] tackles heterogeneity by running satellite kernels on heterogeneous cores. Satellite kernels are light-weight runtimes which run on peripherals and provide a limited set of functionality. System calls implemented in the coordinator kernel are executed on the host PC. Helios needs to decide where to run functionality. Is uses manifests which declaratively define positive or negative affinities on a per channel basis to guide the placement of processes to CPUs in a heterogeneous system. The affinities define, whether the application benefits from zero-copy messaging or whether it prefers to avoid any interference. Using the affinity manifest, Helios places the application by considering the affinity manifest and hardware utilization.

The Hydra framework[139] uses a declarative approach to reason about

available hardware resources in a heterogeneous system consisting of CPUs and programmable offload devices on which tasks can be executed. Using an XML-based description language, the Hydra framework selects suitable devices to which it places executions of functionality, thus achieving greater utilization of processor resources while reducing the complexity for the programmer.

2.4 Declarative reasoning in networks

Declarative reasoning in networks is related to the work in this thesis, because first, Barrelfish has a distributed-systems like structure and second, the multicast tree construction presented in chapter 7 reasons about the network-like hardware and constructs to construct a multicast tree. The section therefore summarizes some related work of the networking field.

Rhizoma is an overlay which deploys distributed applications to a set of nodes in the internet[142]. It declaratively reasons about network links and offered features of the nodes (such as CPU, memory or available disk space). It tries to satisfy application requirements, which are also given in a high-level declarative way. Based on hardware knowledge (nodes, links) and application requirements it decides on the number of nodes and locations to use for deployment.

COOLAID is a system which declaratively manages network configurations in the increasingly difficult to manage large networks[29]. It captures knowledge from device vendors and service providers as well as online status information in a formal and uniform way. Declarative queries allow deriving valid network configurations and support network operators to run a large network.

Declarative routing allows the implementation of various routing protocols with few lines and therefore low code complexity[82]. This is a step towards easier deployment of new routing protocols, which makes the overall network more extensible while still guaranteeing robustness. Somewhat similar, but in a completely different environment, a declarative query constructs the multicast tree in chapter 7.

In the context of the Semantic Web, the resource description format

(RDF)[138] is widely used to represent and reason online about resources. RDF is a model to describe resources in a machine readable way and was originally designed for the Web. It extends the linking structure by named relationships in order to support automated reasoning about the content of semi-structured data. RDF is expressively almost equivalent to the logic programming approach (ignoring the constraint and optimization extensions).

2.5 Summary

In this chapter I explained that declarative languages can reduce the code complexity involved in reasoning about complex hardware in order to decide how to adapt to it. This makes declarative languages interesting in an operating system targeting heterogeneous hardware. Policy code, which decides how to adapt to hardware, can be implemented at a high level by describing the desired solution to be achieved.

Especially constraint logic programming is interesting in that context. First, the logical reasoning allows deriving knowledge about the hardware. Second, the constraint solver helps to model resource allocation algorithms, because resource allocation often means relating resource requirements to each other. This can easily be modeled as constraints.

To evaluate, whether this technique works, I use Barrelfish, because its mechanisms' behavior can be directly influenced by derived policy parameters.

The next chapter explains the design and implementation of the SKB, the reasoning facility used for this thesis.

Chapter 3

The system knowledge base

The previous chapter claimed, that high-level declarative languages are suitable to deal with the complexity involved in operating systems that are adaptive to heterogeneous hardware. To evaluate this, I built a real reasoning engine for Barrelfish, on top of which case studies prove the claim.

This chapter presents the system knowledge base (SKB). It is the main facility for reasoning about hardware and software state in Barrelfish. After a short introduction, the chapter defines how the term *knowledge* is used in this thesis and gives some motivating examples of knowledge processing in an operating system. It further presents the design principles and and the implementation, before explaining how the operating system and applications can use the SKB. Finally, the chapter presents an evaluation of this main facility.

The system knowledge base (SKB) is a user-level OS service which provides a rich representation of the hardware in a high-level declarative way[118]. The goal is enabling system services and applications to adapt the currently underlying hardware by incorporating deep hardware knowledge to improve performance and optimizing resource consumption by using devices and resources in an appropriate way.

3.1 Introduction

As stated in section 1.1, hardware becomes increasingly complex and diverse. It is essential that system software and applications automatically adapt to the underlying hardware. Manual tuning of software for a specific hardware system is not possible anymore. Instead, the operating system has to learn about the underlying hardware. It has to gather information about the hardware during runtime first and then it has to reason about it and to adapt to it according to the gathered information. Because the hardware becomes more complex, the operating system has to learn as many details as possible about every single device and all connections between several devices. This leads to a big amount of data which has to be interpreted in an accurate way. Obviously, the complexity of interpreting fine-grained detailed data in many different contexts is a complex task. Ideally, the complexity should not be repeated in every system component and especially not in every application. There should rather be a service which transforms data into context-specific knowledge. Clients of this service should be able to ask high-level questions and should get the desired knowledge in response. This alleviates applications from the burden of interpreting low-level data themselves.

Although most of the complexity can be pushed to such a service, it is important to keep the code complexity as small as possible also in the service itself. Readability and maintainability of code with low complexity is much higher and results in fewer bugs. Basing the service on high-level declarative facts representing information in a specific context and running high-level declarative algorithms describing a desired solution enable programmers to write concise, understandable and maintainable code. As described in section 2.1.3, CLP allows writing high-level declarative code. Rules can be formulated based on stored facts, on variables and on constraints between variables leading to a description of the problem or actually of the desired solution. Due to this reason, CLP is the programming paradigm of choice for this thesis.

This chapter presents the system knowledge base (SKB), the central OS service responsible for storing and managing hardware knowledge and for executing reasoning algorithms. The SKB allows adding knowledge in

a high-level declarative way. Furthermore, it allows the uploading and executing of declarative algorithms based on stored facts and additional input parameters.

The SKB's architecture allows future extensions. It is not at all possible to include all the policy or knowledge code from the beginning in the SKB. In the future, new hardware types and new devices will appear. Additional facts need to be added by external modules and new declarative algorithms need to be added as well. The SKB must allow external modules to provide further facts, knowledge or policy code for specific scenarios. By choosing an architecture where the core provides the basic infrastructure to add facts, upload policy code and allow for querying, external modules can be built around it and add their own facts and policy code, which again will be made available for the rest of the system.

This chapter explains the SKB's architecture and design principles. It further describes how clients add facts and how they upload and execute algorithms in the SKB. The chapter also discusses advantages and disadvantages of this approach.

Parts of this chapter have been published[116, 117, 118]. A tutorial describing how to get a basic application running with the SKB can be found on the public Barrelfish wiki[16].

3.2 Background

This section defines the term *knowledge* used in the context of this thesis. The term *knowledge* appears throughout the whole thesis and is a key point of it. After the definition, the section gives an overview of different types of knowledge bases to put the system knowledge base in context.

3.2.1 Knowledge

Data, information and knowledge

The realm of knowledge in knowledge engineering covers a range of specific concepts[97]. In the context of this thesis, knowledge refers to three

of these concepts: knowledge storing (knowledge base), knowledge representation and knowledge application (reasoning)[97]. Knowledge is based on information which itself is based on data. Niederliński defines the three terms *data*, *information* and *knowledge* in the following way[97]:

- *data* is given by 0/1 vectors. These vectors represent numbers, letters, signs or more complex structures including pictures or sound. Data vectors are typically classified in data types such as bytes, chars, floating point values, arrays or structures. Data is a result of measurements, human interactions or processing of existing data.

- *Information = data + meaning of data + purpose of data*. Information is therefore a purpose-oriented set of meaningful data. Information is stored in some form of databases. It appears as a result of some target-oriented action.

- *Knowledge = Information + goal + ability to use information to achieve goal*. Knowledge refers therefore to information relevant to some goal and the ability to process the information in a way to achieve the goal. Knowledge is represented by a set of facts, rules and mathematical models.

Knowledge in the system knowledge base

In this thesis, *data* is often a result hardware discovery by a data gathering process running in the OS. It queries the underlying hardware and gets out a lot of data, which it stores in the system knowledge base and which has to be interpreted and used to achieve specific goals later on.

Information in the context of this thesis is typically a result of some driver which understands the meaning of the data belonging to a specific device. The driver knows how to use this data and therefore there is a purpose for the data in the context of the driver.

To reason about the hardware and take smart decisions, declarative algorithms are implemented based on information stored in the system knowledge base. The purpose of the algorithms is to fulfill a specific goal based on the information. It knows how to use the information to reach the

goal. The algorithms are typically executed by specific drivers, resource managers or other operating system modules. For example, drivers have a specific goal, namely to correctly initialize and operate a device. This includes correct hardware resource allocation. Drivers, and especially the algorithms used by them, gain *knowledge* about hardware by using information and transformation rules to achieve the goal. One of the most complex representatives is the PCI driver (see chapter 6). Its goal is to allocate conflict-free physical addresses to all devices while meeting complex hardware requirements. It uses PCI information and knows how PCI allocation works.

3.2.2 Knowledge bases

This section shows that there are different interpretations of the term *knowledge base* possible. Because the thesis implements a knowledge base, it is important to define what type of knowledge base the thesis refers to.

The purpose of a knowledge base is storing, organizing and managing knowledge. An interface allows clients to query the knowledge base and retrieve answers to specific questions. The two main types of knowledge bases are *human-readable knowledge bases* and *machine-readable knowledge bases*.

Human-readable knowledge bases provide information to users in form of text, tables or figures. A human may search for specific keywords or may follow a predefined structure of categories until he reaches the knowledge item of interest. "Frequently asked questions" (FAQs) pages are an example of human-readable knowledge bases. Users read through FAQs and try to match their question with the answers given in the FAQs. Organizations might provide human-readable knowledge-bases in their intranet, such that users learn about infrastructure, for example. A knowledge base of this form might be organized as a hierarchy of categories. The user selects one of the top categories according to what he needs to learn or lookup and follows down a tree which allows him to select increasingly fine-grained subcategories until he finds the item of interest. Another form of human-readable knowledge-bases are used to support users using a company's product. The Microsoft knowledge base[92] is a human-

readable knowledge base providing specific technical knowledge to users of Microsoft's products. A user can search through the knowledge base to find desired entries. A single webpage contains information regarding the searched keywords. The user reads through the page to get an answer to his question. Similarly, Apple[9], Mozilla[95] and many more provide knowledge bases to support their users in many scenarios.

Machine-readable knowledge bases provide knowledge in a form, that a machine can reason automatically about and take decisions based on this knowledge. Classical deductive reasoning can be used to start from a set of given facts to reach a logical conclusion. Rules define how facts can be transformed and combined such that the logical conclusion can be reached. Machine-readable knowledge-bases are often used in artificial intelligence to reason about available facts and to take decisions based on them. Expert systems[14] take decisions based on facts like a human expert. They consist of a machine-readable knowledge base and a reasoning engine. The reasoning engine reads the machine-readable facts, applies rules as described by a programmer and deduces new knowledge. There are systems which use machine-readable knowledge bases and reasoning engines to decide on further steps to be taken based on current knowledge. Rhizoma decides how many and which servers it needs to acquire from the network to reliably run an application meeting the user's requirements on latency, connectivity and availability[141, 142]. The Microsoft Registry[122] contains machine-readable information about hardware, software and various configurations. While the Registry does not directly contain a reasoning engine, the operating system has the possibility to read facts identified by keys from the registry.

The system knowledge base in this thesis is clearly a machine-readable knowledge base. Algorithms rely on machine-readable information and automatically derive new knowledge by applying transformation rules and unification.

3.3 How does the SKB help the operating system?

It is necessary to know the exact role of the SKB in the operating system. This section defines the SKB's purpose followed by some supporting examples. It further lists common patterns found in problems, for which the SKB is a suitable facility to solve these problems. Finally, the section provides a guideline according to which one can decide whether it is worth to model a concrete problem in the SKB.

3.3.1 Purpose

The purpose of the SKB is providing a general storage for high-level declarative facts as well as an execution environment to execute declarative algorithms. The goal of the SKB is to serve as a central point where the OS, drivers and applications collect all information which might be interesting not only for themselves, but also for other modules. The SKB provides a uniform and standardized way of querying hardware information and software state to OS components and applications. By using high-level facts, services and applications do not need to know specific details about how to get access to information registers of devices. They also do not need to worry about how to interpret hardware information, which is usually provided as bit fields in registers. High-level facts provide the information of interest in a register-layout-independent way. Facts are therefore easy to read by machines, and even by humans, and they always have the same format, independently of how the hardware manufacturer decided to expose the information on the particular piece of hardware by registers.

Another purpose of the SKB is forming a basis to build reasoning algorithms. These algorithms describe a higher-level problem based on stored facts. Additionally to stored facts, parameters can be passed to algorithms. Parameters are the better choice over stored facts, if their values change quickly. On top of stored facts, rules combine several facts to produce new, high-level knowledge. This new knowledge can be further processed by higher-level reasoning algorithms which describe a complete problem.

Generally, the reasoning algorithms enable system software and applications to take informed decisions on how to make best use of available hardware resources.

3.3.2 Examples

So far, the description of the SKB and its purpose was rather abstract and generic. In this section, I sketch some basic examples of information which goes into the SKB.

The cache is an important part of the hardware which significantly affects software's performance, depending on whether it is used the right way or not. Cache information is provided by the SKB as high-level facts, which means that properties such as cache size, cache line size, level or associativity can be queried independently of the actual CPU architecture. It is the SKB's responsibility (together with its datagathering services described in section 3.5.3) to get the information using the appropriate low-level mechanism. On x86 CPUs, the `cpuid` instruction provides cache information in an encoded way, while on other architectures there are other low-level mechanisms to gather low-level data or the information might even come from online measurements instead of information registers. At the end it does not matter to the clients, how the cache facts were produced, as long as the client can query the SKB for cache line sizes, associativity and other properties important to the application in a uniform and abstracted way. The schema in section 3.5.2 shows (among other fact formats) the concrete representation of cache information.

Section 3.3.1 mentions that quickly changing values should rather be input parameters to reasoning algorithms instead of stored facts. The current CPU utilization is one example of information which changes quickly. It is better to pass this value as argument rather than storing and constantly updating it as fact in the SKB.

NUMA-aware allocation makes use of a simple reasoning algorithm. Finding the destination core's NUMA region involves combining the core's affinity domain with the memory region's affinity domain. The reasoning algorithm in the SKB derives an allocation policy which gets passed to the actual memory allocator. The memory allocator only provides the mecha-

nism of allocating memory. With the derived allocation policy in the SKB, the memory allocator can be instructed to allocate memory from a specific range (which is not necessarily the calling core's local memory).

Not only memory appears in the physical address range, but also many devices export their registers through a memory mapping to be set up by the operating system. Algorithms to derive policies on where to map which device run in the SKB based on facts about available physical address space and facts about device properties and their dependencies. PCIe allocation is one of the most complex problems of allocating physical address space. A detailed description of the PCIe configuration and its policy code to derive physical address allocation based on available address windows and device requirements is given in section 6.

The SKB derives not only lower-level policies, like memory and physical address range policies, it also derives policies for complete higher-level problems described in CLP. As an example, applications describe what properties in terms of hardware resources they would like to meet. A high-level description of application requirements and available hardware allows the OS to derive a core to application mapping. Chapter 8 describes in detail how global knowledge about running applications and hardware in the SKB is used to derive CPU core allocation policies.

3.3.3 Common patterns of resource allocation descriptions

The use-cases often have similar patterns in describing the desired resource properties. Often, problems need a description of numbers, addresses, address ranges and dependencies between them. The most generic description starts without assuming any concrete values. This means, the problem description starts with variables representing the numbers, addresses and ranges. Constraints between them relate the variables and describe their dependencies in a way, which is abstracted from concrete values. The variables often describe physical address ranges, RAM, NUMA nodes, number of cores or a cache hierarchy.

For example, bus and device drivers, which need to configure ad-

dresses and resources for devices, describe their allocation problems similarly to placing algorithms, which need to place applications on cores and NUMA nodes. Both types of algorithms operate on addresses or address ranges representing resources such as RAM or physical address regions. The concrete allocation of an address range depends on many factors which in fact limit the possible address set per resource. In fact, addresses are integers and hardware or software given limitations on supported address ranges are constraints on these integers representing addresses. Hardware resource allocation has often hardware given constraints and clear allocation and dependency rules, such as address alignment requirements or dependencies on other allocations.

In contrast to mechanism code, complex allocation algorithms only execute once in a while. The algorithms derive policy parameters which are valid longer term. Mechanism code operates in the fast path of the system. Mechanisms consider policy parameters to take fast decisions. By separating policy from mechanism code and running policy code in the SKB off-fast path, system performance does not suffer from a high execution time of the policy code, as long as mechanism code works properly on previously derived policy parameters[1].

Furthermore, it is often desirable, that policies and mechanisms are clearly separate. CLP not only allows describing various allocation policies such as physical address ranges for devices or core to application allocation, it also naturally leads to a clear policy/mechanism separation. While CLP allows describing allocation policies, it does not have direct access to the mechanism code and even less to registers. The use of the SKB for policy code enforces the programmer to think about a clear policy/mechanism separation.

To summarize, as long as allocation algorithms operate on integer values and as long as there are clear allocation rules constraining the possible set of addresses and modeling dependencies between them, CLP is a perfect match for implementing complex algorithms at a low code complexity for the programmer.

[1] High execution time means in the order of tens of milliseconds, as the concrete use-cases will show.

3.3.4 When to use the SKB

At a high-level, the SKB should be used whenever knowledge processing is involved. OS components, drivers and applications should make their part of knowledge available to other parts of the system. Likewise, the SKB should be used whenever hardware knowledge or high-level software state needs to be queried. Ideally, policy code should be implemented in the SKB, as long as the problem can be described using clear rules. The SKB has a global view and its high-level language reduces the code complexity. Also, this leads to a natural policy/mechanism separation.

To answer the question of whether to use the SKB in a more general way, I identified a number of characteristics of formulating problems that may apply. In the following paragraphs I discuss the general properties of a problem that may suit a CLP-based solution. If most of the following characteristics apply, a CLP-based solution may be appealing:

Configuration parameters need to be allocated from a constrained region For example, if there is a set of smaller address regions that need to be allocated from a bigger available address regions, the base address of every region can be translated to a variable to be assigned a concrete value by the CLP program.

Parameters have clear constraints If the configuration parameters have clear constraints (for example, natural alignment), these can easily be expressed as a CLP constraint.

Dependencies between parameters If there are dependencies between multiple parameters (for example, the placement of address regions defined by base and size parameters, such that position of one region influences where others can be placed), it is a good idea to use CLP. Constraints allow expressing these dependencies *before* concrete values are assigned to variables, leaving great flexibility in parameter allocation while still meeting the dependencies.

Permutations of configurations If meeting dependencies between configuration parameters might cause a large permutation and reassignment of other parameters, CLP can handle this cleanly by first collecting and considering all constraints, before assigning concrete values to variables. The imperative alternative would be to search for valid permutations by backtracking, which might be too expensive, and easily leads to complex code.

Handling special cases natively and cleanly Handling special cases in an imperative language often becomes messy quickly, because they are usually treated as workarounds added to the core code. By contrast, CLP allows additional constraints to be assigned independently of the core search logic, simplifying the treatment of special cases.

3.4 Design

This section describes the design of the SKB. The section starts by explaining the design principles, because they guided the overall architecture. Understanding the overall architecture is a requisite to correctly interact with the SKB. It further describes the possibilities of adding facts and the role of algorithms. Finally, the section talks about security.

3.4.1 Design principles

The design of the SKB is guided by a list design principles. The following paragraphs list and explain the design principles in detail.

The SKB should be the central knowledge engine. The SKB should allow every system component and application to add, query and modify facts. If it is used as the central knowledge engine, it provides a global view of the overall hardware and system state. Consequently, the SKB is designed as a service such that clients can connect to it and interact with it through a well-defined interface.

Provide high-level uniform information. The information should be provided in an abstract, easy-to-use and uniform format. The format of the information should be independent of the actual mechanism used to gather the information. It should also be easy to search information and match pieces of information with known values. While the SKB offers a high-level RAM-based storage for information, it is also the client's responsibility to prepare the information in a machine-independent format, before storing it to the SKB. Furthermore, the SKB has to support reading files with a priori knowledge from data sheets. The files should contain knowledge in the same high-level format.

Allow uploading and executing of declarative algorithms. The SKB should allow clients to upload application-specific policy code to the SKB. The code has to be declarative and should describe problems based on stored facts and additional input parameters. Algorithms can create variables, match them to facts or constrain them based on facts stored in the SKB. The SKB tries to assign values to variables such that constraints and relations on variables and stored facts can be met. The SKB allows clients to execute algorithms within the SKB and retrieve the results.

The SKB serves as a policy engine. The SKB provides a basis to implement policy code. It does not itself enforce policies and it provides no mechanism to apply derived policies. Security or register access, for example, have to be implemented outside the SKB. Furthermore, the SKB is reactive as in a classical server approach. It does not execute by itself. It only acts on behalf of clients.

Expressiveness. The SKB should not impose restrictions on the format and types of knowledge added to its storage. Also, it should not restrict algorithms in what they want to express. Instead, the SKB should support algorithms to be as expressive as possible. The SKB needs to support storing knowledge of current and future hardware, no matter, how complex hardware will be in the future. Likewise, the SKB should support any reasoning algorithm, even if it has to express complex relationships between

several facts in the future. A flexible and expressive query and update language is needed to retrieve the information stored in the SKB. Clients of the SKB should be able to express exactly what information they are interested in or how to add new data or update existing data.

Provide a convenient high-level interface to clients. Since the SKB is the central point of knowledge, it should be convenient for clients to add, modify and query facts by means of a high-level interface. The interface must be expressive enough to add detailed information about hardware and to add fine grained performance data from online measurements. The interface should also provide a simple mechanism to upload algorithms and execute them within the SKB. It should also allow to directly posing constrained optimization queries on stored facts and input parameters.

Policy/mechanism separation. As already mentioned, the policy/mechanism separation is a main enabling factor for simplifying operating systems code in this thesis. In fact, for all use-cases in this thesis the separation allowed the implementation of readable small and clean mechanism code as well as readable small and clean policy code. Furthermore, the mechanisms should work correctly without querying the SKB on the system's fast path. The clear policy/mechanism separation enables the system to use mechanisms on the fast path while executing policy code once in a while on an off-fast path. The high-level language basically enforces policy/mechanism separation.

The SKB should be machine independent. More precisely, the way of adding knowledge and the language, in which algorithms are implemented, should be machine-independent. Two properties are important to meet this requirement. First, if knowledge is described as uniform facts, it does not matter of what type the machine architecture is. High-level PCI knowledge (like base addresses and sizes) are the same on x86 and Sparc64, for example. Second, if algorithms are implemented in a language running on top of a language runtime, the algorithms are not bound

to a specific architecture. The high-level declarative code will be compiled at runtime to the specific underlying machine.

Data structures should be extensible. It should be straightforward to extend data structures in the SKB. Together with those extensions, algorithms need to be adapted as well.

The SKB should be modular. It should be easy to build additional functionality around the SKB. This includes support libraries, additional interaction mechanisms, data providing functionality and services built on top of it.

The SKB has to be able to boot early on system boot-up. Because the SKB is a central knowledge engine, it is used to configure hardware, which is an early task of an operating system. Hardware discovery and configuration as well as booting cores, finding memory regions and coordinating boot-up of most of the system is based on SKB information. The SKB therefore needs to be self-contained and as independent as possible from other system components.

Support concurrent access. On a manycore system, there is obviously a lot of concurrency. Since the SKB is a central point of information and there will be many clients interacting with the SKB at the same time, there will be concurrent access. The SKB must implement some synchronization to allow concurrent access in a safe way.

3.4.2 Overall architecture

Based on the design principles, the SKB is a self-contained user-level reactive OS service providing the facility of storing knowledge and running algorithms. Figure 3.1 shows the overall design of the SKB and its interactions with other modules.

The SKB is the central point of the complete knowledge infrastructure. Around the core SKB service, discovery and monitoring modules provide

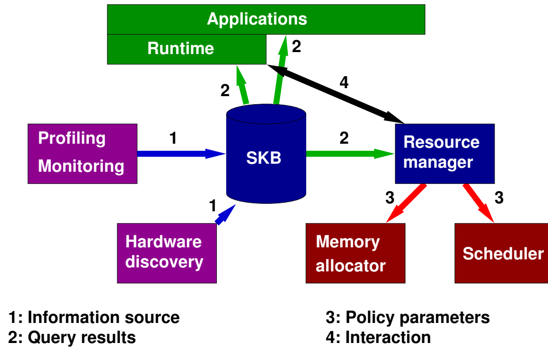


Figure 3.1: Overall system design

information and store them as high-level facts to the SKB. Device manager and resource manager closely interact with the SKB by adding and deleting facts and running algorithms within the SKB, whenever needed. Complex device drivers, such as the PCI driver, store all device related information at discovery time to the SKB and finally execute policy code, like, for example, an allocation algorithm within the SKB. Applications can directly query the SKB for information about the hardware, such as, for example, cache hierarchies. Managed language runtimes may optimize applications execution by querying the SKB and by combining the hardware knowledge with the internal knowledge of the application.

Apart from the SKB and its clients, there are libraries, event mechanisms, standard query functions and standard data-gathering modules available to facilitate the interaction with the SKB and to provide a base set of data and queries.

To ensure, that the SKB can serve as a policy engine to run hardware configuration algorithms early at the startup of the system, it is built as a statically linked and completely self-contained service. It runs from a RAM disk on which the base algorithms are stored. Unlike a physical disk, the RAM disk is accessible even before any hardware is configured.

3.4.3 Core

The SKB is a single-threaded event-based OS server. It is only reactive and does not perform any operation by itself. Upon a request, it performs and action and returns the result. In contrast to the base server, event-mechanisms such as Octopus, (see section 4.3) built on top of the core, may actively send asynchronous notifications on changes to stored facts.

The SKB embeds a CLP language runtime such that expressive algorithms can be implemented by clients and executed within the SKB on behalf of them. The CLP language runtime does not restrict the format of stored knowledge nor what an algorithm may express or compute. The only restriction is that the syntax of facts and algorithms is correct according to the CLP language. As discussed in section 3.8.2, this freedom is a nice feature for a research system, but it is a risk for a production system. There are, however, solutions to this problem.

The core of the SKB exports three basic services. First, the SKB provides storage for facts. Facts are kept in memory as long as the SKB is running. The SKB gets populated by external programs whenever the system boots up. There are different sources from which the SKB gets populated. Most of the facts are added by discovery and monitoring modules through the exported interface. A second method is to let the SKB load a file of facts into its memory-based storage. This is especially useful for facts which can only be known from data sheets. An example are PCI IDs to driver binary mappings (see section 5.1.2).

Second, the SKB allows executing queries of the facts. Single facts can be queried by matching the fact name and providing variables for the fields belonging to the fact. Queries can be built in a way that only fields of interest are returned as the result, instead of returning complete facts. Queries can also construct results by taking parts of information from different facts. Fields of different facts can be unified to each other. A typical scenario includes an equality join where one field is available in at least two facts. Unifying this field of both facts provides corresponding fields of both facts as a result. A query can combine fields od different facts to a new fact. It can arbitrarily name the new fact and add the desired number of fields to the new fact. It may or may not store the new fact to

the SKB. In most scenarios it does however not make sense to store these constructed facts, as they can always be reconstructed again. Storing them would require updating them, in case the base facts change. Queries do not change the state of the SKB by themselves. Queries are read-only. Only, if the client instructs the SKB explicitly, the SKB's state can be changed. Storing newly created facts explicitly, or deleting facts explicitly, are two ways of changing the SKB's state.

Finally, the SKB allows loading and executing algorithms from a file or through the interface on behalf of an OS service or an application. The algorithm gets stored in the SKB's memory. It can be called by name at any time after it is loaded. Algorithms can use any available facts. Additionally, input parameters can be passed to the algorithm whenever it gets called. The caller also passes variables to construct the output of the algorithm. The caller can define the format of the output in an arbitrary way.

3.4.4 Interface

The SKB exports a simple string-based interface through which facts can be added, queries can be sent and algorithms can be called. A string-based interface does not impose any restrictions at all, which is one of its design principles. Whatever the embedded language runtime supports, can be sent through the interface. While for a production system it may be desirable to restrict the interface, it is a perfect interface for a research system. It allows exploring many different techniques and algorithms without complicating the interaction with the SKB.

The basic interface to the SKB is based on messages (see section 2.2.4). The interface allows the sending of any string to the SKB. Results are received in form of strings as well. Additionally, the interface returns an error number and a string containing the error description back to the calling client. The client should therefore always check the error number for possible errors.

For this thesis I chose to implement a blocking interface to the SKB. Consequently, a call to the SKB blocks the client as long as the SKB is still executing the client's request.

The single-threaded nature of CLP makes it almost impossible to execute several requests at the same time without adding additional external mechanisms. A blocking interface, where one client after each other is serviced, naturally synchronizes executions of the calls. This facilitates the implementation of the request handling dramatically. The drawback is that clients do no useful work while they are waiting for a potentially long running query. Obviously, an asynchronous interface would allow clients to do useful work while they are waiting. It would still be possible to execute one query after each other. The SKB would need to buffer requests and remember request-to-client mappings. It would dequeue one request after each other, execute it and upcall the client with the result.

There is a second reason, why executions cannot easily be parallelized. Every execution of an algorithm potentially accesses every fact and potentially modifies every fact. If multiple algorithms execute at the same time (even in separate processes), there might be conflicts between algorithms reading facts and algorithms trying to update the same facts. Isolation mechanisms, like the ones used in database management systems, would be necessary to guarantee data consistency.

Since the SKB should be used off-fast path, blocking a client for the time of a request should not harm, especially for the conceptual research in this thesis. Clients which wish to continue processing while waiting for the result, can create a separate query thread. In this case, the main thread can continue executing while the waiting thread blocks. It is the client's responsibility to prevent multiple threads from calling the SKB concurrently.

The exported interface is the basic mechanism to communicate with the SKB. It only allows sending and receiving strings, but it neither helps to produce the right query strings nor to parse result strings. A client library builds on top of this basic interface and provides many convenient functions to assemble query strings and parse and interpret result strings. I explain the client library in section 3.6. Most clients should be fine with the client library, but in any case, they can always use the basic interface. The basic interface is defined the following way:

```
interface skb "SKB RPC Interface" {
    rpc run( in  string input,
             out string output,
             out string str_error,
             out int    int_error);
};
```

The parameter `input` refers to the input string. This can be any query, update or algorithm call in string form. More generally, it may be any string which is a valid input to the CLP system. As the interface does not restrict the input, a client might instruct the CLP system to behave in a certain way, depending on what special input string the CLP system supports. The parameter `output` refers to the output generated by the query or algorithm. The output is exactly as the CLP system produces it. The query or algorithm should create an output in a suitable way for the client. Even though the client has complete freedom to generate arbitrary output, there are some conventions, if it wishes to use the client library to parse the output. If it violates the conventions, it has to parse the output itself using the basic interface. In case of an error, `str_error` contains the error message, as the CLP system produces it. `int_error` is the return value of the CLP system invocation in the SKB. The concrete meaning of the return value depends on the CLP system. Whether the error string contains additional information also depends on the CLP system used. As explained in section 3.5, this thesis uses ECL^{i}PS^{e} as CLP system. Therefore, the meaning of return values and error strings are defined by ECL^{i}PS^{e}. A value of zero is a successful invocation of the system while every other value indicates an error.

3.4.5 Facts, schema and queries

The SKB does not enforce any fact format nor does it come with a predefined data schema. Clients can add facts in a format that suits them. Likewise, clients implicitly define the data schema by adding facts with a certain number of fields[2]. The only restriction is that facts must be syntac-

[2] Fields are like attributes in a database scenario.

tically correct according to the CLP system.

The advantage of such a flexible schema is that applications can start using the SKB as data store and policy engine without modifications necessary to the data schema. This is especially useful for applications, because there is no in advance knowledge what kinds of applications will be executed on a machine. Additionally, applications are developed by people outside the core team.

The risk of not having standardized the data model is that it gets messy. Data may be replicated in a different format or it might be unclear how to query existing data.

It is a tradeoff between flexibility and keeping the SKB clean. In this thesis, the focus is on the feasibility and usability of having an SKB at all, rather than on how to create a clean data model and how to enforce and maintain it.

The correct syntax of queries is defined by the CLP system as well. Because the SKB provides a string-based interface, there are no restrictions of implementing algorithms. Only, they must be correct according to the CLP system.

3.4.6 Data gathering

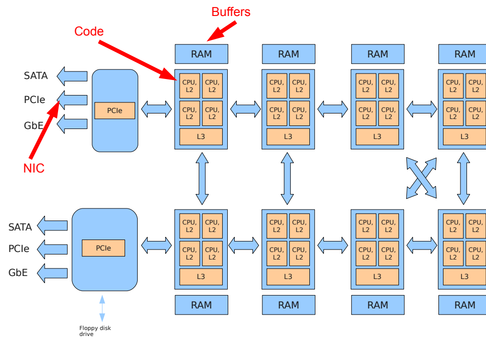
Before the SKB can answer queries about hardware properties or software state, it needs to be populated with detailed information. Because the SKB is a purely reactive OS service, it does not perform data gathering on its own. It has to be populated with facts by external programs such as bus or device drivers and applications.

I identified three ways of populating the SKB with information[118]. The first way results from *resource discovery*, such as traversing ACPI tables, enumerating and monitoring the PCIe or USB bus, and by querying registers of specific devices. Resource discovery and monitoring is done by drivers, which understand how to query specific pieces of hardware and how to store high-level general facts to represent the information in a generic, but still detailed way. This is an ongoing process, as devices may be hotplugged or removed, in which case the information in the SKB needs to be updated.

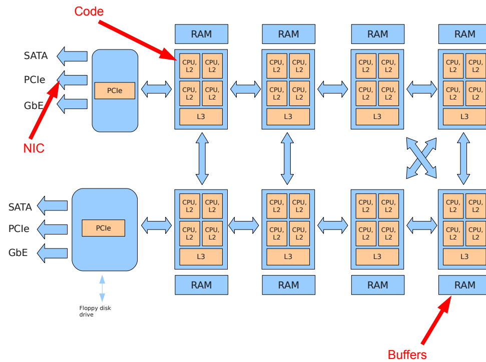
Online measurements, such as cache and memory latency measurements, are a second source of information. Measurements provide a view of hardware characteristics, as an application experiences them when it is running on this hardware. This derived logical topology view of the hardware does not necessarily correspond to the actual topology on every machine. As an example, the topology of the 8x4 cores AMD machine, as illustrated in figure 3.2, implies that running the network driver code close to the network interface card (NIC) and allocating packet buffers on this same node leads to the best performance in terms of UDP echo network throughput. The measured network throughput in this configuration in figure 3.2(a) is 668MBit/s. However, the measurements show that the packet buffer should be allocated as in figure 3.2(b). This leads to a network throughput of 888MBit/s. In this case, the actual topology does not correspond to the derived topology based on the measurements. It is therefore important to not only learn about hardware by resource discovery, but also to learn the concrete behavior through online measurements. Only this provides a realistic view of the machine and shows how an application experiences it.

Finally, there are cases where *a priori knowledge* derived from device data sheets has to be asserted, as there is no way of discovering information details at runtime. Device type information or a serial number of the device is sufficient to load a facts file providing a priori knowledge about this specific device. As an example, the *datagatherer* presented in this thesis uses the `cpuid` instruction on x86-based architectures to gather information about the cores and caches. This returned information has to be interpreted based on datasheets which match the current processor architecture.

It is obvious that device drivers should be responsible to add information about their devices. They know best how to access device registers and how to discover device properties and features. This is also the case for bus drivers. The PCIe bus driver is basically a device driver which knows how to handle the PCIe configuration space and how to derive facts for the SKB. It does not need to know details about every device, as the concrete device driver can take over the responsibility of adding further details of the device it is taking care of.



(a) Placing according to hardware topology: 668MBit/s



(b) Placing according to measured topology: 888MBit/s

Figure 3.2: Measured Topology vs. actual Topology: The measurements imply that the rightmost bottom node is closest to the NIC.

There are however cases where the responsibility of adding facts to the SKB are less obvious. Measurements of the memory hierarchy is a good example. The memory system does not need a driver and therefore there is no specific OS service already running, which could take over these measurements. In some cases it is even not feasible or not desirable that the driver itself adds facts about its device. An example of this special case is the CPU driver and its device, the CPU (see also section 2.2). In the Multikernel architecture, OS functionality is moved from the CPU driver to user-space. The CPU driver should not need to connect to the SKB and start adding facts about the CPU. To solve this problem, the SKB provides datagathering modules to query hardware information of common interest and to perform measurements. The datagathering modules run as separate applications and have to be started externally. The SKB does not start the datagathering application by itself. The concrete implementation of the datagathering application is described in section 3.5.3.

3.4.7 Algorithms

Algorithms transform information into knowledge by means of logical or mathematical rules, as explained in section 2.1. They match a number of facts, combine or transform them or parts of them and provide an answer to the client. There are basically three different types of algorithms and three different ways of loading them into the SKB.

While every client has full freedom of creating its own queries and algorithms, there are common queries which are used by different clients in different scenarios. It is not necessary to re-implement those in every client. It is advantageous, to collect these queries in a single place and load them once.

Some queries however are application-specific and only serve one application. In those cases, it is best if the application provides its queries and if it is responsible for loading them into the SKB.

Finally, some queries might be of common interest, but depend on information provided by a certain external application. In this case, it makes most sense, if the external application provides the information first and loads the high-level queries later on. Other applications can execute

those queries, without needing to know where the information came from and how the queries have been implemented.

Common queries

The generic ability of querying hardware information provides a great flexibility. Every fact can be considered in the query and facts can be matched to each other. However the complexity grows with the number of facts which have to be joined to derive an answer to a typically high-level question. A better strategy is to define goals[3] for commonly used queries. These goals answer high-level questions by always matching the necessary set of facts. This way, the complexity is removed from the client. The maintainability is much higher, because only one goal needs to be adjusted in case the fact formats change. Additionally, these goals are reusable for many clients and scenarios.

Therefore, the SKB loads standard common queries. Typical common queries are about basic hardware properties. The OS and applications need to know how many cores are installed and how much memory is available. Furthermore, the OS and applications often want NUMA-aware memory allocation and query the SKB for NUMA regions and core to NUMA affinity. Similarly, applications which care about cache optimizations query the SKB for cache information. These queries do not need to be replicated in every application. Instead, they belong to the basic set of queries which get directly loaded by the SKB.

The ACPI driver is an example that adds platform facts for general use to the SKB. Other clients can make use of them. The common queries loaded by the SKB access some of the facts added by the ACPI driver.

Application-specific queries

Every application has the freedom to add its own facts and to execute its own policy code. The policy code can access its own facts and combine them with system facts already contained in the SKB. Since there is

[3] In ECL^{i}PS^e, goals are like functions. These are basically rules defining how to combine facts.

no access control in the SKB (see also section 3.4.8), every application-specific fact uploaded by the application can also be accessed by every other client. Section 8.7 gives an example of an application which makes use of application-specific facts and algorithms.

External queries of common interest

The typical case is that clients are responsible for parts of the information in the SKB. In the common case, they also load the corresponding goals, rules and algorithms into the SKB. The client, which adds the data, knows best how to interpret it and what types of high-level questions can be asked. It knows exactly, how to get knowledge out of the gathered information. Also, if the facts need to be extended, it is one step to also extend the queries and algorithms based on them, without necessarily needing to change the high-level query.

The PCIe driver is one example which provides detailed facts about all devices and loads a set of functions to process the facts. It does not only run the allocation algorithm based on its own facts, but it also enables other clients to learn about PCIe devices. To facilitate processing PCIe information, the PCIe driver loads a second set of generic queries. The device manager is one example which needs to have at least a high-level understanding of which devices are installed. Specific device drivers can query high-level PCIe information related to the device. It is even extremely simple to implement a `lspci`-like tool to display all PCIe devices installed in the system in a nice way.

3.4.8 A note on security

The current version of the SKB does not have any security mechanisms for different reasons.

First, Barrelfish does not have a security framework. There is no notion of users or superusers and there is no possibility to identify processes[4] as trusted or not trusted. Consequently, there is no authentication and no

[4] A process is called domain in Barrelfish.

authorization possible and the SKB accepts queries, updates and algorithm calls from every process.

Second, the SKB does not check the submitted queries, updates and algorithm calls for embedded update and deletion statements. This means that code can be injected which alters or deletes all facts in the SKB.

These two security issues have an impact in a number of ways. Apart from the problem that every process can alter or delete everyone else's facts, arbitrary long running algorithms can be executed within the SKB. Since the SKB is single-threaded, this prevents any other processes from using it. This could be solved by creating a new instance of the SKB, whenever an algorithm has to be executed. The main instance serves as a master and keeps the most up-to-date data and loaded algorithm code, but does not execute algorithms itself. It would however complicate the implementation, especially the data management. If algorithms execute on different instances, data consistency has to be ensured like in a database management system (see also section 3.8.2).

This thesis is about the feasibility and usability of a service like the SKB. Security issues are not addressed in this thesis.

3.5 Implementation

This section presents the concrete implementation of the SKB server and, most importantly, the concrete CLP system used, as this has implications on the syntax of facts and queries accepted by the SKB. As mentioned in section 3.4.4, the interface does not restrict what facts, queries or algorithms are sent to the SKB, but the concrete CLP system embedded in the SKB does. Therefore, this section presents a summary of the accepted syntax for facts. It then moves to the implementation of the datagatherer, which queries information of interest to most of the clients. Finally, it describes, how common queries get loaded and how the implementation handles the early startup of the SKB, which has to be available for hardware configuration tasks.

3.5.1 Implementation of the SKB server

The core of the SKB is implemented partly in C and partly in ECLiPSe [10, 30]. The SKB program itself is implemented in C. The SKB program starts up and initializes itself as an OS service. It creates a message channel, and starts listening for incoming commands (see also section 2.2.4).

The SKB program embeds the ECLiPSe engine, which itself is implemented in C. The ECLiPSe engine is a managed language runtime for the high-level ECLiPSe CLP language. It is responsible for executing all the CLP code. It also takes care of storing, searching modifying and deleting facts, as instructed by the executed CLP code. The ECLiPSe runtime does therefore not perform operations by itself. It only performs operations on behalf of the executed CLP code. The ECLiPSe engine comes with a source code compiler, and a byte code to machine code compiler with optimizer. Source code gets compiled down at least to byte code at runtime. The language runtime might decide to compile parts of the code – either source code or byte code – to machine code at runtime. The actual functionality of the language runtime is implemented in ECLiPSe CLP code as well. When the main function of the SKB invokes the initialization function of the ECLiPSe engine, the ECLiPSe engine loads core functionality from a compiled ECLiPSe CLP file. This CLP code instructs the ECLiPSe engine to load more compiled files, as the functionality is all implemented in CLP and not in C.

The main SKB program interacts with the ECLiPSe engine by means of function calls. Every function call invokes the ECLiPSe runtime which executes some CLP functions. After reaching a defined state, the CLP code returns, causing the invoked C function to return back to the invoking SKB function.

The main server functionality is implemented in a loop. Whenever the SKB receives a request through the exported interface, it reads the input, invokes the ECLiPSe engine and sends the computed output back to the client.

3.5.2 Facts and schema

Because the SKB is based on ECL^{i}PS^{e} and the interface directly passes strings to the CLP engine, facts and queries are given in ECL^{i}PS^{e} syntax. Facts are basically named tuples. The syntax is given below in a EBNF-like format.

```
lower     ::= "a" - "z".
upper     ::= "A" - "Z".
digit     ::= "0" - "9".
number    ::= digit {digit}.
fact      ::= predname"." | predname "(" args ").".
predname  ::= lower {lower | upper | digit | "_"}.
args      ::= arg {"," arg}
arg       ::= atom | variable.
atom      ::= atomname | number | fact | list.
atomname  ::= lower {lower | upper | digit | "_"}.
variable  ::= "_" | upper {lower | upper | digit | "_"}.
list      ::= "[]" | "[" fact {"," fact} "]".
```

Facts are not only identified by their names. The arity (the number of arguments) further defines the facts. This means that facts with the same name, but different arities can co-exist. A query needs to provide the name of the fact it wishes to match and an arity. It will only match those facts with the same arities, even if more facts with the same name, but different arities, exist.

Building a knowledge base out of flexible ECL^{i}PS^{e} facts results in an extremely flexible data schema. The data has an implicit schema given by the names, arities and argument values of the facts. There is no prior schema or type definition available. Numbers and lowercase strings are treated as constants without type. Furthermore, every application adds its own facts with its own format schema during runtime. The schema is therefore changing or rather extended constantly. Because ECL^{i}PS^{e} uses unification to search for specific facts, it allows queries to try to match everything with everything. If it is not the same (for example, a number and a string), it is not unifiable and it will fail.

To query the SKB, the names and formats of the facts of interest have to be known. It is therefore necessary to follow conventions about fact

names, arities and attributes and their meanings. The declarative nature of the facts facilitate understanding their meaning by simply looking at them. The ECL^{i}PS^e command `listing` outputs the complete, dynamically added data[5]. Some well-known facts are listed below.

```
apic(ACPI_ProcessorID, APICID, Availability). % 1 = Yes, 0 = no
bridge(pcie|pci, addr(Bus, Dev, Fun), VendorID, DeviceID,
       Class, SubClass, ProgIf, secondary(Sec)).
device(pcie|pci, addr(Bus, Dev, Fun), VendorID, DeviceID,
       Class, SubClass, ProgIf, IntPin).
interrupt_override(Bus, SourceIRQ, GlobalIRQ, IntiFlags).
rootbridge_address_window(addr(Bus, Dev, Fun), mem(Min, Max)).
bar(addr(Bus, Dev, Fun), BARNr, Base, Size, mem|io,
    (non)prefetchable, Bits (64|32)).
fixed_memory(Base, Limit).
apic_nmi(ACPI_ProcessorID, IntiFlags, Lint).
memory_region(Base, SzBits, SzBytes, RegionType, Data).
currentbar(addr(Bus, Dev, Fun), BARNr, Base, Limit, Size).
pir(Source, Interrupt).
ioapic(APICID, Base, Global_IRQ_Base).
prt(addr(Bus, Dev, _), Pin, Source).
rootbridge(addr(Bus, Dev, Fun), childbus(MinBus, MaxBus),
           mem(Base, Limit)).
mem_region_type(Nr, Type).
memory_affinity(Base, Length, ProximityDomain).
cpu_affinity(APICID, LocalSAPIcEID, ProximityDomain).
tlb(APICID, level, data|instruction, AssociativityCode,
    NrEntries, PageSize).
cache(name, APICID, level, data|instruction, size,
     AssociativityCode, LineSize, LinesPerTag).
associativity_encoding(vendor, level, AssociativityCode,
                       Associativity).
cpu_thread(APICID, Package_ID, Core_ID, Thread_ID).
maxstdcpuid(CoreID, MaxNrStdFunctions).
vendor(CoreID,Vendor (amd|intel)).
message_rtt(StartCore, DestCore, Avg, Var, Min, Max).
nr_running_cores(Nr).
```

[5] It does not return the statically defined facts in an algorithm file.

3.5.3 Datagatherer

As mentioned in the design section of the SKB (section 3.4.6), some hardware pieces do not have a specific driver, which could add hardware facts to the SKB. In some cases there are drivers, but it is not desired that the driver adds facts to the SKB. Therefore, the SKB provides modules, which add facts for parts of the hardware which are of common interest.

The current implementation consists of several separate functions linked to one program: the datagatherer. The datagatherer needs to be started separately from the SKB. Once the first instance of the datagatherer runs, it spawns itself on every available core[6]. So far, there is only a datagatherer for x86-based platforms implemented[7].

First, each datagatherer instance queries CPU core information by calling the `cpuid` instruction several times according to the specification[5, 65]. It interprets this data based on the specification and adds facts about CPU features as well as the cache hierarchy to the SKB. The complete cache hierarchy, including sharing of caches between cores, can later be derived by SKB queries. Every cache has its own identifier. The identifiers allow deriving knowledge about sharing between the cores. Since every core runs its own datagatherer, the combined information is only available after they all terminated. As a consequence, the information about which cores share the same cache is only available after the datagatherers on the sharing cores have finished adding the cache identifiers. All information is added on a per core basis. This is important to support core heterogeneity, as other core types might have different cache characteristics.

Each datagatherer also measures the latencies to all levels of caches and to all available NUMA-nodes. It adds this information to the SKB. This provides a logical, measured view of the complete memory system. To ensure clean measurements, the instances measure the latencies one after the other. Measuring latencies concurrently from all CPU cores would result in high interconnect usage and finally in wrong numbers. The synchronization is done using Octopus, which I explain in chapter 4.3.

[6]The available cores are read from ACPI tables

[7]Datagatherers are platform-dependent, because they need to know how to get access to information at the register level.

After that, each instance queries supported features on each core. This includes, for example, power management capabilities and various other features. The features are added to the SKB per core.

3.5.4 Common queries

The SKB implements common queries in the file `queries.pl`. These queries are mostly related to platform information. Clients can learn about installed cores, NUMA regions and affinities. Also, Barrelfish uses an internal continuous core numbering. This numbering might differ from the actual hardware identifiers of CPU cores. The mapping is stored to the SKB. Clients can use these facts to translate core numbers to hardware core identifiers and vice versa.

The SKB always loads this file when it initializes itself. Because it is a regular CLP file, it might instruct the ECL^{i}PS^e engine to load further files. If, in future, more files of common interest should be loaded during initialization time, the additional file names can be added to the `queries.pl` file. It is not necessary to change the source code of the SKB.

3.5.5 Startup

The SKB starts early in the boot process of Barrelfish. A RAM disk contains all necessary rules and fact files as well as the compiled core functionality of the ECL^{i}PS^e engine such that the SKB can run even before hardware like a disk is configured. This enables the SKB to be used for basic hardware configuration like PCIe as well[8]. The SKB is completely self-contained and compiled as a statically linked application. The only dependency is on the memory server. However given that every other part of the OS needs memory, the memory server gets started early in the startup procedure of the system as well. A soon as it starts executing, it initializes itself, loads core functionality from the RAM disk and then loads standard query files from the RAM disk. It exports itself as a service and the rest of the system can start using it. Barrelfish treats the SKB as

[8]This is a requirement to access a disk afterwards.

a special service, like few other services such as the memory server. Because regular services export their references by means of the SKB, it is unfeasible for the SKB to export itself by means of itself. The connection endpoint of the SKB is therefore treated specially and belongs to the few well-known ones.

3.6 Client library

The section describes the client library, because this is the common way of interacting with the SKB. Small code fragments illustrate the most common ways of using the library.

On top of the basic interface described in section 3.4.4, the client library provides higher-level APIs to interact with the SKB. This library takes care of connecting to the SKB, sending requests using the basic interface to the SKB and receiving results. Additionally, the library includes functionality to create query strings in a similar way `printf()` creates strings from text and variables. Likewise, the library includes functionality to parse the result strings in a `scanf()`-like way. Parts of the result strings can therefore be copied or converted to variables in an easy way. Because many queries produce lists as a result, the library implements a simple form of a cursor to walk through lists and to extract all elements to C variables.

Typically, clients link to this library and use its functionality, rather than using the basic interface directly.

3.6.1 Using and initializing the library

The applications only need to link to `libskb.a` and include `<skb/skb.h>`. First of all, the library has to be initialized and a connection to the SKB has to be set up. A single function call takes care of both:

```
errval_t err = skb_client_connect();
if (err_is_fail(err)) {
    DEBUG_ERR(err, "connection to SKB failed");
    ... some useful error handling ...
}
```

This call sets up internal data structures like buffers for queries and results. It then creates a connection to the SKB and prepares everything for receiving results. The function indicates errors in its return value. If successful, the rest of the functions defined in the header file can be used to interact with the SKB.

3.6.2 Interacting with the SKB

Four main functions are used to interact with the SKB:

- `int skb_add_fact(char *fmt, ...);`
- `int skb_execute_query(char *fmt, ...);`
- `errval_t skb_read_output(char *fmt, ...);`
- `bool skb_read_list(struct list_parser_status *status, char *fmt, ...);`

The following subsections explain these functions in more detail.

Adding facts

Facts can conveniently be added using the function `skb_add_fact()`. This function works in a `printf()` manner. The first argument is a format string. After that, an arbitrary number of parameters can be passed such that they match the format identifiers in the format string. The function then produces a complete string and sends it as a message on the basic interface to the SKB.

The code fragment below shows how to add a simple fact. Only the fact name has to be provided, because it does not have any fields. Note that all facts have to be terminated with a dot.

```
errval_t err;

err = skb_add_fact("simple_fact.");
if (err_is_fail(err)) {
    DEBUG_ERR(err, "adding fact to the SKB failed");
    ... some useful error handling ...
}
```

The next code fragment shows how to add an n-ary fact. Fields can either be "hard-coded" or passed as parameters. In this case here the constant number "17" and the constant string "pcie" are hard-coded while the other parameters are passed as variables and matched in the format string.

```
int id = 5;
char description[] = "device";
int value = 0xfa;

err = skb_add_fact("nary_fact(%d, %s, 17, pcie, %d).",
                   id, description, value);

if (err_is_fail(err)) {
    DEBUG_ERR(err, "adding fact to the SKB failed");
    ... some useful error handling ...
}
```

Executing queries

Queries are executed in a similar way. The query string is prepared in a `printf()`-way. Queries match facts to variables. The variables of interest should be included in the output. The query has to explicitly construct the output by *writing* variables to the memory-based output stream. Not only variables, but additional text to structure the output and facilitate parsing can be written to the output stream. The following code fragment queries the `nary_fact` which was added above. The ID which is of interest is defined by the variable `id`. The description is not important for this query,

this means that it does *not define* it by not passing a value it should match and it does *not read* it by not passing a variable at that position. Instead it uses the underscore character which stands for an anonymous variable in Prolog[24]. Therefore, every value will unify with the anonymous variable which basically is the same as ignoring it. Leaving it away is not an option, because the arity has to match the arity of the stored fact. In this case here, the fact has an arity of five. The query makes sure that only facts with the constant number "17" will be unified. The fields of interest are the type and value. Therefore, the query provides two variables to be unified with the stored fact. These variables are part of the result. The query writes them to the output stream. To make parsing easier, it encapsulates the values in a `res()` tuple.

```
errval_t err;
int id = 5;

err = skb_execute_query("nary_fact(%d, _, 17, T, V),
                         write(res(T, V)).",
                         id);
```

Executing algorithms

At a high-level, algorithm execution is the same as query execution. Parameters with concrete values can be passed and variables to be matched are passed the same way as in query execution. The difference is, that the "query" string does not directly match stored facts, but instead a rule (or function). An algorithm can be seen as a CLP program consisting of several rules which transform stored facts or output of underlying rules into knowledge within a given context.

Rules have to be created first. Either they are uploaded directly through the interface or they are loaded from a file. Assume that the file `rules.pl` contains the following rule:

```
binary_fact(ID, V2) :-
    nary_fact(ID, _, 17, _, V),
    V2 is V * 2.
```

The `nary_fact` is matched first and the value is doubled before returning it. The code fragment below shows how to load the file.

```
errval_t err;
int id = 5;

err = skb_execute_query("[rules].");
```

The next code fragment shows how to execute the algorithm. It is perfectly fine that the execution passes a variable named `Val` instead of `V2`, because `Val` will be unified to the value of `V2`. How to implement programs can be learned from the Prolog book[24] and the ECL^{i}PS^{e} book[10].

```
errval_t err;
int id = 5;

err = skb_execute_query("binary_fact(%d, Val),
                         write(res(Val)).",
                         id);
```

Reading the output

The client library provides the function `skb_read_output()` to interpret the output string. Matching a number is similar to matching a number in `scanf()`. The `%d` conversion matches a decimal number. Matching text has to be done using `%[a-z]`, because the `%s` conversion would eat the rest of the string.

The `res(Type, Value)` output is a `res` element with a string value and an integer value. It can be read like in the code fragment shown below. The program passes `res(%[a-z], %d)` as a pattern to the `skb_read_output()` function together with the two variables used to store the result in a `scanf()`-like format.

```
errval_t err;
char text[80];
int val;

err = skb_read_output("res(%[a-z], %d).", text, &val);
```

Reading an output list

In some cases an algorithm produces a list of output elements. As an example, the algorithm could collect the `type` and `value` fields of all `nary_facts` stored in the SKB and return them in a single list of the form `[output(string1, integer1), output(string2, integer2), ...]`. In this case, the caller should iterate over the result list and convert all values to the corresponding C values.

The program prepares the pattern to be matched. Here it reads `output` elements with a string and an integer value, each. The `skb_read_list()` function parses the current element and stores the values in the passed variables. A `status` element remembers which element was processed last by this function. The status has to be initialized before being used in the while loop, by calling the function `skb_read_list_init()`. The code fragment below queries the SKB, initializes the `status` structure and iterates over the output list.

```
errval_t err;
char text[80];
int val;
struct list_parser_status status;

... execute query ...

skb_read_list_init(&status);

while(skb_read_list(&status, "output(%[a-z], %d)",
    textoutput, &number,) {
        ... do something with the values ...
}
```

3.7 Evaluation

This section evaluates the SKB in terms of code complexity and resource consumption characteristics. Because the SKB provides a service to its clients, but does not perform operations by itself, it cannot be evaluated in terms of performance. The performance evaluation depends heavily on how the SKB is being used by the client. The use-case chapters evaluate the algorithms in terms of performance.

3.7.1 Code complexity

One of the most important goals of this thesis is building an infrastructure which not only enables clients to take informed decisions, but also to take them with a small code complexity. This means that expressive algorithms should be implementable with few lines of code and they should be as readable and maintainable as possible. It should be clearly stated that the property of reduced code complexity applies mostly to the use-cases. It does however not necessarily mean, that the infrastructure provided by the SKB can be implemented with only a few lines of code. It is ok, if the

Functionality	LOCs C	LOCs ASM	LOCs CLP
ECL^{i}PS^e	97161	110	51469
SKB server	510	0	0
ClientLIB	300	0	0
Total	97971	110	51469

Table 3.1: LOCs

complexity is pushed towards the central SKB, if it helps to reduce the code complexity for most of the clients. Table 3.1 summarizes the number of lines of code needed to implement the SKB.

As the table shows, the total number of lines of code is relatively large[9]. This is not problematic for several reasons. First, as the table shows, the largest part is the ECL^{i}PS^e code. It accounts for 97161 lines of C code, some lines of assembly code and 51469 of CLP code providing the core functionality of ECL^{i}PS^e. Fortunately, programmers of clients (OS services and applications) do not need to maintain the ECL^{i}PS^e code. Second, the part of the SKB code which has to be maintained (initialization, exporting as a service) is only 510 lines of C code. This is relatively small and easily understandable.

3.7.2 Memory overhead

This section summarizes the memory overhead caused by the SKB. Because the SKB is used by various clients, the overhead does not account completely for each of them. In contrast to execution time, it can be amortized over several hardware and system configuration use cases.

Table 3.2 shows the breakdown and the total memory overhead of the SKB. The statically linked SKB program of 1.5MB includes the complete necessary code with library functions, which normally would be available as shared libraries. It also includes the complete language runtime with compiler and optimizer. The 600kB for the RAM disk not only con-

[9]LOC counts were generated using “SLOCCount” by David A. Wheeler.

	Size
Solver executable (statically linked)	1.5MB
RAM disk	600kB
Dynamically allocated RAM	60MB
Total	62.1MB

Table 3.2: Memory overhead

tains user CLP programs (such as the algorithms), but also the complete CLP core logic and basic ECL^{i}PS^e CLP goals, which are all implemented in CLP as well and stored as precompiled CLP files. Finally, CLP requires a sufficiently large preallocated heap, used to store facts as well as to compile, store and run CLP code. Additionally, CLP code creates many temporary variables and lists during execution on the heap. Finally, the backtracking stack needs to be large enough to allow creating the necessary choice points when searching the solution tree. The 60MB dynamically allocated RAM is used both for the temporary working heap and all hardware-related facts used by Barrelfish. This includes PCI data, and a description of available cores, memory hierarchy, performance profiles, etc..

3.7.3 Performance

Performance in terms of CLP code has different meanings. First, the time to execute a specific CLP algorithm can be measured. This time needs to be related to the actual compute complexity of the algorithm, which is not always obvious. The declarative nature of CLP allows a concise description of a problem, even if it has the same complexity as, for example, the bin-packing problem. Second, performance might be measured in terms of how good the result of an algorithm is. In this case, it depends on what the actual goal of the algorithm is and how well it can reach it or how close the result will be to the optimum. Finally, performance might be measured in terms of performance increase of a specific mechanism, if it is configured

with the "right" policy parameters produced by CLP code.

All three cases heavily depend on the actual algorithm executed, its goals and the mechanisms, which use the derived parameters. It is therefore not feasible to provide performance evaluations in this chapter. Instead, every use-case provides a separate performance evaluation in terms of algorithm execution time and "goodness" of the derived parameters according to the goals it should reach.

3.8 Discussion

This thesis evaluates the declarative language approach to express complex decision and hardware configuration problems. It also evaluates how well a new operating system on increasingly diverse and complex hardware can reason about hardware and adapt to it at runtime without prior hardware knowledge. My hypothesis is that the SKB as a reasoning engine, which runs high-level declarative algorithms, forms a good foundation to reason about the complexity of modern hardware, while reducing code complexity of the algorithms.

As the use-cases shall show, the approach is positive and turns out to be useful, but not without challenges. This section describes the common advantages and disadvantages of this approach.

3.8.1 Advantages

Clear policy/mechanism separation Maintaining a sharp distinction between, on the one hand, the algorithm code used to derive suitable policy parameters and, on the other, the mechanism applying the policy parameters has a number of strong benefits.

First, the algorithm can be clearly understood in isolation from the mechanism code, making it easier to both debug and maintain. Indeed, I developed, tested and debugged every algorithm "offline" in a vanilla ECL^{i}PS^{e} running on Linux using facts from a variety of machines copied out from ECL^{i}PS^{e}'s `listing` command on a running SKB instance. Only after this phase, I put the algorithms into service in Barrelfish's SKB. It is

also useful to be able to test this code by writing correctness conditions in ECLiPSe which are then validated automatically.

Second, the mechanism code is simplified, since it is no longer threaded through the algorithm code. Verifying that the mechanism code, written in C, is correct, becomes a simpler task, and the chances of breaking this code when changing the algorithm code itself reduces to almost zero.

Separation of special cases Special cases can be handled entirely in the declarative language, without polluting the mechanism code, written in C. The mechanism code only applies the final result, no matter how many special cases are modeled in the declarative algorithm code.

Moreover, special cases often are additional constraints which can be added besides the mainline algorithm code, *without* actually changing the mainline algorithm code. For the most part, additional constraints are one-line references to existing functions, and hence easy to add to the system.

All of this results in a clear separation within the declarative code between special cases and the solution description.

Flexibility of data structures Hardware information in traditional operating systems is typically represented by a set of simple, ad-hoc data structures (tables, trees, hash tables) whose design is determined largely (and rightly so) by performance concerns in the kernel. Using the SKB, detailed hardware information is represented in form of facts. The only exception is the system's fast path, which needs fast and specific data structures providing the minimal necessary information as quickly as possible.

High-level facts greatly facilitate reasoning about the information in ways not foreseen at design time. Facts added by independent programs, and originally thought to be used for specific use-cases, can be unified later on by third processes to gain further knowledge. For example, ACPI information about physical address region types can be transformed easily into regions not suitable for device mappings. The logical unification mechanism provided in languages like ECLiPSe makes this expressible in a single rule. Furthermore, this representation can be changed over time without concern for disturbing critical kernel code.

Late-binding of algorithm ECL^{i}PS^e allows adding new functionality as well as replacing functionality at runtime. This feature provides considerable flexibility. At any time, parts of an algorithm can be exchanged as needed without needing to change the mechanisms applying the algorithm's result. Even if the algorithm was executed already, it is possible to replace single parts of it. The next invocation will execute the new functionality.

Platform independence In many cases the policy code remains the same over different architectures. Because ECL^{i}PS^e is a managed language runtime which executes a high-level description of a problem formulation and because facts are given in a uniform way, it can run unmodified on different architectures. Only the architecture-specific *mechanism* code needs to change. This makes algorithms highly portable. Furthermore, only short mechanism code has to be ported, reducing the chance of introducing bugs when porting.

Reuse of functionality While CLP may be regarded as a somewhat heavyweight approach, the functionality provided is close to that required by many parts of a functional OS – in some ways, the system knowledge base might be regarded as analogous to parts of the Windows Registry[122] or the Linux `sysfs` file system[94], albeit with a much more powerful type system, data model, and query language. Barrelfish uses this functionality to represent various types of hardware knowledge and internal software state. Along with the authors of Infokernel[11], the thesis argues for making a rich representation of system information available for online reasoning and CLP provides a powerful tool for achieving this.

Complete description with constraints The complete solution to a problem can be described solely on variables and constraints on single variables and between different variables. Variables can be related to each other even before concrete values have been assigned. By implementing algorithms in this abstract form, the programmer leaves complete freedom

to the solver, which assigns values to variables. Therefore, no solution will be missed just because of suboptimal values assigned by a programmer.

Reduced code complexity and maintainability The use-cases show, that complex algorithms and policy code can be implemented with few lines of code. The complexity is reduced significantly. The code is not only better maintainable, because of the reduced complexity, but also because of the better structure. As already described, there is, for example, a clear structure between the mainline algorithm and special cases.

3.8.2 Disadvantages

Unsurprisingly, the approach also has some significant drawbacks.

Constraint satisfaction is no silver bullet Many allocation problems can be translated to simple rules and, consequently, can be expressed in a natural way. Special cases can be modeled as additional constraints to the base formulation of the allocation problem, keeping everything, with special cases, in a declarative description. However, this does not automatically lead to a solution in a reasonable amount of time. Constraint solvers have a well-known tendency to explode in complexity (and, consequently, time of execution) without careful specification of the problem, and the use-cases in this thesis are no exception in this regard.

Part of this is due to ECLiPSe being a relatively simple solver by modern standards, but much of the complexity is inherent. In practice, the onus is on the programmer to guide the solver by careful annotation of the problem. In some cases, it is advantageous, if the programmer knows how the solver instantiates the rules and variables and what strategy it uses to probe values. Rules, which are "solver-friendly" reduce the runtime involved in searching valid values meeting all the constraints. This makes the source code more complex than a simple specification of the constraints – the ECLiPSe code in this thesis is carefully written to avoid an explosion in complexity and runtime.

For example, in the concrete case of ECLiPSe, the programmer first creates variables, applies constraints and finally passes all variables to be probed to the solver in form of a list. Even if the declarative description of a problem has no order or control flow, the solver is still deterministic. The solver probes the variable list starting at its end. If the programmer knows that a re-probing of one variable would cause an almost complete permutation of the search space, this variable should be probed first (or at least as early as possible). Re-probing other variables might be fine, because they do not largely influence values of all remaining variables. In this case, the programmer can greatly reduce execution time by first sorting variables in a way that the "hard" variables are at the end of the list and get probed first. And still, the problem is described in its full generality and in a completely declarative way. Sorting variables, before passing them to the solver, does not influence the problem description.

Increased resource usage Even with the heuristics described above, ECLiPSe, compared to C, is an interpreted, high-level language with high execution time overhead. Additionally, a CLP algorithm works by propagating constraints and then probing values rather than assigning values in a straight-forward iterative way. Clearly this leads to longer execution times.

While the ECLiPSe CLP solver used for this thesis was easy to port and embed in an OS, it is relatively slow by modern standards. An alternative would be a more modern Satisfiability Modulo Theories (SMT) solver like Z3[34]. Z3 could express most of the constraint constructs used in the use-cases. The logical unification would need to be expressed differently. However, most probably it would significantly improve the execution time. SICStus Prolog[120] and SWI-Prolog[130] allow CLP programming through the CLPFD library[27]. GNU Prolog[36] allows CLP programming and even comes with a compiler to produce standalone executables.

Nevertheless, for some classes of problem the execution time overhead is not critical as long as it remains reasonably small relative to the use-case's complexity. Given that it runs off-fast path, an execution time of

less than a second should be acceptable.

Apart from being a programming language, CLP can also be used as an algorithm design tool: it aids in considering requirements, constraints and rules. Once implemented, CLP code can be compiled to C code and finally to a standalone executable – although ECL^{i}PS^{e} cannot currently output its internally-generated machine code in the form of an independent executable, other systems such as GNU Prolog[36] do produce standalone executables of constraint logic programs. This combination of CLP as a design tool and the compilation of the code down to an executable preserves many of the benefits, such as maintainability and clean design, while offering reasonable performance.

In the extreme case, CLP solutions can be applied completely statically. For example, resource constrained devices, such as small battery powered sensor nodes or embedded systems, usually have a fixed hardware configuration and often even a fixed set of tasks to run. The algorithm can run offline, on a standard PC with facts gathered on the device. The outcome of the algorithm can be embedded in the device's boot image and applied as if it was run on the device itself. With the approach in this thesis, it is particularly easy to run the algorithm on a standard PC. The algorithm and the facts are written in a platform-independent way and can be executed on every ECL^{i}PS^{e} instance, independently on the underlying device.

Large code base While the SKB enables clients to implement their algorithms with considerably less code (C and ECL^{i}PS^{e}), it does employ a large body of code in the form of the CLP solver. The port of ECL^{i}PS^{e} in Barrelfish consists of 97161 lines of C[10], plus a handful of assembly-language lines. In addition, the core CLP libraries add 51469 lines of CLP, many of them quite long. The complete solver executable (statically linked) consists of 1.5MB for a 64-bit x86 OS. Additionally, a compressed RAM disk of 600kB provides the necessary ECL^{i}PS^{e} files. This is clearly significant, and adding this amount of code to the boot image of an OS raises at least two concerns.

[10]LOC counts were generated using "SLOCCount" by David A. Wheeler.

First, there is the issue of code bloat. On modern hardware, the boot process is not unduly impacted by the overhead. Still, loading the SKB and running hardware configuration algorithms in CLP at boot up increases the boot up time. On the other hand, as mentioned above, the CLP solver does provide a valuable data management service to many parts of the OS as a general name server and policy engine, and so the cost in code size should be amortized over the whole set of client subsystems which use it.

Second, there is the extent to which the CLP solver itself can be trusted. The OS and all the SKB clients rely on ECL^{i}PS^{e} behaving correctly. Since it is a mature, general-purpose system which is actively maintained, the expectation that it is reliable and relatively bug-free should mostly hold. However, it is unlikely that a complex piece of code like ECL^{i}PS^{e} will be formally verified, which makes this approach less attractive for high-assurance operating systems. However, such systems typically are written to specific hardware platforms, obviating the need for complex configuration logic.

For high-assurance, formally verified systems, a better application of this approach would be to apply the ideas at compile time, which would integrate with the seL4 approach[72] of modeling the entire OS in a high-level language, which is then translated (in a way that preserves the verified properties) to C.

Finally, it is often worthwhile to model complex algorithms in CLP for which an imperative solution would otherwise be too complex to implement. It is however often simple to check the correctness of the algorithm's output in an imperative way. The correctness checking code can easily be implemented in C and fully verified. This property would allow runtime validation of the results of the CLP search, without the need to rely on ECL^{i}PS^{e} behaving correctly for all possible inputs.

Boot sequence Configuring hardware at OS boot time in a high-level language like CLP means that the language runtime has to be started early in the boot process. Barrelfish may be unique in loading a full CLP system before configuring hardware.

Perhaps surprisingly, this imposes very few requirements on the OS.

The SKB, like most of the components, executes in user space as in a classical microkernel design. However, CLP requires very little of the OS to be functional beyond basic (non-paged) virtual memory and a simple file system, initially from a RAM disk image.

The dynamic nature of the solution allows loading further functionality after an initial configuration when disks, networking interfaces, etc. come online.

Learning curve Most OS programmers use C rather than ECL^{i}PS^{e} to implement algorithms, and the learning curve for a language like ECL^{i}PS^{e} is almost certainly steeper than for C. However, it is likely that someone with a basic knowledge of ECL^{i}PS^{e} will find it easier to understand small and simple high-level code than a complex, imperative C version.

Furthermore, the SKB is by no means the first system which employs logic programming in an operating system – for example, Prolog has been successfully used to provide network configuration logic in Windows[61].

Expressiveness is a risk Using a complete CLP language runtime with no restrictions at all allows any problem to be expressed in form of a CLP program. While this is a desired feature, it comes also at a risk. Long-running algorithms block the SKB and make it unusable for other clients, while the algorithm is still executing.

For a research system it is a nice feature, however, for a production system, it would be necessary to either restrict algorithms or to bind the execution time. Another approach would be to spawn a child SKB for every execution of an algorithm. This would lead to a similar architecture as many internet servers have. Every client is handled by a separate thread or even a separate process. Algorithms would still have access to all facts, but would not block other clients, even if their execution time would be high.

Apart from a long execution time, there is a second risk which may arise. Conflicting constraints prevent the solver from finding a valid solution in which case the output is simply "No.". Algorithms need therefore to be designed carefully in any case. Algorithms specifically designed to

solve one problem can easily be implemented in a way that no conflicting constraints are applied. However, more dynamic algorithms, where multiple applications can add more requirements in form of constraints, are at the risk of not finding a solution. In these cases, the mainline algorithm needs to check in advance whether conflicts may arise. They need a fallback scenario or a way to restrict what additional constraints applications can add.

3.8.3 Approaching a configuration problem in CLP

CLP can handle many complicated requirements on resource configuration. However, its expressive power is also dangerous: one can easily create unmaintainable and sub-optimal code in CLP if a problem is tackled in the wrong way. It is essential to follow some general rules when approaching a problem formulation in CLP.

First, it is essential to define an appropriate data structure and to create every configuration parameter variable (such as, for example, memory addresses) only once, so that all necessary constraints can be applied to the single variable standing for a parameter. For hardware configuration, a data structure which mirrors the hardware topology is natural, and allows dependencies between devices to be expressed in the data structure between the items representing them. The data structure should contain one variable for each parameter (such as memory address), which will be be assigned a concrete value by the CLP system. Next, the data structure is walked and constraints applied to the variables in such a way that no temporary variables are created, and no constraints are mistakenly applied to these temporary variables. Unfortunately, when using a mix between CLP and Prolog (as in ECL^{i}PS^{e}), it is easy to create temporary variables by mistake. Finally, the variables should be collected and passed to the CLP solver to instantiate them with concrete values.

3.9 Summary

With the SKB I built a central service which takes care of storing knowledge in a high-level declarative way. Furthermore, its embedded declarative language allows clients to run declarative algorithms based on these high-level facts. The SKB servers as a policy engine for various configuration and decision problems. Furthermore, the complexity of decision and configuration tasks can be pushed to the SKB and significantly reduced through the use of a high-level declarative language.

A simple client library facilitates the interaction with the SKB, without restricting the expressiveness provided by the CLP language.

Several use-cases presented in the following chapters prove the usefulness of the SKB. The problems to be solved and how they are approached using the SKB are described in the respective chapters. A discussion of advantages, disadvantages and possible challenges along with an evaluation of the concrete algorithms is given on a per chapter basis.

Chapter 4

Coordination

This chapter presents Octopus, the coordination service in Barrelfish. While a distributed structure allows applying algorithms from the distributed field, it also suffers from similar problems like synchronization, naming, distributed locking and coordination of service instances. Services running on different cores do not necessarily know each other and still have dependencies. The service dependencies need to be resolved externally and services need to be coordinated.

This chapter presents the design and implementation of Octopus, which is a native extension of the SKB. It provides easy-to-use, high-level, uniform coordination primitives and event mechanisms. It directly benefits from the advantages provided by the language runtime of the SKB. Further, the SKB already runs as a centrally available service. This forms a nice ground to embed coordination and event functionality both of which are available from the moment Barrelfish starts up. Although it is implemented on top of the SKB, the careful implementation ensures reasonable performance.

4.1 Introduction

The distributed structure of Barrelfish reduces the complexity of single software components. Each component is only responsible for a specific task. On the other hand, the OS still provides functionality as a whole to the complete software stack. This requires components to interact correctly, even if they do not know each other. An external coordination and synchronization service enables them to coordinate. At the same time it removes complexity involved in synchronization code from the components.

To take an example, the basic hardware configuration must be done, before drivers initialize single devices. Likewise, device drivers need to initialize devices, before services, based on them, can be used by the rest of the system. A network interface card (NIC) driver, for example, executes as a separate program. It must however wait for the PCIe driver to configure the PCIe bus, before the NIC driver is allowed to access the network card's registers.

At boot time, services must be started in an order which respects their dependencies and, preferably, minimizes startup latency. As devices (and cores) come and go, drivers must be started up and shut down, while meeting their dependencies. Effective power management requires knowledge about device dependency: shutting down a USB controller or PCI device, for example, should only be done if the dependent devices are safely shut down as well. Moreover, the OS now has complex synchronization requirements between components: hotplug events may involve careful coordination between PCI managers, ACPI subsystems and device drivers.

In existing systems, resolving these dependencies and implementing synchronization patterns between OS components and modules is typically hard-coded into the components themselves. In some cases, the synchronization is implicitly ensured by the control flow of the program. A Linux kernel, for example, initializes subsystems by calling the initialization function from the main initialization function. There are however cases, where synchronization between concurrently running components is hard-coded in an ad-hoc way. This leads to high complexity in the OS and therefore also to correctness and reliability issues.

The main goals of Octopus are to provide a clean and high-level interface such that dependencies can be expressed by components, which do not necessarily know each other. Octopus should also serve as a basis to implement distributed synchronization primitives with a small code complexity. Finally, as coordination of OS components is necessary from the boot up of the system, it is a requirement that Octopus runs from the beginning, without depending itself on other OS services.

Octopus is inspired by facilities such as Chubby[26] and Zookeeper[63]. It is based on the SKB for several reasons. First, it directly benefits from the declarative language facilities and thus, the code complexity involved in synchronization primitives can be reduced. Second, the SKB is a central service running anyways. By putting Octopus functionality on top of the SKB, it will always be available without increasing the OSs complexity by means of additional services. Finally, the SKB is designed in a way that it can boot up early. Therefore, Octopus is available early as well, which is one of the important requirements to synchronize OS boot up.

Octopus has been mainly implemented by Gerd Zellweger. He describes the work in his Master's Thesis report[143]. Additionally, the design, concepts and the use-cases presented in this chapter have been published in a recent paper[144].

4.2 Background

Octopus is a synchronization facility with events and a fast and simple key-value store. Therefore, this section summarizes related work on both topics.

Octopus builds on ideas from the distributed computing field. Traditionally, data centers have faced complex coordination problems at the level of distributed systems on clusters. Chubby[26] and Zookeeper[63] provide coordination and synchronization for large collections of machines. They organize information in a hierarchical name space and export a file system-like API. Zookeeper and Chubby are used as a multipurpose tools for various coordination tasks such as configuration management, storage, group membership, leader election, locking and mutual exclusion. Some-

what unexpectedly, Chubby also increasingly replaced DNS as a general-purpose name server internally at Google. Both systems use state-machine replication to achieve high availability, using variants of Paxos[76] for consensus among nodes.

With the increasing demand of fast access to data at massive scale, key-value stores favor simplicity in terms of data model and query complexity over strong guarantees such as the ACID properties provided by centralized relational database systems. Distributed key-value stores implement a form of distributed hash table[35, 47], providing eventual consistency. Redis[111] is one example of a centralized RAM-based key-value store with optional master-slave replication and persistence. It aims to be lightweight and high-throughput, and stores schema-less data under keys. Redis provides a flexible set of atomic operations on single data items.

Publish-subscribe systems allow flexible interaction in distributed systems and feature three key ideas[41]. First, *space decoupling* means that interacting parties do not need to know each other. Second, *time decoupling* means that interacting parties do not need to be actively participating at the same time. Finally, *synchronization decoupling* means that publishers never block on generating data and subscribers get asynchronous data events.

In the OS context, D-Bus[46] is an interprocess communication facility for Linux and other operating systems which also supports a limited form of coordination: processes can wait for events from specific objects, and the D-Bus daemon can start processes when messages are sent to them.

4.3 Approach

This section explains the design of the Octopus service and shows in detail, how data items are stored and forwarded. It is important to know that the high-level language in Octopus serves to reduce complexity when building distributed synchronization primitives.

4.3.1 Design principles

Octopus borrows ideas from the interfaces provided by Chubby and Zookeeper. The main goals are facilitating distributed coordination and event handling, while reducing the code complexity involved in programming such functionality. However, the OS environment is somewhat different from a large cluster. The requirements and resulting design principles are therefore different. The paragraphs below summarize the design principles applied in the Octopus design.

Independent service The service must be *self-contained*. Octopus should make the coordination of the OS boot-up possible. This is only feasible, if it does not rely on many other OS services (such as the file system or network). Octopus should help to model dependencies, but it should not create new ones itself. Every functionality necessary should be included in a single program image.

Centralized service The code complexity should be as low as possible. To reduce the code complexity and to make the first goal possible without needing anything special from the OS, Octopus should be a lightweight, *centralized* coordination service rather than a replicated system. On a single machine, an OS can still assume a reliable interconnect and no single CPU failures, at least in the medium term. A centralized service should therefore not cause availability issues.

Loosely coupling Information providers and consumers should be *loosely coupled*. Services, which do not necessarily know each other, appear and disappear during runtime of the system. A loose coupling can be achieved by an asynchronous interface with a fast, flexible and scalable data and query model.

Non-blocking interface The query interface should never block clients, even if their queries cannot be answered at the query time. The interface should be completely *non-blocking*. Instead of blocking, clients should be

able to sign-up for future notifications, which Octopus sends, whenever the query can be answered. Asynchronous events should notify clients about any data store changes of their interest.

High-level interface The code complexity involved in using the coordination service and implementing additional synchronization and coordination primitives based on Octopus should be minimized. A *high-level interface* achieves high expressibility at a low code complexity. This is why Octopus should export a high-level interface to its clients.

4.3.2 Octopus

This section describes the overall architecture of Octopus. Based on the design principles listed in the previous section, Octopus implements distributed, named synchronization primitives such as locks, barriers and semaphores above a key-value store and associated event delivery system. Octopus unifies synchronization, name service, and event handling for the OS. Octopus exports a convenient API for the key-value store to clients. Using the API, clients access the key-value store. They can add, modify and delete data or search for data items. Also, the API allows enabling notifications for specific changes of the key-value store. Finally, for the publish-subscribe system, clients can subscribe using the API and publishers publish data by means of the API. It is therefore a complete, but still simple and convenient API providing access to all functionality of Octopus.

Octopus handles two types of data. On the one hand, it handles *transient* data in the publish-subscribe case. This data never gets stored, but only forwarded in form of notifications or publish events to clients. On the other hand, it handles *persistent* data. This data gets stored to the key-value store and remains in RAM during the lifetime of the OS or until it is deleted.

Octopus abstracts the key-value store behind high-level *record* entries, and a *query and update language* enables clients to add, query and modify records. Clients register for events at the record level. The two advantages

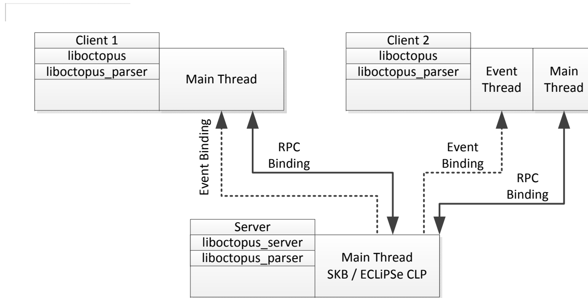


Figure 4.1: Octopus: General Architecture

of a high-level language are reduced code complexity and independence of the implementation. Section 4.3.3 describes records and the query language in more detail.

Octopus is built as an native extension to the SKB as shown in Figure 4.1. Server functionality is in a library `liboctopus_server` linked with the SKB. Clients link to `liboctopus` which exports the Octopus API and communicates with the Octopus service. The `liboctopus_parser` library parses query and answer strings on both sides.

While Octopus is strongly integrated with the rest of Barrelfish, the ideas it embodies are widely applicable to any OS trying to manage a complex multicore machine.

4.3.3 Records and Record Queries

Records are the basic data unit in Octopus. Clients add records to persistent storage and retrieve, modify or delete them. They can also register for *addition* and *deletion* events on patterns matching records of interest. Octopus also provides a publish-subscribe API for records which is similar but bypasses storage.

Records

Records consist of a name and an optional list of attribute-value pairs. The syntax is based on JSON (JavaScript Object Notation)[67], since it is easy to read and write for humans and machines. The following example shows a record called `hw.pci.device.1` representing a PCI network card:

```
hw.pci.device.1 {
    bus: 0, device: 1, function: 0,
    vendor: 0x8086, device_id: 0x107d,
    class: 'C'
}
```

Sequential records are a special form of records. Octopus appends a monotonically increasing number to the name defined by the client. It returns the new name to the client, allowing clients to create multiple, unique, ordered records, and serving as the basis for synchronization primitives.

Record Queries

Record queries use an extended version of the record entry syntax, allowing regular expressions for record names and attribute values and the special character '_' to match to any name or attribute value. Constraints on attribute values further specify whether records are part of the result or not. Record updates can depend on the currently stored value, as in SQL's `UPDATE` statement.

The following example matches records with any name but only those with `device <= 1`, `vendor > 100` and `class` matching the regular expression `C|X|T` belong to the result. An update sets `bus` to `5`, but only if the current value is `0`.

```
_ { bus: 5, bus == 0, device <= 1,
    vendor > 100, class: r'C|X|T'
}
```

4.3.4 Record Store

Whenever the Octopus service receives an *add* or *del* query from a client, it parses the query and performs the respective operation on persistent storage. Octopus stores the attribute-value pair list to the storage hash table with the record name as key. For *get* and *update* queries, it matches them to the stored records and returns the result to the client.

If the client is interested in future *add* or *del* events it creates a *trigger* along with the query and passes it to Octopus. The client specifies whether the trigger is persistent and should send an event whenever the query matches, or whether it should be automatically removed after the first event.

Octopus stores the trigger to the persistent storage hash table. Because record queries do not need to specify a fixed name, Octopus generates a trigger ID which serves as the key. Expected attribute-value pairs and constraints get stored with this ID.

The full record store API of the server is:

```
(names, err, t_id?) = get_names(q, trg?);
(record, err, t_id?) = get(q, trg?);
(err, record?, t_id?) = set(q, trg?);
(err, t_id?) = del(q, trg?);
(err, t_id?) = exists(q, trg?);
```

`get_names` returns an array of record names matching the query. `get` returns the first record to match the query. `set` inserts a new or updates an existing record. `del` deletes a record. `exists` is similar to `get`, but only returns an appropriate error code. All calls may install a trigger, in which case the server returns the trigger ID to remove it in the future. Creating triggers is done as follows:

```
(trg) = mktrigger(in_case?, send_async, mode, handler_fn,
    client_state);
(err) = rmtrigger(t_id);
```

`mktrigger` creates and configures a trigger according to flags passed by clients. It also installs the user handler function and user state. `rmtrigger` removes the trigger identified by its ID.

`in_case` defines the installinf of the trigger only if a specific error happened during the query invocation (e.g., no record found). `send_async` is used to indicate whether the trigger event should be returned to clients in a synchronous or asynchronous fashion. `mode` is a bitmask used to indicate interests in specific (i.e., add or delete) events. `handler_fn` and `client_state` are arguments supplied by the user. In case of an asynchronous trigger, `liboctopus` uses these to call `handler_fn` and supplies `client_state` along with the matching record and event type as an argument.

4.3.5 Publish-subscribe

Apart from storing records, Octopus offers the publish-subscribe model to clients. Publishers publish records and subscribers get these records in form of an event. Octopus does not store published records. The record format is the same as described in section 4.3.3. Subscriptions are defined using the same record query language. Similar to triggers, subscriptions have to be stored to persistent storage. Whenever a record gets published, Octopus considers stored subscriptions, matches their specified constraints and, in case of a match, it sends an event to the corresponding client. On the client side, the same handler function can be used as for triggers. Subscriptions remain installed until they explicitly get removed by the client. Octopus provides a simple API for the publish-subscribe model described below.

```
(subscription_id, err) = subscribe(handler_fn, client_state,
    subscription);
(err) = unsubscribe(subscription_id);
(err) = publish(record);
```

4.3.6 Implementation

Octopus is implemented as a native extension to the SKB. As such it benefits from ECL^{i}PS^e's logical unification, backtracking, constraint evaluation and regular expression facilities. Queries are automatically matched

to stored records. This reduces code complexity both for applications and Octopus itself.

Octopus allows searching for records based on attribute values, and potentially all records need to be considered. While this matching works fine with ECL^{i}PS^{e}'s backtracking and matching facility, its performance suffers from doing such a full search. To overcome this problem, Octopus implements an attribute index specifically to improve matching/search performance. The *attribute index* remembers all record names having a given attribute. Thus, Octopus quickly finds all potential matching records. The index is implemented as a skip list[108], which behaves similarly to a binary tree. Finding triggers or subscriptions, given a record, is the opposite problem. Given a record, a bitmap index indicates whether a trigger or subscription ID is relevant.

4.4 Use-cases

This section presents use-cases, derived from real problems in Barrelfish. Especially the name service and the device manager (see also chapter 5) are heavily used in Barrelfish and as such important parts to understand.

4.4.1 Synchronization primitives

Octopus implements high-level synchronization primitives based on records. These are intended to coordinate distributed applications and are not suitable for fine-grained access control among threads sharing an address space. Following the general goal of this thesis to reduce code complexity wherever possible, it is clearly a goal to build synchronization primitives with few lines of code, while still providing the necessary functionality. The key-value store in combination with change events provides a useful basis for such primitives: new clients can query existing state, such as whether a client already holds a lock, for example, and existing clients receive change events, such as a client has released a lock, for example. Currently, Octopus implements two synchronization primitives. These are based on the ones in Zookeeper[63]:

Locks: In an approach reminiscent of eventcounts and sequencers[112], acquiring a lock creates a sequential record using the lock name, agreed on by the clients. The client owning the record with the lowest number holds the lock. Other clients issue an *exist* call on the previous record to their own and pass a one-time trigger on its deletion. When the lock holder releases the lock (i.e., deletes the record), the next waiting client wakes up on the deletion event. As with eventcounts, no starvation occurs and the locks are fair in waiting time. The unification of the key-value store `exist` call and the event registration for the deletion of a record, which will wake up the client when it gets the lock, allows the implementation of distributed locks with only few lines of code. This reduces the code complexity of the implementation of distributed locks significantly.

Barriers: Barriers ensure that different tasks start executing a section simultaneously. Octopus contains a double barrier implementation based on sequential records. Every client entering the barrier creates a sequential record and queries if the number of records corresponds to the expected number of clients entering the barrier. If so, it creates a special record indicating that all clients are ready. Otherwise, it creates a trigger waiting for this special record. Leaving a barrier works the other way around. Every client deletes the previously created record and waits for deletion of the special record. The last client deletes the special record which triggers the event that all clients left the barrier. Again, triggers for creation and deletion of the special record when entering or leaving the barrier respectively ensure that clients wake up only at the time when all clients have reached the synchronization point.

4.4.2 Name service

Barrelfish needs a service registry or name service, as does every distributed system[101, 133]. As explained in section 2.2.5, every service registers a message channel by name.

Clients resolve them by name, or more complex attribute-based queries. Octopus allows implementing an expressive service registry in Barrelfish using records of the form

```
servicename { iref: <nr> }
```

where `servicename` is the well-known name and `<nr>` is the internally used reference expected by Barrelfish's connection function. Service dependencies are resolved by searching for a specific service name and waiting until it appears as a record. Octopus's trigger API allows clients to install triggers for service references. While services, upon which a client might depend, are not up yet, clients can do useful work. As soon as the service appears, Octopus sends a notification to the client. An event-based internal structure of the client directly benefits from this mechanism. Dependencies can easily be solved this way. Furthermore, clients and services are loosely coupled and do not need to know each other. The only knowledge a client must have is the service name.

4.4.3 Application coordination

A manycore machine offers a high degree of parallelism. Ideally, applications make use of real parallelism. The common way of exploiting parallelism is creating threads on multiple cores and synchronizing them using shared locks. In a Multikernel architecture, the OS assumes there is no shared memory and no cache coherence. OS services and applications exploit parallelism by creating process instances on several cores. These instances have to be synchronized using explicit message-passing. Coordinating instances using the low-level messaging interface quickly gets complicated. Synchronization protocols have to be designed and implemented over the messaging interface.

The presented synchronization primitives drastically reduce this complexity. There is a trade-off between performance and code complexity. If it is performance-critical, an application might rather use synchronization directly based on Barrelfish's message passing facilities. Otherwise, using the primitives provided by Octopus might be just as fine.

An example application in Barrelfish is the datagatherer application presented in section 3.5.3, which collects information of every core. A separate instance runs on every core and collects per core information. Gathering per core information does not depend on other instances and the

instances therefore do not need to be synchronized. However, datagatherers also measure access latencies of the cache and memory hierarchy. The measurements are only stable, if not all instances measure all memory locations (such as NUMA nodes) at the same time. This would stress the interconnect which finally leads to the illusion of a flat memory hierarchy. The datagatherer synchronizes the measurements of the memory hierarchy by high-level synchronization primitives provided by Octopus.

Applications which want to make use of information collected by the datagatherer obviously depend on its termination. Each datagatherer instance adds a record after it is done collecting all information. Applications can install a trigger to get a notification, as soon as the datagatherer instance terminated. While waiting, the applications can do useful work.

4.4.4 Device management and system bootstrap

Device management and system bootstrap require careful synchronization between different layers of drivers and OS services. Especially at bootstrap, the right order of drivers need to be started, depending on what devices have been discovered by previous drivers. Finally OS services export higher-level functionality on top of certain devices.

Kaluga, Barrelfish's device manager, coordinates the starting of drivers as well as the hardware-related parts of the system bootstrap. Because CPU cores are treated as regular devices with the CPU driver as device driver, Kaluga basically coordinates the Barrelfish's bootstrap. Kaluga and the system bootstrap are explained in detail in chapter 5.

4.5 Evaluation

The evaluation of Octopus in this section is mainly about code complexity, because reducing complexity involved in synchronization primitives is one of Octopus' main goals. Still, a reasonable performance is of importance and evaluated in this section as well.

Functionality	C	CLP	Flex	Bison
Octopus	3188	355	150	94
Barriers	102			
Locks	87			
Semaphores	106			

Table 4.1: Lines of code

4.5.1 Code complexity

The code complexity is measured in terms of lines of code needed to implement distributed synchronization primitives. Table 4.1 shows a functionality breakdown with lines of code[1]. As the table shows, Octopus itself is implemented partly in C and ECL^{i}PS^e. It needs about 3200 LOCs of C code and about 360 LOCs of CLP. Additionally, the lexer and parser of the records and record queries are implemented in a handful LOCs of flex[45] and bison[20]. More importantly, the synchronization primitives built on top of Octopus are implemented in roughly 100 LOCs of C code. They only need to add records, install triggers and finally delete records using the Octopus API. Because the Octopus service runs inside the SKB, the records are available to any client and therefore these synchronization primitives work for distributed synchronization without any additional protocol necessary. The goal of keeping code complexity low for building synchronization primitives can be met.

4.5.2 Performance

While high performance is not the primary goal, Octopus should at least provide reasonable performance. The microbenchmarks in this subsection prove reasonable performance compared to existing systems.

The test system, on which the measurements were done, is a TYAN Transport VX50 B4985 PC with two dual-core AMD Santa Rosa CPUs running at 2.8 GHz. The Octopus server and the client ran on different cores on the same CPU package. The preallocation of a big heap as de-

[1] Generated using David A. Wheeler's SLOCCount

scribed in section 3.7.2 and the big size of internal dictionary hash tables reduce garbage collection during the experiments almost completely. Single outliers, due to context switches and other system effects, are removed from the graphs.

The first experiment[2] is a strawman comparison to Redis[111]. Implementation-wise, Octopus is similar to Redis, though simpler and less optimized. The client-server connection on both systems is different. In Redis, the client-server connection goes through a Unix domain socket, while for Octopus it is a regular message channel in Barrelfish. The two test setups are as follows: Redis 2.4.7 runs on Linux 2.6.32 pinned to one core. The provided `redis-benchmark` program plays the role fo the client and performs the measurement. Octopus runs on Barrelfish. A separate client program performs the measurements. On both test setups, the client issues `get` calls with 256 byte payload. The clients measure the achieved throughput of `get` calls on both systems

Figure 4.2 shows that the peak for Redis is at about 90000 ops/sec and for Octopus at around 60000. The scalability of both systems is similar. The performance hit in Octopus is due to ECL^{i}PS^e. Each `get` call involves the ECL^{i}PS^e engine.

A second benchmark[3] therefore measures the overhead caused by ECL^{i}PS^e on `get` calls. The client retrieves a specific record of an increasing number of stored records, up to 1.4 million.

Figure 4.3 shows the latency to retrieve one record out of a varying number of stored records, as shown on the graph's x axis. The "RCP call" line includes the complete time to retrieve a record. The "ECL^{i}PS^e CLP" line shows only the portion of the time spent in the ECL^{i}PS^e engine. The overhead of ECL^{i}PS^e compared to the overall latency it roughly 80%.

The measurements show, that Octopus' performance is reasonable. It can be used to coordinate different applications or synchronize a distributed application without suffering extremely from the performance. Obviously, if high-performance is necessary, an application-proprietary, manually optimized synchronization mechanism based directly on Bar-

[2] The experiment has been conducted by Gerd Zellweger.

[3] The experiment has been conducted by Gerd Zellweger.

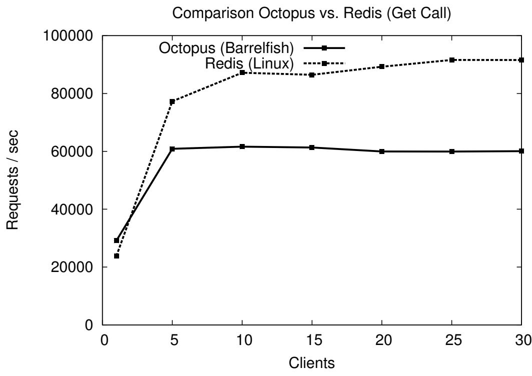


Figure 4.2: Throughput Octopus vs. Redis

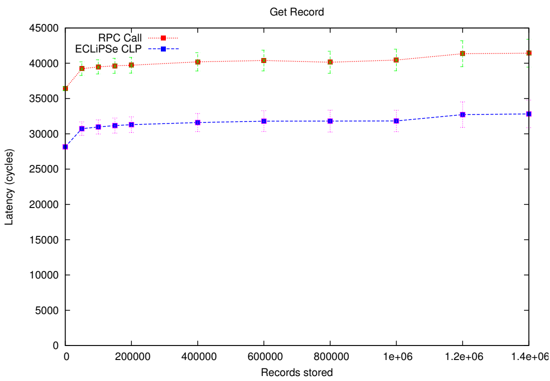


Figure 4.3: Time spent in ECL^{i}PS^e compared to overall time used to retrieve a record.

relfish's message-passing interface is advantageous. A high-level, generic coordination and synchronization service can never achieve the same performance as a mechanism specifically tuned to one application.

4.6 Summary

Octopus has been proven to be useful to solve some of the coordination and synchronization problems found in Barrelfish, as the use-cases demonstrate. The low code complexity involved in implementing primitives on top of it are a net benefit. Octopus' performance costs are acceptable compared to the benefit of having much simpler code to manage distributed applications and dependency resolving of different drivers and OS services. A key insight borrowed from large-scale clusters systems is that it is beneficial to separate coordination from the rest of the system code.

Chapter 5

Device management

This chapter describes the device discovery and management process with Kaluga. Kaluga is the device manager in Barrelfish. Kaluga is based on Octopus records, as described in chapter 4. It is entirely event-driven and keeps all the state in the SKB.

At power-on of a computer, devices, memory and even CPU cores have to be initialized before they can be used. Before that, the operating system has to scan the hardware and check, what kinds of devices are installed in the machine. This is the process of *hardware discovery*. It is typically an ongoing process. The user might plug in more devices at runtime. A common example is a USB device, which the user can plug in and remove at any time.

Once a device is discovered, the OS needs to start the right driver and assign the device to the driver instance. The OS needs to keep track of discovered devices and associated drivers, a task termed *device management*. More concretely, the OS's device manager is responsible for device management.

Part of this work was published in Gerd Zellweger's master's thesis[143], who also implemented most of the code, and in a recent paper[144].

5.1 Kaluga

Kaluga is the device manager in Barrelfish. It coordinates all processes involved in discovering and operating devices, starting at the bottom with CPU drivers 2.2, whose devices are CPU cores, up to "regular" devices, such as an ethernet card driver. Kaluga is based on Octopus (see section 4) and keeps all state in form of records in the SKB.

As the term "device manager" implies, Kaluga is responsible of *managing* the process of discovering devices and *managing* the drivers responsible of initializing and operating specific devices. It does however not access any device by itself and therefore also does not discover any device by itself. Kaluga starts the appropriate drivers at the appropriate time based on the current state. These drivers either explore the hardware, discover devices and add information to the SKB, or they attach to a specific device and initialize and operate it. Kaluga reacts to every hardware-related information added by drivers. Based on the type of information, it starts a new driver or possibly signals an already running one. This is an ongoing process and allows a continuous process of hardware discovery.

To operate correctly, Kaluga needs two types of information. On the one hand, it needs to know what types of devices are installed, such that it can start the right device driver. On the other hand, it needs to know which driver is suitable for a specific device. The former type of information results directly from the device discovery process, while the latter has to be defined by the device driver's programmer, as he is the only one who knows exactly what kind of driver he implemented. This information is stored in *driver mapping files* (see section 5.1.2).

5.1.1 Architecture

Kaluga is implemented as a user-space service on Barrelfish. It is purely event driven and reactive. The implementation uses Octopus records and the event mechanism to get notifications about future records added by drivers. The hardware records are explained further in section 5.1.3.
Kaluga's task is to start the appropriate driver, whenever a new record has been added by a driver. It translates the information contained in the record

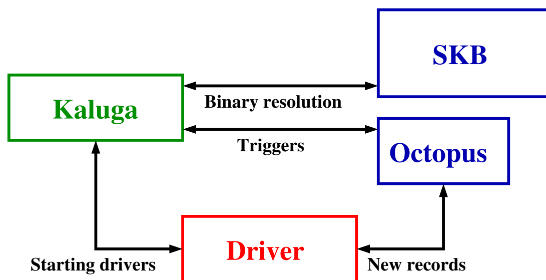


Figure 5.1: Interactions between Kaluga, Octopus and the SKB.

to a driver binary by looking up the driver binary name in device mapping files. The device mapping files are explained in section 5.1.2.

As section 5.2.1 explains, hardware discovery is a recursive process. Drivers add records about discovered devices. This will trigger Kaluga to start appropriate device drivers, which will possibly add more records.

Figure 5.1 shows how Kaluga interacts with Octopus, the SKB and the bus and device drivers. The interaction with Octopus is installing triggers for hardware records and getting notifications about new records. The interaction with the SKB is unifying hardware ID information with a suitable driver binary. Finally, the figure shows that drivers add records about discovered devices.

5.1.2 Driver mapping files

Every device has a physical device ID of some form, depending on the type of device (e.g., PCIe device or USB device). To make a device usable for the system, a specific device driver needs to initialize and operate it. Every device also has at least one driver binary which is able to operate the device and export its hardware capabilities in a form such that the upper layers of the software stack can use it.

Driver mapping files contain a mapping between device IDs and names of executable driver binaries. The driver mapping files form a hierarchy.

There is a main file which may include further mapping files for specific drivers. The driver knows exactly for which device IDs it is suitable. Each driver should have its own driver mapping file, such that it can be included in the main driver mapping file at install time[1]. The driver might specify a range of device IDs which it can handle. In case multiple drivers have a mapping for the same device, the device manager choses one to run. Kaluga loads the main driver mapping file at startup. Because the driver mapping files are written in CLP, each include statement causes the SKB to load the included files as well. This part of the necessary information is now available.
The current format of the mapping files is as follows:

```
pci_driver{
    binary: "e1000n",
    supported_cards:
    [ pci_card{ vendor: 16'8086, device: 16'107d,
                function: _, subvendor: _, subdevice: _ },
      pci_card{ vendor: 16'8086, device: 16'1096,
                function: _, subvendor: _, subdevice: _ } ],
    core_hint: 0,
    interrupt_load: 0.75,
    platforms: ['x86_64', 'x86_32']
}.
```

The device ID is defined here by the `vendor`, `device`, `function`, `subvendor` and `subdevice` fields. These values are device characteristics which appear whenever the PCIe driver scans the bus for devices. These are not the only characteristics provided by the PCIe hardware, but the ones which make most sense to identify devices based on *what* they are, rather then *where* they are located. Fields like PCIe `bus` or PCIe `device` numbers would make less sense, as these values depend on the slot, into which the user installed the device. A device driver will never know that in advance and should only care about constant (and PCIe-slot independent) values as found in the driver mapping files.

[1] Barrelfish has no facility to “install” a driver yet. This basically means, that a driver developer has to add the necessary mapping entries by hand, at the moment.

Whenever Kaluga receives a notification, that a new PCIe device was discovered, it considers all the entries by a unification algorithm and finds the most appropriate driver to start.

`interrupt_load` is for future use and should provide an initial hint on the expected interrupt load. This allows Kaluga to reason about that and to distribute drivers, with a high expected interrupt load, to different cores.

5.1.3 Hardware records

Kaluga queries hardware records to learn about installed hardware. It issues a `get_names` call and installs a trigger, such that it gets future records, whenever a new device has been discovered. It makes use of the regular expression facility provided by Octopus to specify a range of names, all related to discovered hardware. The format of the records is as following:

```
r'hw\\.some_name\\.[0-9]+'
```

Basing device management on Octopus records has a number of advantages. First, dependencies are resolved by the event mechanism. For example, the PCIe driver first initializes the basic PCIe infrastructure, before it starts adding device records. This ensures, that specific device drivers start only after the basic configuration is done. Second, the driver framework is flexible and modular. Whatever device might be found in the future, it will work, as long as a driver is available. It does not depend on the implementation of Kaluga. In Barrelfish, every device is treated like a regular device with a device driver, even CPU cores. By adding records also for CPU cores, Kaluga starts the CPU driver for every discovered core, although the mechanism of actually starting the core is somewhat different than for PCIe devices. Finally, this driver architecture naturally supports hotplugging. Independent, of whether the initial hardware scan finds a device or a hotplug event later triggers the bus driver to initialize the new device, the bus driver will add a hardware record for the discovered device. Because Kaluga keeps the trigger for ever, it will be notified about the new device and will start a driver, whenever it receives an event. This works even for cores. Similarly, Kaluga can easily be extended to

support deletion of records. In this case, it might shut down the associated device driver and possibly notify further dependent services.

5.2 Hardware discovery

Hardware discovery refers to the process of learning, what kinds of hardware are installed in a given machine. Depending on the type of hardware, there are different techniques to be used to discover installed hardware. This is not a problem, because Kaluga only *coordinates* hardware discovery. It is the responsibility of the concrete bus and device drivers to use the right technique to discover hardware.

At the early stage of booting, the hardware discovery process has to check for the availability of basic hardware features. In Barrelfish, this is partly done by the basic system and partly by Kaluga, as explained in more detail in section 5.2.1. Based on that, the discovery process starts the right platform driver which finally scans the installed hardware to learn which drivers need to be started.

Hardware discovery needs knowledge about the current state. Only after the architecture is known, concrete hardware features can be queried. Likewise, only after knowing, whether a PCIe bus is available, can PCIe devices be scanned.

This section explains the hardware discovery life-cycle as well as the basic information required for it to work correctly. A short evaluation shows that Kaluga, based on Octopus and the SKB, is implemented with few lines of code while providing a rich functionality.

5.2.1 Hardware discovery life-cycle in Barrelfish

On x86-based system, the BIOS runs only the bootstrap core, on which the OS gets started-up. All the other cores remain in a halted state and the OS is responsible to start them. Devices are not fully configured, only the really necessary ones to load the OS get configured by the BIOS.

The hardware discovery life-cycle starts by discovering the base architecture and basic hardware features on the single core, on which Barrelfish

starts up. It adds all basic information about the architecture to the SKB, which is already running at that stage (see section 3.5.5).

As an example, the first CPU driver and monitor (see section 2.2.2) already know on what architecture they are running. This information is known (at least at a high-level), because the right binary is loaded and executed by the bootloader. The monitor adds a fact[2] to the SKB, saying of which architecture at least the first core is.

Kaluga, running on the first core, queries the CPU core for the ACPI availability flag. Based on ACPI availability, Kaluga starts the ACPI driver. On x86-based systems, ACPI is the starting point to find “root” pieces of hardware, such as PCIe root bridges, I/O APICs and CPU cores. The “root” pieces of hardware have a separate driver which Kaluga starts based on the new record added for the “root hardware”. It is the responsibility of the concrete drivers to initialize the “root devices” and to further query them, whether more hardware is attached under them. From there it continues with the rest of the hardware by letting drivers scan the device tree of their responsibility.

To continue the example, Kaluga starts a PCIe bus driver, if the ACPI driver discovered a PCIe root complex. The PCIe bus driver enumerates devices under this root, performs a basic address configuration (see chapter 6) and generates new records. This triggers Kaluga to start PCIe drivers, such as, for example, a PCIe USB host controller. This driver in turn enumerates the USB bus and adds further records causing Kaluga to start USB device drivers, and so on. CPU cores beyond the first one are treated the same as regular devices. Whenever Kaluga receives a record event for a core (typically from ACPI), it starts an appropriate kernel (or “CPU driver” in Barrelfish parlance) based on the driver mapping database.

The discovery and driver startup process is recursive. Starting a driver is caused by previously adding a record by another driver. Every driver can, in turn, add more records, which will cause Kaluga to start more drivers. This is an ongoing and never ending process. Figure 5.2 shows the hardware discovery life-cycle.

[2] In this case a regular fact, not a record.

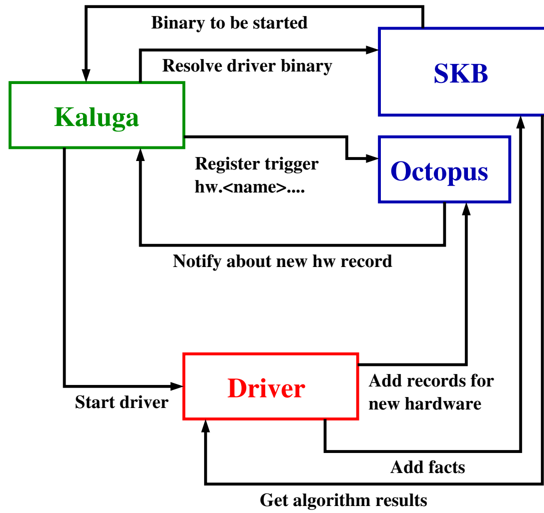


Figure 5.2: Hardware discovery life-cycle.

5.2.2 View hotplugging as the default case

Having such an event-based device manager, which interacts with the SKB, is the basis for hotplugging devices. In fact, all devices in Barrelfish are treated as hotplugged devices at least from the perspective of device management.

While it is common to support hotplugging USB devices, it is less common to support CPU hotplugging in current commodity systems. The driver framework of Barrelfish offers this functionality in a natural way. Kaluga's architecture is the first step to support hotplugging in the system. Even more importantly, it is the first step of supporting a completely distributed system structure. This kind of device management and ongoing device discovery process, in combination with treating CPU cores as regular devices with a device driver, allow adding and removing any device – including cores – at runtime.

Obviously, drivers (and finally applications) need to support hotplugging as well, a feature, which leads to many interesting questions, should it be supported at its full flexibility in a distributed-systems-like OS. Drivers (and applications) need to react correctly on hotplug events. Especially the removal of devices may cause interesting interactions between different running services that depend on the device. Kaluga, however, is the first enabling step towards this direction.

5.2.3 Minimize basic architecture and platform information

Although most of the hardware knowledge can be queried and derived by looking at different pieces of information, some basic "hard-coded" information is necessary to boot a computer. First of all, the system's bootloader needs to load the right binary for the architecture, which means that this information is known, at least to the bootloader. Depending on the hardware system, knowledge about basic memory mappings or available mechanisms to query hardware information is necessary to load the right drivers which finally scan the hardware. For example, knowing that the architecture is "x86_32" is not sufficient, if the code runs on a SCC.

The memory architecture is completely different compared to the memory architecture of a regular "x86_32" computer.

Supporting a flexible, modular, distributed-systems-like OS that targets an unforeseeable range of diverse hardware, demands that the "hard-coded" or a priori knowledge, even of basic information, is minimized. This makes it possible to hotplug complex devices, which provide more cores (such as an SCC, for example), in the future.

At startup of Barrelfish, only the CPU driver, the monitor, the memory server, the device manager and the SKB run on the bootstrap core. At this stage it is already clear, what type of architecture the system has, because the right binaries have been loaded by the bootloader. This basic information is, to some extent, "hard-coded" and can go directly to the SKB. It is however the only "hard-coded" information necessary to decide how to continue. For example, it is enough information to decide, whether the ACPI feature flag needs to be queried or not and finally, whether the ACPI driver should be loaded or not.

5.2.4 Device information

There are two classes of device information: the high-level information, that a device is installed and the detail knowledge about how it is working. In order to start the right driver, Kaluga needs to know about a device's availability. It does however not need to know exactly how the device works.

The detailed device knowledge can be in a format suitable to the device driver. This knowledge gets mainly processed by the driver and should be optimized for its needs. In contrast, the high-level information, that a device is installed, should be as generic as possible. This information is added in form of records (as described in section 5.1.3) and used by Kaluga to get notifications. The set of different records added should be as small as possible, because for every record format, Kaluga needs to install a trigger to monitor it.

The PCIe bus driver is a complex hardware configuration example, which needs a deep hardware knowledge (see chapter 6). The hardware knowledge is gained by scanning the PCIe hardware and storing basic in-

formation using different facts to the SKB. Later on, PCIe uses these facts to run the PCIe address allocation algorithm. This knowledge is mostly local to the PCIe bus driver and PCIe device drivers. But Kaluga only needs to know that PCIe devices are installed, such that it can start the appropriate device driver. The basic PCIe records therefore only needs to provide information about the vendor, device ID and so forth, as described in section 5.1.2. It does not care about base addresses or requesting physical memory sizes.

5.3 System Bootstrap

Booting an OS is a complex task. Hardware has to be initialized and exported to clients, drivers and OS services have to be started. On modern hardware, other CPU cores also need to be started by the OS. So far, Kaluga and Octopus proved very useful in simplifying the bootstrap process in Barrelfish. The current solution builds on both the name service and the Kaluga device manager.

As the hardware life-cycle section explained (see section 5.2.1), a significant amount of the bootstrap process depends on discovering and initializing devices by starting the appropriate device drivers. Drivers not only start and operate the device, they also register with the name service to make the device available to the rest of the system. Depending services wait for required service references before they register with the name service. This way, the OS boot process is well coordinated. The uniform abstraction of dependencies behind Octopus records and triggers has significantly reduced special-case code in many parts of the OS.

5.4 Evaluation

The evaluation in this section is of a qualitative nature. First, the section evaluates whether Kaluga works correctly by checking that drivers for available devices get loaded. Second, the section evaluates the number of lines of code necessary to implement Kaluga. It is one of the thesis'

main goals to keep code complexity as low as possible. Performance is not evaluated, as the bootstrap process depends on various things not controllable solely by Kaluga.

5.4.1 Correctness

"Correctness" in the case of a device manager means that the right drivers should be started according to the driver mapping database, whenever a new device was found by some previous device driver. It is therefore necessary to manually inspect the system configuration and to derive expectations on which drivers will get loaded by Kaluga.

Evaluating whether Kaluga works correctly was done by booting Barrelfish on our different x86_64-based development machines and by ensuring that all the drivers, for which a device is available in the system, get started. All of the machines support ACPI and all of them have at least one PCIe bus and at least one e1000 network card. Kaluga should therefore start the PCIe bus driver and at least the e1000 driver. By manually checking the booted system, it became clear that the drivers got started correctly by Kaluga.

5.4.2 Code complexity

Table 5.1 shows a breakdown of the LOCs used to implement Kaluga and the device mapping files. Kaluga is implemented in only 759 lines of C code[3]. Additionally, there are 78 lines of ECLiPSe code in form of driver mapping files and unification algorithms to match stored device IDs with device IDs passed to Kaluga by record events. Kaluga loads these files at startup. This is why the LOCs account for Kaluga in this evaluation. The small device manager is capable of fully controlling device drivers. The ECLiPSe approach, used to reason about suitable driver binaries, offers a great flexibility on deciding which drivers to start.

Currently, Barrelfish only has five drivers as separate modules which can be started on demand by Kaluga. This explains, why the driver map-

[3] LOC counts were generated using "SLOCCount" by David A. Wheeler.

Functionality	C	ECL^{i}PS^e
Kaluga	759	
Unification algorithm in driver mapping files		19
Driver mapping entries for 5 drivers		36
Data structure definitions		23
Total	759	78

Table 5.1: Lines of code

ping files only consist of 78 lines of ECL^{i}PS^e code. The driver modules have to appear as multiboot modules in the `menu.lst` file, otherwise Kaluga has no access to the binaries[4].

5.5 Related work

Devices in Linux are represented by entries in the `/dev` directory. Early versions of Linux used a statically populated directory with a fixed name to major/minor number mapping according to the "Linux Assigned Names and Numbers Authority" (lanana)[77]. This approach had a number of problems[75]. First, devices were bound to names in `/dev` according to the enumeration order. The device, which was found first, got the first name of this class of devices. In the case of USB devices, for example, it means that USB devices can get new names, if other USB devices are plugged-in or removed. Second, Major and minor numbers are only 8bit values. The static mapping of major numbers to device classes limit the classes of devices, and if vendors invent new devices, it becomes increasingly hard to assign a new static major number. Third, `/dev` became too big. A statically populated Red Hat 9 has over 18000 entries[75]. Recent versions of Linux use `udev` which populates the `/dev` directory with device nodes, whenever devices are actually discovered by the kernel. In contrast to the older `devfs`, `udev` does not enforce any name policies in the kernel, but follows name policy rules defined in configuration files.

[4] At the moment, Barrelfish does not have a filesystem. If it had one, it still would have to be available from a RAM disk together with the disk driver, because these modules are a requirement to load files from a disk.

FreeBSD uses a kernel-based DEVFS[68] on which device nodes are created on demand whenever a new device gets discovered. The devd daemon[83] receives events from the kernel when new devices are discovered or devices disappear. It is able to configure devices and to load device drivers by considering a configuration file telling which driver is suitable for which device. devd's configuration files allow the administrator to define arbitrary commands to be executed on every *attach*, *detach* and *nomatch* event. The *nomatch* event is generated, if no currently available (loaded or compiled-in) driver claimed the discovered device.

5.6 Summary

Device management on a distributed systems-like operating system requires a flexible approach of device management. The distributed systems nature assumes that nodes (like CPU cores or devices) join or leave the system during runtime. Managing hardware and system bootstrap is complex enough, but allowing hotplugging and removal of any kind hardware complicates it even more.

Further, Barrelfish targets a wide range of diverse hardware. It is unknown, how the hardware looks like and how architectures ar going to evolve. The effort of manually adapting software to new architectures, as they emerge, is too high. Instead, the system should adapt itself to the actual hardware. For device management, this means, that the device manager should not assume any knowledge. Further, it should not need to deal with concrete hardware knowledge. Keeping the management at a high-level, such that it really only performs the task of management, abstracts the device manager completely from the hardware. The abstracted high-level unification mechanism to match driver binaries and devices works on all hardware platforms, even on future hardware.

With Kaluga, Barrelfish's device manager based on Octopus, a first step towards a distributed system with hotplugging as the default case could be realized. Kaluga is flexible and easily extensible, because the reasoning is based on Octopus records and on knowledge in form of CLP facts. Furthermore, the evaluation shows, that Kaluga has a low code com-

plexity, which is one of the main goals of the thesis.

Chapter 6

Declarative PCI configuration

This chapter presents the first case study of the SKB. To validate the claim that the CLP approach can be used successfully for low-level hardware configuration, I implemented the PCIe configuration problem as a high-level declarative CLP algorithm. The PCIe configuration problem is one of the most complex hardware configuration problems found in current systems. Therefore, it is a good case study to demonstrate that even complex problems can be solved by means of CLP programs.

Programming PCIe bridges in a modern PC is a surprisingly complex problem, and it is getting worse as new functionality such as hotplug appears. Existing approaches use relatively simple algorithms, hard-coded in C and closely coupled with low-level register access code, generally leading to suboptimal configurations.

The PCIe driver implemented for this thesis follows a radically new approach. Along the discussion of policy/mechanism separation (see section 3.4.1), this PCIe driver separates hardware configuration *logic* (algorithms to determine configuration parameter values) from *mechanism* (programming device registers). The latter is implemented in C, and the

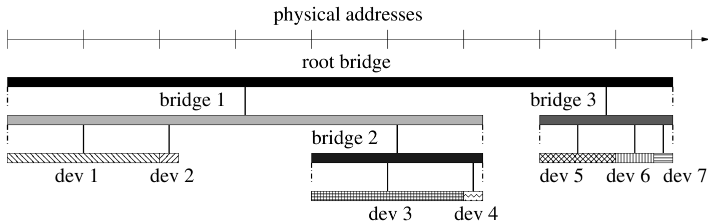


Figure 6.1: Example PCI tree with one root, three bridges, and 7 devices, showing the decoding of addresses from one of the three physical memory spaces (e.g., non-prefetchable). Bridge base addresses and limits are bounded by the union of the base and limit addresses of their children.

former as concise CLP algorithm in the SKB. The PCIe driver implements full PCI configuration, resource allocation, and interrupt assignment.

The work presented in this chapter has been published[116, 117].

6.1 Introduction

Configuring physical address regions of the PCIe bus for all devices and bridges is a complex problem. Many dependencies between base addresses of devices and bridges have to be met and special cases have to be handled. Fortunately, the PCIe allocation specification follows clear rules and, as stated in section 3.3.4, this makes it a candidate for a CLP-based solution. The same is true for allocating and routing interrupts. Existing operating systems code uses relatively simple algorithms to configure the PCIe bus. These algorithms are simple by the necessity of being hard-coded: they require low-level access to device registers to achieve their goals, and usually run early at system start-up within the OS kernel.

Figure 6.1 illustrates a simplified PCIe configuration. The OS code must allocate memory regions to each PCIe device and each PCIe bridge in the bus hierarchy, in such a way that every device receives correctly-sized areas of physical addresses in two different address spaces (I/O and

memory mapped) and two distinct address regions in the memory mapped case (prefetchable, and non-prefetchable). These areas must all be aligned to device-specific boundaries, may not overlap, and should fit into the total amount of physical address space available for such hardware in the system.

This allocation problem is particularly hard, because there are numerous restrictions on device allocation: certain devices must be placed at a fixed address, others incorrectly decode addresses not assigned to them, and platform hardware components such as ACPI sometimes reserve regions of physical address space, which means that the address ranges must be allocated around these "holes". Furthermore, the list of problems varies from machine to machine, requiring the allocation code to adapt automatically to the underlying hardware.

Most existing operating systems deal with this problem with simple algorithms in C. Special cases are intermangled in the main allocation code. The result is complex and hard to debug, and (as the evaluation in section 6.5 shows) can lead to unpredictable and inefficient allocation of space. In some cases (such as Linux on Intel platforms) the OS does not even try to solve the full allocation problem, instead it relies on the platform BIOS to provide an initial allocation, which is difficult to change.

In this chapter I show that pushing the allocation algorithm *logic* into the SKB and separating it from the configuration *mechanism*, which writes the derived values into the base address registers, leads to much simpler, more maintainable and easily portable code. I exploit the unification facility of ECL^{i}PS^{e} to turn PCIe information into knowledge about bus hierarchy and I model the actual hardware-given constraints and allocation rules as constraints on base addresses per PCIe device. Only the register access functions are written in C. These are extremely simple, because they just write the derived addresses into base address registers. The result is a PCIe bus driver, which completely solves a complex problem with few lines of code.

6.2 Background: PCI allocation

Configuring the PCIe bridges found in a typical modern computer is emblematic of a wide class of hardware-related systems software challenges: it involves resource discovery followed by the allocation of identifiers and ranges from compact spaces of identifiers and addresses. More importantly, a range of hardware bugs and/or ad-hoc constraints on particular devices lead to a plethora of special cases which make it hard to express a correct algorithm in imperative terms. Worse, new hardware (whether system boards or devices) appears all the time, and system software must continue to work, or evolve to handle new cases with a minimum of disruptive engineering effort.

This section describes the PCI programming challenge in detail by starting with the "idealized" problem, which appears relatively straightforward, and by progressively introducing the complexities that, combined, are the reason that even modern operating systems only partially solve the problem.

6.2.1 PCI background

A PCI (or PCI Express) interconnect is logically one or more n-ary trees whose internal nodes are *bridges* and whose leaves are *devices*[25, 104]. The root of each tree is known as a *root bridge* or *root complex*. Connections in the tree are known as *buses* (in legacy PCI they are electrical buses, whereas in PCI Express the bus is a logical abstraction over point-to-point messaging links). Non-root bridges are said to link *secondary buses* (links to child bridges and devices) to a *primary bus* (the link to the bridge's parent). High-end PCs often have two or four root complexes, and hence multiple PCI trees within a single system. Non-root devices can be attached to any bus in a PCI interconnect. Each device implements one or more distinct *functions*. A PCI "function" is in fact what most people think of an independent "device" which has its own bus address represented by the bus number, the device number and the function number and which operates independently of other functions.

Driver software on host CPUs accesses PCI functions by issuing mem-

ory reads and writes or (in the case of the x86 architecture) I/O instructions. These requests are routed down the tree by the bridges, before being decoded by a single leaf device. Each function decodes a portion of the overall memory and I/O address spaces using a mapping that is configured by the host system through standard PCI-defined registers on each bridge and function.

Each function of a non-bridge device may decode up to 6 independent regions of either memory or I/O address space. These regions are defined and configured by *base address registers* (BARs) implemented by each function. The PCI driver queries each BAR to determine its required size, alignment, address space (memory or I/O), and, in the case of a memory-space BAR, whether the memory is *prefetchable* or *non-prefetchable*, and then reprograms the same registers to allocate definite addresses. Although it goes against strict PCI terminology, in the rest of this chapter the term "device" denotes a PCI function, i.e., a single logical device with up to 6 BARs.

Bridges also decode addresses to route requests between their parent and secondary buses. Unlike other devices, however, bridges use three pairs of base and limit registers instead of BARs, one each for prefetchable memory, non-prefetchable memory, and I/O space. Each bridge therefore decodes 3 independent, contiguous regions of IO or memory address space. The addresses used by every device below a bridge (including bridges on secondary buses) must lie within these three regions.

In summary, a host CPU accesses a PCIe device by issuing a transaction on the system interconnect with a physical address that lies in a region decoded by the root bridge of the corresponding PCIe tree. This is routed down the tree by bridges; at each level, each bridge on a bus compares the address issued by the CPU to the ranges defined by its base and limit registers. If it matches, the bridge forwards the request to its secondary bus. Each device on a bus compares the address to the regions defined by its BARs, and if the address matches, consumes it and generates a reply.

The PCIe programming problem is to configure the base and limit registers of every bridge, and the BARs of every device function, to allow all the hardware registers of every device to be accessible from a CPU. As Figure 6.2 shows, this can be achieved in many different ways, leading to

different usage of the available physical address space and different device locations in that space.

The next section specifies the requirements for any PCIe programming solution, starting with the basic properties of a solution in the "ideal" case, and progressively refining the list by adding real-world complications.

6.2.2 Basic PCI configuration requirements

Every bridge in a correctly-configured PCI tree decodes a subrange of the addresses visible on its parent bus. In order for all devices behind a bridge to be reachable, PCI requires that:

1. The *bridge window*, defined by its base and limit registers, must include all address regions decoded by all devices and bridges on the secondary bus.

In order that a request is forwarded by at most one bridge, sibling bridges sharing a bus must decode disjoint address ranges. Since a bus may contain both bridges and devices, all bridges and devices on a given bus must decode disjoint address ranges within the range of the parent bridge. This applies in all of the address spaces:

2. Bridges and devices at the same tree level (siblings) must not overlap in either memory or I/O address space.

3. The prefetchable and non-prefetchable memory regions decoded by a bridge or device must not overlap.

Regions of addresses in PCIe must also be aligned. For a BAR, the base address must be "naturally" aligned at a multiple of the region's size. Similarly, a bridge's base and limit registers also have limited granularity, giving us the following alignment constraints:

4. BAR base addresses must be naturally aligned according to the BAR size.

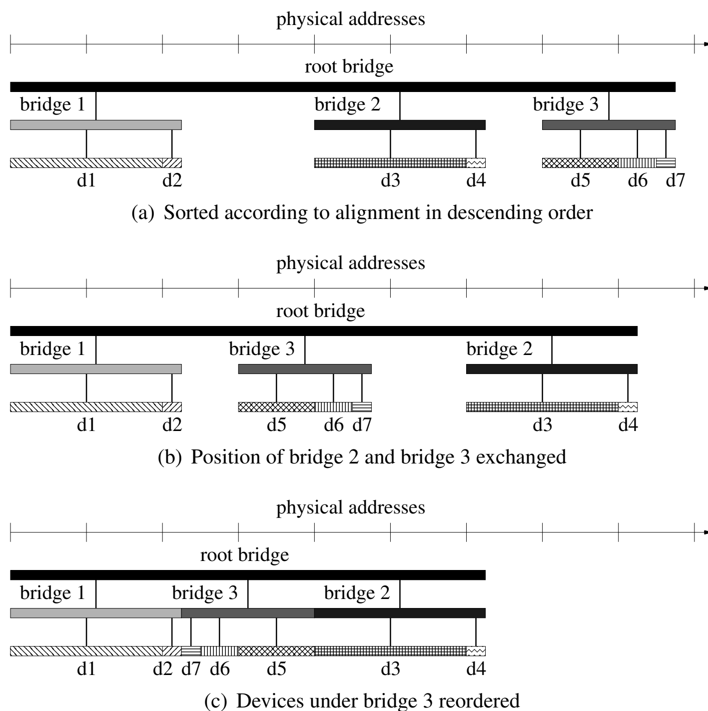


Figure 6.2: Alternative PCI configurations (only memory space resources are shown)

5. Bridge base and limit register values for both memory regions must be aligned to 1MB boundaries.

6. Bridge base and limit register values for the I/O region must be aligned to 4kB boundaries.

These requirements constrain the possible locations of device BARs and child bridge base and limit registers within the region decoded by the parent bridge, potentially leading to gaps in address space for padding, as in Figures 6.2(a) and 6.2(b).

As described so far, configuring a PCIe tree is a non-trivial problem, but can still be efficiently programmed by, for example, executing a post-order traversal of the PCIe tree, sorting devices and bridges by descending alignment granularity, and allocating the lowest suitable address range in the appropriate address space at each step. Unfortunately, requirements like the need to align region addresses make it non-trivial to generate configurations that make efficient use of address space, and the simple post-order traversal results in a solution like that in Figure 6.2(a) where large padding holes need to be inserted between devices.

The following subsections progressively list the additionally complications that make an imperative solution to this problem a considerable programming challenge.

6.2.3 Non-PCIe devices

The first complication is that certain non-PCIe devices and hardware registers appear at fixed physical memory addresses inside the region allocated to a PCIe root complex – for example, IOAPICs and other “platform” devices on PC systems. The presence and location of these devices vary from machine to machine and may be discovered through platform-specific mechanisms such as ACPI[59]. For correct operation, no PCIe device should be configured to decode such an address region.

7. Devices must not decode reserved regions of physical address space given by, for example, ACPI, or used by other known non-PCIe devices such as IOAPICs.

6.2.4 Fixed-location PCIe devices

Some PCIe devices may be initialized and enabled by platform firmware at early boot time, for example USB controllers, network interfaces, or other boot devices. Naïvely reprogramming the BARs of such devices may lead to machine check exceptions or crashes since the device may be active, and performing DMA operations. Most operating systems avoid reprogramming the BARs of such devices, which means that their existing address assignment must be preserved. This further constrains the address ranges usable by parent bridges.

8. Certain PCIe devices determined at boot cannot change location, and must retain addresses assigned to them by the BIOS.

6.2.5 Quirks

Hardware has bugs, and both devices and bridges can report incorrect information, fail to support valid resource assignments, or behave incorrectly when specific register values are programmed. These problems are known as PCIe “quirks” and affect a wide range of shipping devices – the Linux 2.6.34 kernel lists 546 quirks – leading to a collection of workarounds in commodity operating systems. As Table 6.1 shows, in the Linux kernel there were between 20 and 50 commits to the file `quirks.c` (which contains workarounds for buggy or otherwise anomalous PCIe devices) every year since 2005. Since new hardware appears every year, and does not seem to be any less complex or buggy with time, this trend is likely to continue and therefore a clean, portable, maintainable, and easily evolvable way to handle quirks in software is desirable.
The PCIe quirks currently handled by the Linux kernel mostly fall into the following categories:

- devices that provide incorrect information about their identity as bridges or non-bridges;
- devices which decode more address range than advertized, or which decode address regions not assigned to them;

Year	Number of commits
2005	26
2006	47
2007	49
2008	43
2009	42
2010	23

Table 6.1: Changes to Linux quirks.c

- standard devices which are hidden by platform firmware, but which could otherwise be normally used;
- undefined device behavior (data loss on the bus, reduced bandwidth, system hangs, etc.) when particular (and otherwise valid) values are written to the device's configuration registers.

In the latter case, the PCIe configuration process must ensure the problematic register values are never written, which imposes additional constraints on valid address assignments. Thus:

9. Configurations that would cause problematic values to be written to registers on specific devices must be avoided.
10. Incorrect information from PCIe discovery must be corrected before calculating address assignment.

A further complication arises from the ambiguity as to whether some hardware is a PCIe device or not. For example, on some (but not all) contemporary PC systems, IOAPIC registers appear to software as the BAR of a PCIe device, but the IOAPIC is also defined as a "platform device" whose location in the physical address space can also be configured using other mechanisms (such as setting the base address value by ACPI mechanisms), or in some cases may not be changed as this would violate assumptions in firmware such as ACPI or would simply crash the machine, because, while

routing interrupts, the IOAPIC cannot be reached anymore. On such systems, the BAR corresponding to the IOAPIC must be programmed with a fixed value to ensure it is consistent with other assignments of the address. This can be summarized as follows:

11. Certain platform devices appearing within a BAR of a regular PCIe device or bridge must be treated as PCIe devices with a fixed address requirement.

6.2.6 Device hotplug

Hotplugging, the addition or removal of PCIe devices at runtime, raises another challenge. When a device is plugged in, the OS is notified by an interrupt from the root bridge, and must allocate resources to the BARs of the newly-installed device before it can be used. However, this may require reconfiguring and/or moving the address allocation of bridges and other devices in order to make enough address space available for the device, since it was not present at system boot.

Changing the resource allocation of existing devices requires the driver to temporarily disable the device, potentially saving its current state first. After the new resources are programmed to the BARs, the driver needs to restart the device using the newly allocated resources. Depending on the device, it may need to bring the device to the saved state.

This is a disruptive process and, worse still, may not be supported by all devices, so the reallocation of resources which occurs on hotplug typically attempts to move the fewest possible existing devices and bridges.

12. Configuration should minimize the disruption caused by future hotplug events as much as possible.

13. Hotplug events should cause the minimal feasible reconfiguration of existing devices and bridges.

14. Hotplug-triggered reconfiguration may not move devices whose drivers do not support relocation of address ranges.

6.2.7 Discussion

It should by now be clear that PCIe configuration is a somewhat messy problem characterized by a large (and growing) number of hardware-specific constraints which nonetheless have effects which propagate up and down the PCIe tree. Consequently, most “clean” solutions written imperatively in a language like C sooner or later fall foul of an exception which can greatly complicate the code, compromise its correctness, reduce the efficiency with which it can manage physical address spaces, and in some cases prevent it from supporting the full PCIe feature set.

The PCIe specification[25, 104] describes the mechanisms and requirements for correct configuration of a PCIe system, but does not specify any particular algorithm to be used in this process, leading to a variety of different (usually incomplete) solutions in current systems. These solutions are being iteratively refined and improved to handle more complex scenarios such as device hotplug[89, 134], leading to greater complexity.

A resource allocation algorithm for a hierarchical tree structure such as PCI has been patented by Dunham[38]. This algorithm sorts devices with fixed requirements according to their base address in ascending order, and all other devices according to their alignment requirements (size) in descending order. It then allocates resources to devices and bridges using a first-fit strategy starting at the lowest-level secondary bus, allowing it to determine the size requirement for the lowest-level bridge. Once its size is set, a bridge is then treated as a fixed-size device for allocation at the upper levels, and placed using the same first-fit allocation. Bridges are considered to have fixed address requirements if a device at any level below the bridge has a fixed requirement. As it encodes a specific traversal of the resource tree, this algorithm is roughly comparable to the postorder traversal discussed in section 6.5.5 and used in varying forms by several current systems.

Most current operating systems, including Linux[115, 134] and FreeBSD[13] on x86-based platforms, rely on platform firmware (BIOS or EFI) to allocate resources to most devices before the OS starts, and then run one or more post-allocation routines[12] to correct any problems in the allocation, allocate resources to devices left unconfigured by the firmware,

and handle known quirks as devices are discovered and started. This approach cannot guarantee success (though it often works): if a bridge is programmed with an address region that is too small to allocate all the devices behind it, there may be no way to grow the size of the bridge's address region without moving other bridges, and thus some devices behind the bridge will be rendered unusable despite sufficient address space being available overall. This problem is exacerbated by device hotplug, as it is impossible to predict at start-up the required size of all devices. Even so, this simplistic allocation strategy leads to substantial code complexity: the complete PCIe drivers of x86 Linux and FreeBSD account for approximately 10k and 6.5k lines of C code respectively, and device-specific quirks account for an additional 3k lines of code in Linux.

On hardware platforms other than x86 (such as Alpha/AXP), the firmware does not implement PCIe configuration, and Linux instead performs a complete allocation using a greedy approach: devices are sorted by their requested size in ascending order, and resources are allocated for each device in that order[115]. This can also lead to unusable devices behind a bridge, due to a suboptimal ordering of devices causing a shortage of address space. Note also that very little code is shared between this implementation and the one for the PC platform: bug fixes or feature enhancements for one architecture may not be easily applied to another.

Until recently, Microsoft Windows used a similar strategy to x86 Linux and FreeBSD for PCIe configuration, running a fix-up procedure to correct deficiencies in the firmware allocation. As with Linux and FreeBSD, this was unable to resize or change the address regions decoded by bridges, leading to potentially unusable devices[90]. Windows Vista and Server 2008 introduced a new re-balancing algorithm[89], allowing a bridge's resources to be modified according to the needs of its secondary bus, and increasing the likelihood that all PCIe devices could be configured. However, this requires additional driver support for re-balancing, and the iterative approach can lead to highly complex multi-level re-balancing. Multi-level re-balancing is a potentially complex operation because increasing a bridge's window size can require the bridge to be moved to a new address region, in turn requiring more space from the parent bridge due to address alignment constraints. In the worst case, multi-level re-balancing can lead

to a complete permutation of the PCIe tree.

6.3 PCIe resource allocation

The previous section detailed the PCIe configuration problem and current approaches to solving it. This section describes the implementation of PCIe configuration in Barrelfish, and the following section 6.4 describes a solution to the closely-related problem of interrupt allocation, before evaluating both in section 6.5.

PCIe resource configuration can be viewed as a constraint satisfaction problem. For a given system the variables are the base address allocated to each device BAR, and the base and limit of each bridge for each memory region it decodes. A correct solution may be expressed as an assignment of integer values to these variables satisfying a series of constraints: alignment, sizes, and non-overlap of regions.

The difficulty in PCIe resource allocation arises from satisfying these complex constraints. Such complexity is difficult to manage in a low-level systems language like C, but fortunately its runtime performance is not critical to the functioning of the system as a whole. This gives the freedom to reformulate it in a declarative language, where the challenge becomes closer to defining *what* result is required, than *how* the result is to be produced.

The implementation of the PCIe resource configuration algorithm is a constraint logic program. This program operates on a high-level data structure representing the PCIe tree, consisting of numeric variables and constraints between them that determine the possible solutions. Rather than worrying about how to allocate concrete addresses to bridges and devices, it is important to specify the correct set of constraints to guide the CLP solver. Before explaining the constraint logic in detail, the next section describes the separation between C and CLP code.

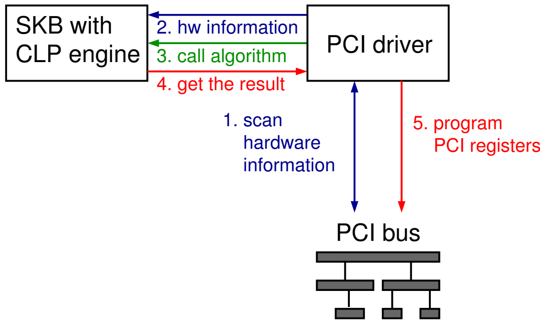


Figure 6.3: Interaction between the PCIe bus driver and the SKB.

6.3.1 Approach

The PCIe driver in Barrelfish explicitly separates the PCIe configuration algorithm, expressed in CLP and running in a user-space service, from the register access and device programming mechanisms, implemented in the usual C code as part of the PCIe subsystem of the OS. This has several advantages. First, it decouples the details of the configuration algorithm from the device access code, allowing to exchange and evolve the algorithm independently of the device access mechanisms. Second, the algorithm is expressed only in terms of the generic PCIe bus – all architecture-specific details are confined to the device access code, or to quirks expressed independently of the main logic. This makes the allocation algorithm portable, because it only operates on high-level facts about the PCIe devices, bridges and memory regions. Finally, the device programming code written in C remains small, simple and robust, reducing the likelihood of bugs. The CLP code is loaded and executed in the SKB.

Figure 6.3 shows the steps performed to configure the PCI bus using the declarative algorithm running in the SKB. The explanation below of the steps taken during configuration of the PCI bus refer to this figure.

The PCIe driver performs device discovery as the first step in configuring the PCIe bus (figure 6.3). The location of root bridges is determined

by platform-specific mechanisms such as ACPI[59]. The driver then walks the entire bus hierarchy, determining the complete set of bridges, devices and BARs that are present by reading out PCIe registers. During this step it also assigns bus numbers to un-numbered bridges and disables address decoding such that the newly computed addresses can later be safely programmed. As part of this pass, the PCIe driver inserts high-level ECL^{i}PS^{e} *facts* in the SKB (step 2 in the figure). These facts describe the set of present bridges, devices and BARs, according to the following schema:

```
rootbridge(addr(Bus, Dev, Fun),
          childbus(MinBus, MaxBus),
          mem(Base, Limit)).

bridge(pcie | pci,
       addr(Bus, Dev, Fun),
       VendorID, DevID, Class, SubClass,
       ProgIf, secondary(BusNr)).

device(pcie | pci,
       addr(Bus, Dev, Fun),
       VendorID, DevID, Class, SubClass,
       ProgIf, IntPin).

bar(addr(Bus, Dev, Fun),
    BARNr, Base, Size,
    mem | io,
    prefetchable | non-prefetchable,
    64 | 32).
```

These facts encode all information needed to run the PCIe configuration algorithm. A root bridge is identified by its PCIe configuration address (bus, device and function number), the range (minimum and maximum) of bus numbers of its children, and its assigned physical memory region. Bridges and devices are identified by their address, and carry standard identifiers for their vendor, device ID, device class and subclass, and programming interface. A bridge also includes the bus number of its secondary bus, and a device includes the interrupt pin which it will raise (which is used by

the interrupt allocation routines described in section 6.4). Finally, for each BAR it stores its base address (which may have been previously assigned by firmware), required size, region type, and whether it is a 64-bit or 32-bit BAR.

After creating the facts, the PCIe driver causes the SKB to run the configuration algorithm to compute a valid allocation (step 3 in figure 6.3). The initialization algorithm is described in the following section. Its output is a list of addresses for every device BAR and every bridge, which can be directly programmed into the corresponding registers by the driver. For example:

```
buselement(device, addr(6,0,0), 0, 0xC0000000, 0xD0000000,
         0x10000000, mem, prefetchable, pcie, 64),
buselement(bridge, addr(0,15,0), secondary(6), 0xB0100000,
         0xD0000000, 0x1FF00000, mem, prefetchable, pcie, 0).
```

In this example, the 64-bit PCIe device at bus 6, device 0, function 0 requests a physical address range of 256MB in prefetchable memory space for BAR 0. The base allocated to the device is 0xC0000000 and the limit will thus be 0xD0000000. The bridge at which the device is attached has a base of 0xB0100000 and a limit of 0xD0000000 in the prefetchable memory space, clearly including this device (along with others, not shown here).

In step 4, the PCIe driver reads the result back from the SKB. It takes the addresses and BAR numbers as well as bridge base and limit values from the output, and programs the specified registers (step 5). While reprogramming devices and bridges, they are disabled in order to prevent transient address conflicts.

Once reprogramming is complete, the bus is fully configured and device drivers can be started. Additionally, the allocation result is stored in the SKB for later use. Whenever a device driver for a specific device gets started, it needs to know the base addresses assigned to the BARs of this device. This can easily be queried from the SKB. Hotplugging (see Section 6.3.4) is another reason to store the result for later use in incremental allocation of new devices.

6.3.2 Formulation in CLP

The description below shows how to turn the configuration algorithm into constraint logic. The rules describe how to allocate prefetchable and non-prefetchable memory regions. The allocation of I/O space proceeds the same way. The only difference is the alignment requirement of I/O bridge windows, which gets passed to the code by a parameter.

Following the description in section 3.8.3, the first step is to convert the facts generated by the PCIe driver to a suitable data structure, and declare the necessary constraint variables. The data structure used is a tree mirroring the hardware topology, whose inner nodes correspond to bridges, and leaf nodes to device BARs or other unpopulated bridges. The constraints are then naturally expressible through a recursive tree traversal. The variables of the CLP program are the base address, limit and size of every bridge and device BAR, and the relationship between them may be expressed by the constraint `Limit $= Base + Size`, which the algorithm later applies. At a high-level, the allocation algorithm performs the following steps for each PCIe root bridge:

1. Convert `bridge` and `device` facts for the given root bridge to a list of `buselement` terms, while declaring constraint variables for the base address, limit and size of each element.

2. Construct a tree of `buselement` terms, mirroring the PCIe tree.

3. Recursively walk the tree, constraining the base, limit and size variables according to the PCIe configuration rules and quirks.

4. Convert the tree back to a list of elements.

5. Invoke the ECL^{i}PS^{e} constraint solver to compute a solution for all base, limit and size variables satisfying the constraints.

The core logic of the algorithm resides in step 3 above. The implementation is a direct translation of the rules described in section 6.2.2 to constraint logic, as described in the following sections.

Bridge windows

Rule **1** states that all bridge windows must include all address regions decoded by devices and bridges attached to the secondary bus. This means that the bridge's memory and I/O base addresses must be smaller or equal to the smallest base of any bridge or device on the secondary bus, and the corresponding limits must be greater than or equal to the highest address used by any device or bridge on the secondary bus.

Although at this stage there are not yet concrete values for the relevant base and limit variables, CLP allows constraining them using a recursive walk of the tree, implemented as shown below.
Note that a tree is expressed as `t(Root,Children)`, where `Root` is the root node, and `Children` is a (possibly empty) list of child trees – ECL^{i}PS^e uses conventional Prolog syntax whereby identifiers starting with an uppercase character (e.g. `Node`) denote free variables, and all others denote constants. Also note the ECL^{i}PS^e operations `ic_global:sumlist`, `ic:minlist`, and `ic:maxlist`, which operate on lists of constraint variables that may not have a concrete value assigned, allow complex constraints to be introduced between them.

```
setrange(Tree,SubTreeSize,SubTreeMin,SubTreeMax) :-
  % match Tree into current node and list of children
  t(Node,Children) = Tree,
  % match node to get its base, limit and size variables
  buselement(_,_,_,Base,Limit,Size,_,_,_,_) = Node,

  % recursively collect lists of sizes, minimum and
  % maximum addresses for children of this node
  ( foreach(El,Children),
    foreach(Sz,SizeList),
    foreach(Mi,MinList),
    foreach(Ma,MaxList)
    do
      setrange(El,Sz,Mi,Ma)
  ),
```

```
  % compute sum of children's sizes as SizeSum
  ic_global:sumlist(SizeList,SizeSum),
  % constrain the size of this node >= SizeSum
  Size $>= SizeSum,

  % if there are any children...
  ( not Children=[] ->
      % determine min base and max limit of children
      ic:minlist(MinList,Min),
      ic:maxlist(MaxList,Max),
      % constrain this node's base and limit accordingly
      Min $>= Base,
      Max $=< Limit
    ; true
  ),

  % constrain this node's limit
  Limit $= Base + Size,

  % output values
  SubTreeSize $= Size,
  SubTreeMin $= Base,
  SubTreeMax $= Limit.

setrange([],0,_,_). % base case of recursion
```

Non-overlap of bridges and devices

Rule **2** states that siblings must not overlap at any level of the tree. In other words, all regions allocated to bridges and devices at the same level must be disjunctive. The following goal ensures this by making use of the `disjunctive` constraint, originally intended for task scheduling, which ensures that regions specified as lists of base addresses and sizes do not overlap:

```
% convenience functions / accessors
root(t(R,_),R).
base(buselement(_,_,_,Base,_,_,_,_,_,_),Base).
size(buselement(_,_,_,_,_,Size,_,_,_,_),Size).

nonoverlap(Tree) :-
  % collect direct children of this node in ChildList
  t(_ ,Children) = Tree,
  maplist(root,Children,ChildList),

  % if there are children...
  ( not ChildList=[] ->
      % determine base and size of each child
      maplist(base,ChildList,Bases),
      maplist(size,ChildList,Sizes),

      % constrain the regions they define not to overlap
      disjunctive(Bases,Sizes)
    ; true
  ),

  % recurse on all children
  ( foreach(El, Children) do nonoverlap(El) ).
```

Non-overlap of prefetchable/non-prefetchable memory

Rule **3** requires that prefetchable and non-prefetchable regions do not overlap. The two regions do not need to be contiguous. The implementation inserts an artificial level in the top of the tree containing two separate bridges, one with all prefetchable memory ranges and another with all non-prefetchable memory ranges of the tree. This gives some freedom to the solver, because the order of the two regions is not explicitly specified by the allocation code, and allows the previously-described logic to operate independently of memory prefetchability. Treating the two regions as completely separate trees causes the prefetchable and non-prefetchable window of every bridge to be at completely different locations, which is

fine. I/O regions do not need to be considered here, because the I/O space is physically different form the memory space and cannot overlap with it.

Alignment constraints

Rules **4, 5** and **6** require a specific alignment for devices and bridges. The following rule constrains the alignment of each element, using natural alignment for device BARs, and a fixed alignment for bridge windows (1MB in the case of memory regions and 4kB in the case of I/O).

```
naturally_aligned(Tree, BridgeAlignment, LMem, HMem) :-
  t(Node,Children) = Tree,

  % determine required alignment for bridge or device BAR
  ( buselement(device,_,_,Base,_,Size,_,_,_,_) = Node ->
      Alignment is Size; % natural alignment
    buselement(bridge,_,_,Base,_,_,_,_,_,_) = Node ->
      Alignment is BridgeAlignment % from argument
  ),

  % constrain Base mod Alignment = 0
  suspend(mod(Base, Alignment, 0), 0, Base->inst),

  % recurse on children
  ( foreach(El, Children),
      param(BridgeAlignment), param(LMem), param(HMem)
    do naturally_aligned(El, BridgeAlignment, LMem, HMem)
  ).
```

Reserved regions

Rule **7** requires that reserved memory regions are not allocated to PCIe devices. In other words, memory regions allocated to PCIe devices should always be disjunctive with any reserved region. The following goal ensures this requirement, by recursively processing a list of bus elements against a list of reserved memory ranges, specified as `range(Base,Size)` terms:

```
% recursive stopping case
not_overlap_mem_ranges([], _).

% bridges may overlap:  no special treatment
not_overlap_mem_ranges(
  [buselement(bridge,_,_,_,_,_,_,_,_)|T], MemRanges) :-
    not_overlap_mem_ranges(T, MemRanges).

% device BARs match this pattern
not_overlap_mem_ranges([H|T], MemRanges) :-
  % for each reserved memory range...
  ( foreach(range(RBase,RSize),MemRanges), param(H)
    do
      % match base and size variable from bus element
      buselement(device,_,_,Base,_,Size,_,_,_,_) = H,
      % constrain this BAR not to overlap with it
      disjunctive([Base,RBase], [Size,RSize])
  ),
  % recurse on list tail
  not_overlap_mem_ranges(T, MemRanges).
```

Fixed-location devices

The allocation algorithm must also avoid moving various initialized boot devices, as formulated in rule **8**. The following goal shows one such example: given a device class (specified by its class, subclass and programming interface identifiers) that should not be moved, it constrains the possible choice of the base address to the one value which is its initial allocation.

```
keep_orig_addr([], _, _, _).
keep_orig_addr([H|T], Class, SubClass, ProgIf) :-
  ( % if this is a device BAR...
    buselement(device,Addr,BAR,Base,_,_,_,_,_,_) = H,
    % and its device is in the required class...
    device(_,Addr,_,_,Class, SubClass, ProgIf,_),
    % lookup the original base address of the BAR
    bar(Addr,BAR,OrigBase,_,_,_,_) ->
      % constrain the Base to equal its original value
      Base $= OrigBase
    ; true
  ),
  % recurse on remaining devices
  keep_orig_addr(T, Class, SubClass, ProgIf).
```

6.3.3 Quirks

Declarative logic programming provides an elegant solution to the problem of quirks. Quirks require the allocation algorithm to correct wrong information as well as apply possible extra constraints to workaround misbehaving devices. In CLP it is easy to define a database of facts for devices needing special treatment. Those facts are implicitly matched against the data structure before the configuration algorithm runs, causing incorrect information to be corrected, and additional constraints on the allocation to be defined, without changing any of the core logic of the algorithm.

Rule **11** requires that non-PCIe devices appearing as a BAR of a regular PCIe device or bridge are treated like PCIe devices with a fixed address requirement. As an example, on some machines, an IOAPIC appears as a

BAR of a PCIe device. If this is the case, the IOAPIC decodes the base address assigned to the BAR rather than directly using one of the valid predefined base addresses for IOAPICs. In this case, the allocation code cannot move the BAR, even if the IOAPIC is not a PCIe device. This conflicts with the core logic of the algorithm, which avoids using all regions assigned to IOAPICs. In order to handle this quirk, a slight modification of the core logic of the algorithm is necessary such that it only avoids using address regions assigned to IOAPICs if they do not appear as a BAR. Additionally the algorithm has to apply the following extra constraint, which ensures that IOAPICs appearing as a BAR keep their original base address by calling `keep_orig_addr` on the specific bus, device and function number of the device on which the IOAPIC claims to be.

```
keep_ioapic_bars(_, []).
keep_ioapic_bars(Buselements, [H|IOAPICList]) :-
  ( % if there is a BAR with the same base as an IOAPIC,
    % then do not move it
    range(B, _) = H,
    bar(addr(Bus, Dev, Fun), _, OrigBase, _, _, _, _),
    OrigBase =:= B ->
      keep_orig_addr(Buselements, _, _, _, Bus, Dev, Fun);
    true
  ),
  keep_ioapic_bars(Buselements, IOAPICList).
```

6.3.4 Device hotplug

In principle, the allocation of resources for hotplugged devices can be handled simply by adding facts for the new device and its BARs, and then re-running the allocation algorithm. However, this may cause all existing address assignments to change (excluding those whose location is fixed, as described in section 6.3.2), and is thus undesirable due to the performance impact of interrupting running device drivers. A more incremental approach is desirable.

With PCI Express it is possible to query bridges for hotplug capabilities (i.e., whether or not they have slots to hotplug a device)[25]. To avoid

moving as many devices and bridges as possible, the initial configuration should leave as many gaps as possible under bridges with hotplug capabilities. This could be implemented as an optimization function that maximizes the free space under hotplug-capable bridges. However, an optimization function considers all possible solutions and takes the one which maximizes the free space. This would lead to a complete tree permutation and is therefore too complex and not feasible in a reasonable time.

A more tractable way of creating gaps under hotplug-capable bridges is adding artificial devices under those bridges while computing the first allocation. Artificial devices have regular `device()` and `bar()` entries with the `vendor` identifier set to `0xffff` to mark the devices as artificial. There will never be a device with this vendor identifier, since `0xffff` means at the register level that no device exists at this bus, device and function number. The `bus` part of the device address is set to the secondary bus number of the bridge with hotplug capabilities. This ensures that the artificial device belongs to this bridge. The device number has to be unique under every bus, but can otherwise be an arbitrary number, which does not yet exist on the bus, for artificial devices. Since it is not known in advance, whether the device will have BARs in the prefetchable, non-prefetchable or I/O space, it is necessary to create one BAR in each of the three spaces. The following example shows an artificial device under a hotplug-capable bridge:

```
% the bridge with a hotplug-capable slot under it
bridge(pcie, addr(3, 0, 0), 0x1033, 0x125, 6, 4, 0,
       secondary(4)).

% artificial device with vendor set to 0xffff and all
% other fields to 0
device(pcie, addr(4, 3, 0), 0xffff, 0, 0, 0, 0, 0).

% three small BARs, one in each space
bar(addr(4, 3, 0), 0, 0, 8192, mem, prefetchable, 64).
bar(addr(4, 3, 0), 0, 0, 8192, mem, non-prefetchable, 32).
bar(addr(4, 3, 0), 0, 0, 256, io, non-prefetchable, 32).
```

The space occupied by artificial devices can later be used for real devices

hotplugged under a bridge. When a device is hotplugged it is straightforward to check whether there is enough free space available under the bridge. If this is the case, resources can directly be allocated. The allocation under this bridge needs to follow the same rules as the first allocation, for example, the address has to meet the alignment requirement of the newly hotplugged device. Nevertheless, as long as the gap is large enough a simplified, incremental algorithm for local resource allocation can apply the constraints to the newly hotplugged device.

However, since the physical address size requirements of hotplugged devices are not known in advance, it may still be the case that there is insufficient free address space under a bridge. In this case the allocation algorithm tries to extend the local search by moving the bridge, and in the worst case, a recomputation of the complete allocation is necessary. Similarly, it is not known in advance whether a newly hotplugged device will have special requirements such as a fixed address assignment or other hardware quirks. In these cases a complete reallocation may be necessary.

Adding artificial devices to the PCIe tree before computing the first allocation can be handled well by the allocation algorithm and it is computationally less complex than an optimization problem. Figure 6.4 shows that the CLP solution can deal with an almost completely filled physical address region. This means that the available space can almost be filled completely with artificial devices to provide space for later hotplugs. When creating artificial devices, the first step is to compute the sum of the address size requirements of the real devices and to fill the available address regions for PCIe with small artificial devices almost completely. With CLP this is particularly easy, because the artificial devices are placed around the real ones. Moreover, the CLP solution is well-placed to handle complex reconfigurations that may be required by device hotplug, as it specifies the complete set of feasible configurations which will be explored by the solver. Section 6.5.5 presents the results of a benchmark showing the theoretical limits of the CLP approach in handling device hotplugs, in comparison to a traditional postorder traversal.

6.4 Interrupt allocation

This section now describes the closely-related problem of interrupt allocation, which is also implemented in CLP and also evaluated in section 6.5.

6.4.1 Problem overview

Interrupts are another important resource that must to be allocated to devices by the OS. Most PCI devices can raise one or more interrupts. To avoid shared interrupt handlers, the OS should try to allocate unique interrupt vectors to every device. Modern systems, and some modern devices, support message signaled interrupts (MSIs). These map interrupts into the physical address space, and therefore the only requirement is choosing a different interrupt address for every device. However many systems and many PCI devices do not yet support MSIs, and thus, correctly and efficiently configuring PCI interrupt allocation remains a critical OS task.

Each PCI device signals interrupts by asserting one of up to four available interrupt lines (INTA, INTB, INTC and INTD, represented in the CLP code as the integers 0–3). On PC-based platforms, these signals are routed via PCI bridges and configurable *link devices* to global system interrupt numbers (GSIs). This routing is encoded in and configured via platform firmware, using a set of ACPI tables and functions[59]. Starting from a given device and interrupt pin, the mapping is determined as follows:

1. Consult the ACPI interrupt routing tables for the current bus, device and pin number. If there is a mapping for the given pin:
   (a) If the entry names a GSI, the interrupt line is fixed.
   (b) Otherwise, the entry names a link device, and the interrupt is selectable from set of GSIs.
2. Otherwise, compute the new interrupt pin on the parent bus, using the formula (*device number* + *pin*) mod 4, and repeat.

The goal of the interrupt allocation code is to assign unique interrupt vectors to every device. Interrupt sharing is to be avoided wherever possible[88].

It can severely impact performance, since the drivers for devices that share an interrupt must essentially poll their devices to determine if the interrupt is for them. Furthermore, many device drivers do not handle shared interrupts correctly at all. As well as avoiding sharing among PCI devices, specific GSIs are also assigned to legacy (non-PCI) devices and other system devices This should also be avoided by the allocation code. The summary of the requirements for the interrupt configuration problem is as follows:

1. Assign and configure a GSI (possibly translated by bridges and link devices) for every enabled PCI device,

2. Ensure that all allocated GSIs are unique.

3. Avoid reassigning legacy preallocated GSIs.

This problem is not as complex as PCIe address allocation, and therefore less troublesome to implement in C. However, there are still some benefits from using CLP: storing and querying information about possible GSIs and prototyping the algorithm in CLP is highly convenient, the resulting algorithm is portable across different platforms, and the implementation is concise – ensuring that allocated GSIs are globally unique can easily be done using the built-in ECL^{i}PS^{e} goal `alldifferent` (see 6.4.2). These are good reasons to implement interrupt allocation for Barrelfish in the SKB.

6.4.2 Solution in CLP

At start-up, the PCI and ACPI drivers populate the system knowledge base with a fact for every PCI interrupt routing table entry, mapping a device address and interrupt pin to a source, using the schema:

```
prt(addr(Bus, Dev, _), Pin, pir(Pir) | gsi(Gsi)).
```

These facts include addresses of PCI devices without function number, because the same mapping applies for all functions on a multi-function device. The interrupt source is either a name (ACPI object path) identifying the interrupt link device or a direct GSI number, indicating that this interrupt's allocation is fixed.

For each link device, `pir` facts are added describing the possible GSIs that may be selected for a given device:

```
pir(Pir, GSI).
```

In this relation, `Pir` defines the link device name, and `GSI` one of the selectable GSIs for this device (so each link device has multiple facts, one for each configuration).

The CLP code operates on these facts, and the PCI device facts described in the previous section. At the top-level, it determines the interrupt pin used by a specific device, and passes it to `assignirq` to allocate a unique GSI:

```
assigndeviceirq(Addr) :-
  device(_, Addr, _, _, _, _, _, Pin),
  % require a valid Pin
  Pin >= 0 and Pin < 4,
  ( % check for an exising allocation
    assignedGsi(Addr, Pin, Gsi),
    usedGsi(Gsi, Pir)
  ; % otherwise assign a new GSI
    assignirq(Pin, Addr, Pir, Gsi),
    assert(assignedGsi(Addr, Pin, Gsi))
  ),
  printf("%s %d\n", [Pir, Gsi]).
```

`assignirq` takes the PCI address and interrupt pin for the device as inputs, and chooses a possible GSI for the device. It uses `findgsi` (described below) to determine the available GSIs for the device, and the `alldifferent` goal to avoid overlaps:

```
assignirq(Pin, Addr, Pir, Gsi) :-
  % determine usable GSIs for this device
  findgsi(Pin, Addr, Gsi, Pir),
  ( % flag value for a fixed GSI (i.e.  meaningless Pir)
    Pir = fixedGsi
  ;
    % don't change a previously-configured link device
    setPir(Pir, _) -> setPir(Pir, Gsi)
  ;
    true
  ),
  % find all GSIs currently in use
  findall(X, usedGsi(X,_), AllGsis),
  % constrain GSIs not to overlap
  ic:alldifferent([Gsi|AllGsis]),
  % allocate one of the possible GSIs
  indomain(Gsi),
  % store settings for future reference
  ( Pir = fixedGsi ; assert(setPir(Pir,Gsi)) ),
  assert(usedGsi(Gsi,Pir)).
```

Finally, the following CLP function matches the device's address and interrupt pin with the `prt` and `pir` facts to find the possible GSIs (multiple solutions may be found). If no match is found, it recursively performs bridge swizzling until a routing table entry matches (which is always true at the root bridge).

```
findgsi(Pin, Addr, Gsi, Pir) :-
  ( % lookup routing table to see if we have an entry
    prt(Addr, Pin, PrtEntry)
  ;
    % if not, compute standard swizzle through bridge
    Addr = addr(Bus, Device, _),
    NewPin is (Device + Pin) mod 4,

    % recurse, looking up mapping for the bridge itself
    bridge(_, BridgeAddr, _, _, _, _, _, secondary(Bus)),
    findgsi(NewPin, BridgeAddr, Gsi, Pir)
  ),
  ( % is this a fixed GSI, or a link device?
    PrtEntry = gsi(Gsi),
    Pir = fixedGsi
  ;
    PrtEntry = pir(Pir),
    pir(Pir, Gsi)
  ).
```

6.5 Evaluation

The evaluation of the PCIe allocation algorithm is mainly in terms of code complexity and efficiency of resultant solutions. Execution time is also an important metric, but not the main point of this approach. Obviously, some of the evaluation necessarily remains subjective in its comparison with current approaches, not least because code in this approach is factored rather differently from traditional approaches and offers different functionality to, for example, PC-based Linux.

	Devices	BARs	Bridges	Runtime (ms)
sys1	7	11	12	2.0
sys2	13	20	6	14.7
sys3	13	20	6	14.4
sys4	14	22	6	36.4
sys5	12	18	5	10.0
sys6	7	9	6	19.0
sys7	9	14	6	22.2
sys8	15	25	4	6.7
sys9	15	25	4	31.2

Table 6.2: System complexity and execution times for the PCI configuration algorithm

6.5.1 Test platforms

I evaluated the PCIe configuration and interrupt allocation algorithms on nine different x86 PC and server systems, with a mixture of built-in and expansion devices including network, storage and graphics cards installed. I refer to these as sys1 through sys9, and show the number of PCIe elements they include in Table 6.2. All systems have two PCIe root bridges with the exception of sys1, which has one. Here I show the totals for the whole system, as the algorithm allocates resources to all PCIe trees in a single invocation.

All of these systems support USB keyboards in the BIOS, and thus the system initializes the USB controller in firmware at boot time. Consequently, the allocation algorithm ensures this fixed device requirement using the `keep_orig_addr` constraint from section 6.3.2 to prevent the USB controllers from being reprogrammed, and also avoid any memory regions marked as reserved by ACPI or in use by IOAPIC devices. The computation does not include handling other quirks, since the test platforms do not exhibit them and consequently do not exercise that part of the CLP code. The implementation is successful in configuring all PCI buses and devices on all the test systems.

6.5.2 Performance

I measured the time for PCI configuration on the test platforms, and show the results in Table 6.2. This time is for the CLP algorithm and does not include the initial bus walk, nor programming of device registers. As discussed in section 6.3, these remain in C as part of the PCIe driver, and the CLP time dominates the overall runtime.

Compared to the performance of a hard-coded allocation in C, which in existing OSs typically requires less than a millisecond, the CLP solution is substantially slower, but the additional overhead of 10–30ms is only incurred at boot time or after a hotplug event, and so is arguably insignificant to the end user. This computation can be run in parallel with other tasks, and since the PCIe configuration changes rarely, the computed configuration can be cached and re-applied during the next boot process. In these cases, no additional overhead is added to the boot time.

6.5.3 Code size

This section compares the complexity, measured in lines of code (LOC), of the CLP-based approach to the comparable portions of the Linux x86 PCI driver. Such a comparison can never be precise, and must be preceded by several qualifications.

First, in both cases I consider the code related to PCI resource configuration, interrupt allocation, PCI device discovery, maintenance of the data structures representing the PCI bus hierarchy, and the corresponding hardware access mechanisms. Second, I exclude from the Linux figures some PCI-related mechanisms (such as the legacy PCI BIOS interface) that are currently unsupported by the CLP solution. Third, since the PCIe driver in Barrelfish currently only implements two PCI quirks, the hardware quirk-handling code is excluded, but handling of other special cases is retained. Fourth, the functionality offered by this solution and the Linux code is different: Linux implements the solution that attempts to fix up the initial BIOS configuration, whereas the CLP code does a full allocation of addresses. Finally, to emphasize it, the goal is to reduce the complexity of the source code and therefore the number of source lines of code, rather

	C LOC	CLP LOC
Register access	235	
Data structure	817	31
Algorithm		224
ACPI	360	
Interrupts	660	28
Miscellaneous	109	
Total	2181	283

Table 6.3: Lines of code in PCI configuration and interrupt allocation

than the number of generated machine statements.

Table 6.3 summarizes the results for the CLP-based solution and table 6.4 summarizes them for Linux. The relevant Linux code is located in the kernel in drivers/pci. Overall, this approach uses 2464 lines of code, compared to 5210 for the pure C-based Linux version.

Breaking this down, the PCIe driver in Barrelfish uses much less code for reading and writing registers, as the hardware access is regular and independent of allocation. Building and manipulating data structures is also simpler: representing lists and trees is highly concise in $\text{ECL}^{\text{i}}\text{PS}^{\text{e}}$, and allows building much simpler structures in the C domain, resulting in about half the code size. The number of lines of code for ACPI is higher in Barrelfish, since it explicitly handles ACPI reserved regions, whereas Linux relies on the BIOS initialization for this. Code for interrupt assignment is about the same size. Finally, the “core” of the configuration code (in as much as it can be isolated in the Linux case) is 224 lines of Prolog versus 1243 lines of C.

The largest class of code in both implementations is used for maintaining data structures. This is because PCI data must be queried from either ACPI or directly from the hardware, transformed to a meaningful internal representation, and added to a structure. Finally, the configuration proceeds by traversing this structure, accessing and mutating it. The corresponding data structure in the CLP implementation consists mostly of

	C LOC
Register access	842
Data structure	2079
Resource management	1243
ACPI	238
Interrupts	718
Miscellaneous	90
Total	5210

Table 6.4: Lines of code for equivalent functionality in Linux 3.1.6

ECL^{i}PS^e facts which are generated by C but traversed/accessed entirely in CLP, and thus require fewer lines of code than Linux. Despite being large in size in both systems, such code is not the most complex in its logic.

The PCI configuration algorithm uses 224 lines of CLP code in Barrelfish's PCIe driver implementation. This produces a correct and complete allocation, while correctly handling special constraints such as avoiding reserved regions and preserving certain device locations. In comparison, the Linux C implementation uses more lines of code for less functionality (it does not perform a full bus configuration).

Besides the usual benefits arising from a smaller, simpler codebase in terms of source lines of code, the separation of concerns between low-level hardware-specific device access code and a high-level declarative resource configuration algorithm enhances the system's maintainability and adaptability to changing hardware requirements. Complex device- and system-specific constraints, such as quirks, can be incorporated without changing the device access code or core algorithm, and it can easily be ported to other PCI-based platforms.

6.5.4 Handling quirks

An important goal of using a declarative algorithm is the maintainability of the code as well as simplifying of adding new special cases or quirks.

Special case	Goal	Part	CLP LOC	C LOC
No re-assignment	`keep_orig_addr()`	impl.	-	-
		call	1	-
IOAPIC as BAR	`keep_ioapic_bars()`	impl.	10	-
		call	1	-
		get IOAPIC list	3	-
Total			15	-

Table 6.5: Additional lines of code to handle additional special cases

These properties can best be evaluated by showing the number of lines of code which must change when adding a new special case.

To take one example, consider a new PCIe device that does not support the re-assignment of a new address. The implementation already contains the goal `keep_orig_addr()`, which ensures that the device retains its original address and therefore no re-assignment will happen. It is sufficient to call this goal on the newly-found device, and this only requires one additional line of CLP code to specify the case.

A second example was encountered in the course of writing the PCIe driver, and has already been mentioned in section 6.3.3. One system has a special IOAPIC that appears as a BAR, even though it is not a PCI device. In this case, the address in the BAR must not change during the configuration process. The implementation did not contain any goal to handle this special case, and so I had to implement it from scratch. The small goal `keep_ioapic_bars()` shown in section 6.3.3 completely handles this case, making use of the already available `keep_orig_addr()`. The implementation only adds ten lines of CLP code. An additional line is necessary to call the goal and another three lines are necessary to prepare the list of IOAPICs.

Table 6.5 summarizes the additional lines of code necessary to handle these two additional special cases. In CLP handling them is straightforward, because the base variables can be constrained before having actual addresses assigned. As the table shows, the C code did not need to be changed at all.

6.5.5 Postorder traversal comparison

To evaluate the quality of the solutions found, I investigated how they compare to the style of simple postorder traversal used in current operating systems. When allocating resources to a device tree where the size of each device is known in advance, one might expect this approach to be sufficient. I first describe why that is not the case, and then show, experimentally, the advantage of a declarative CLP solution against such a traversal.

Starting with the base address given by the root bridge, such an algorithm traverses down the left-most branch of the tree first, assigning the current base address to each bridge and finally the left-most leaf device, while satisfying alignment constraints. For each device allocation, the device size is added to the base value, plus any padding required for alignment. The algorithm next traverses all child devices of the bridge, before moving up the tree to the next-upper parent bridge, and updating the bridge's limit register in the process, before continuing with the remaining devices and bridges.

Such an algorithm can be simply described and implemented. It ensures that all bridges are allocated a window, which includes their children, and that alignment constraints are satisfied. However, the algorithm is insufficient for PCIe configuration for two reasons:

1. It fails to include constraints that require keeping devices at a fixed address. This requires all parent bridges to decode the fixed device window. Because all parent bridges have to decode a fixed address, all children of every bridge decoding a fixed address have to be placed close to a predetermined address region. This cannot be easily expressed in a postorder traversal of the device tree.

2. Satisfying alignment constraints leads to potentially large amounts of address space wasted in padding, thus preventing a successful configuration when not all devices fit into the root bridge's address range.

To learn how the CLP-based algorithm behaves at extreme cases, where

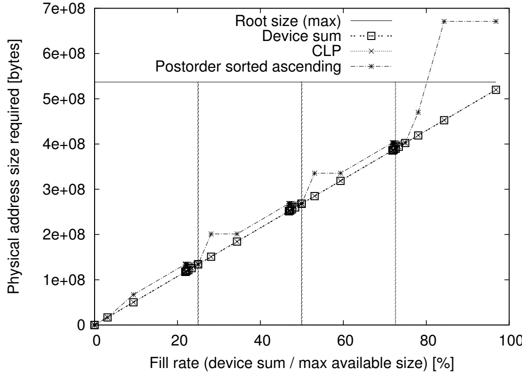


Figure 6.4: Address space utilization of CLP algorithm vs. simple postorder traversal as devices and bridges are added to a simulated system. The CLP algorithm reorders devices as needed, exactly following the *DeviceSum* line, which shows the lower bound. The postorder traversal, which sorts the devices according to size, cannot fit the PCI tree into the given root bridge window. Vertical lines indicate when a new bridge is added; the horizontal line indicates the maximum size of the root bridge window.

lots of resources are consumed by additional devices, I stressed the configuration algorithm in an offline experiment by adding progressively more devices and bridges to a simulated PCIe system. Starting with zero devices and bridges, I added either a device or a bridge on every round and measured the consumed resources by the configuration derived by the algorithm. This scenario is not purely artificial, because it simulates what can happen when devices are hotplugged. I compared the CLP-based algorithm with an improved postorder traversal algorithm, which sorts devices according to their requested size in ascending order. The results are shown in figure 6.4.

The horizontal line *Root size (max)* indicates the given root bridge window size, which must not be exceeded for a successful configuration. The

vertical lines in the figure indicate where a bridge has been added to the PCIe tree. The *DeviceSum* line indicates the sum of the requested size of all installed devices without padding or alignment constraints; this is the absolute lower bound of address space utilization. The data points indicate the address space consumption after having added the next device.

The figure shows that the CLP-based allocation algorithm exactly follows the device sum. Its constraints give it the freedom to reorder bridges and devices, so that no address space is wasted for alignment constraints and a solution can always be found. The best postorder traversal algorithm, which does not respect fixed device requirements, nevertheless cannot fit the devices into the given root bridge window beyond 80% utilization, indicating that such a simple approach has limitations in general.

6.6 Summary

The case study presented in this chapter proves that applying a high-level declarative language to a hardware configuration problem leads to much simple and cleaner code. Furthermore, it can easily adapt to differences and special cases found in different platforms. By picking PCIe configuration as a case study and by proving the feasibility of modeling this problem in CLP, I believe that many other hardware-related problems can be solved likewise.

PCI address allocation is one of the most complex hardware resource allocation problems currently found in PCs, because multiple devices are configured in a single step, and there are many dependencies between devices and bridges, and constraints on the assignment of addresses to groups of devices under a bridge. One might see it as something of a special case. Historically, however, hardware complexity has tended only to increase, with a concomitant increase in software's responsibility to configure it: PCI arose as a solution to the increasing complexity of device configuration in earlier, simpler ISA and ISA-PnP systems, which it resolved by placing a greater configuration burden on platform firmware and system software.

A similar emerging trend can be observed in the configuration of the

interconnect between cores, caches, memory and devices as it gains increasing complexity. Previous systems, such as the older Intel frontside bus architecture, had static interconnects whose architecture was fixed in hardware. However, current interconnects such as QPI and HyperTransport[31] are configurable multi-hop point-to-point networks. Present systems rely on platform firmware to configure these networks statically at boot time, but one can easily imagine a future where system software may be able to dynamically reconfigure the interconnect according to workload requirements, for which a declarative solution in CLP may be appropriate.

The experience in building a high-level declarative configuration algorithm for the complex PCIe resource allocation problem in the SKB has been positive and it is a promising approach for other hardware configuration tasks. The SKB as a reasoning facility, which is already there, greatly simplifies the decision of implementing configuration and allocation algorithms in CLP.

Chapter 7

Efficient Multicast Messaging

In the introduction (section 1.1.2) I argued that the interconnect, which increasingly resembles a network, has to be treated as such, to provide optimal performance. In this chapter I show how a few lines of CLP code derive a hardware-aware multicast tree along which the messaging mechanism forwards messages with low latency and good scalability. I show, that the forwarding tree, which is used to configure the messaging mechanism, adapts to the interconnect topology by only considering high-level knowledge about the interconnect. The knowledge is derived from information gathered at bootup of the operating system. The code does not assume any information and therefore adapts to any kind of hardware, even if it is not known in advance. The mechanism of sending messages is simple, because the policy code, which derives hardware-aware forwarding tables, is completely taken out of the actual messaging code. Further, the CLP code allows concisely expressing an optimal hardware-aware forwarding algorithm. The evaluation in this chapter shows first, that the code complexity is low, and second, that the result – the actual forwarding of a multicast message – has low latency and good scalability, compared to

non-hardware-aware versions.

7.1 Introduction

Manycore operating systems perform operations which need fast global coordination among cores to ensure consistency. Global coordination involves notifying all participants and waiting for their acknowledgement. In fact, at a high-level, it is the concept of multicast communication. Multicast communication notifies all participating nodes and waits for their reply. An example, which needs global coordination, is keeping page table mappings and their access rights consistent among cores. This needs coordination between all cores, because they map a page individually (this is explained in more detail in the background section 7.2) and they have to agree on the access rights of every page.

To ensure that the globally coordinated operations are not limited by the operating system's low-level coordination implementation[1], the operating system needs a fast mechanism to inform every participant and to get their acknowledgements.

Current x86 hardware provides the illusion of shared memory between all cores. Every core can access every memory address and the interconnect hardware takes care of sending *memory reads* or *memory writes* to the right place. In fact, as explained in section 1.1.2, the interconnect is a network. It translates *memory reads* and *memory writes* on remote NUMA nodes to messages conveying the read or write operation. It sends the message to the remote memory controller on which it gets applied. Additionally, the cache-coherence protocol ensures that the cache contents of all cores is consistent. If one core writes to a memory address, which is in the cache of several other cores, they all get invalidation messages for this address. This causes all other cores to load the data from memory again if they access the same address again.

A naive way of notifying every core about an operation can cause the interconnect hardware to send many messages over the interconnect, while

[1]Here it is low-level coordination on the system's fast-path. It is different from Octopus, which coordinates processes at a high level.

a hardware-aware method can minimize the number of messages being sent. As I show in the evaluation in section 7.5.4, this directly translates to overall latency and scalability. It is therefore crucial that the operating system reasons about the interconnect topology and that it adapts its global coordination mechanism to it.

The two extreme, but simple, ways of sending a request to several cores are the following: either use one known memory address to write the request to or use a different memory address for all participating cores and write the same requests to each of them. In the first case, all participating cores poll the same memory address. Whenever the requesting core writes to the memory address, all cores get an invalidation message from the cache-coherency protocol and then read the data item again. For N cores, the data traverses the interconnect N times, scaling linearly with the number of cores. The result is similar to sequential processing. In the second case, the requesting core sequentially writes the same request to N different memory addresses. Again, it scales linearly with the number of cores.

In this thesis I argue that a certain degree of parallelism hides the overall latency and leads to faster and more scalable distributed operations. Hiding latency behind parallelism is a common technique used in CPUs and GPUs by pipelining and hardware level threading[19, 42, 43]. In the context of notifying other cores it means that, after the initiating core has sent the request, other cores process and possibly forward the message in parallel. This leads to some sort of message forwarding tree.

The operating system needs to decide how the forwarding tree should look like. Ideally, it reduces the number of messages being sent over the interconnect. The operating system therefore needs to reason about the interconnect and needs to derive a hardware-aware forwarding tree. In this chapter I show that a few lines of CLP derive a hardware-aware forwarding tree which leads to low overall multicast latency and good scalability.

7.2 Background

This section gives the necessary background about multicast messaging and explains why directly applying algorithms from the distributed field does not work within a shared-memory machine. To measure the performance of the derived multicast tree, the *TLB shootdown* operation is used as an example of a real operating system operation, which needs multicast communication. This section explains what the thesis means by the TLB shootdown operation.

7.2.1 Multicast messaging

Traditionally, multicast communication is used in networks. Multicast trees forward messages along the tree and allow parallel processing and forwarding towards the multicast group. Multicast trees are well studied in different scenarios in the distributed field[66, 136, 137, 140]. A multicast tree uses point-to-point links between nodes. Nodes work in parallel which reduces the overall latency. The same technique seems appealing in an operating system, because it allows cores to work in parallel. As stated in the introduction above (section 7.1), the two extremes (one memory address for all the cores or a separate memory address per core) lead to sequential processing, which is not desirable. Instead, I argue in this thesis that a multicast tree with some degree of parallelism between the two extremes leads to low overall latency and good scalability.

Barrelfish has the right structure to explore multicast messaging techniques (see section 2.2) for two reasons. First, it uses explicit messaging to communicate between cores, instead of shared memory. It knows exactly when and from which source and to which destination messages traverse the interconnect. By explicitly knowing that, the problem of sending data N times over the interconnect for N cores can be avoided. Second, Barrelfish provides messaging mechanisms which allow setting-up message channels between any two cores. The policy code just needs to decide from where to where to create message channels to forward the multicast message. According to the result of the policy code, message channels are created and multicast messages are sent along them. The challenge is to

derive a suitable multicast tree.

Within a shared-memory system, the multicast tree might be a regular n-ary tree or an irregular tree of some form. Unlike in distributed systems, the interconnect between cores does not restrict the tree construction, because of the shared memory between all cores. There is a lot of freedom to chose multicast messaging trees, all having a different overall latencies. A suitable multicast tree therefore depends on knowledge about single-link latencies, but, as the next paragraphs explain, also on further hardware knowledge.

Dijkstra's algorithm, for example, constructs a minimal spanning tree according to single-link latencies. Within the machine, the operating system has a global view over the interconnect network, a requirement for Dijkstra's algorithm. Dijkstra's algorithm optimizes for the shortest paths from a root node to all other nodes. The fact that the shared memory architecture basically provides a full mesh network, means that using the direct link is always shorter than going over another node and adding the two latencies[2]. This means that Dijkstra's algorithm would create a link from the root node to all other nodes, leading to one of the two extreme cases described in section 7.1, depending on the implementation of the mechanism. Two options are possible: the root node uses one channel to send a broadcast message to every other core or the root node uses a separate channel to every other core and sends individual messages. As explained in the introduction section 7.1, both versions lead basically to sequential processing.

To avoid this problem, another approach is necessary to construct a multicast tree. In addition to considering only network characteristics like single-link latencies, the multicast tree construction has to exploit hardware topology knowledge as well. To increase parallelism, while keeping interconnect traffic low, the policy code has to choose inner nodes through which messages have to go, such that forwarding messages from inner nodes to children reduces interconnect traffic.

The optimization algorithm depends therefore on two types of knowl-

[2] This is because the latencies on all single links are similar and adding two links is always longer than the longest single-link latency.

edge, both of which are discovered at boot time and stored as platform-independent high-level facts in the SKB. First, it needs knowledge about single-link latencies. These depend on the hardware, but can be measured easily at the start-up of the operating system. Second, it needs hardware topology information to decide on inner nodes. This information can be discovered at boot-up as well. The actual algorithm depends on the high-level latency and topology facts and remains portable and – implementation-wise – machine-independent. Section 7.3.2 describes the implementation of the multicast tree construction in detail.

7.2.2 TLB shootdown

Page table mappings are cached in the translation lookaside buffer (TLB). Every core has a separate TLB. The TLB is therefore a distributed data structure, even if the operating system's kernel is monolithic. On every memory access, the core considers the page table mapping stored in its TLB to translate the virtual address to a physical one. Additionally, it checks the page's permissions to decide, whether the memory access instruction is allowed to proceed. If one core changes the page mapping or the page's access rights, it needs to notify all other cores first. These remove the cached entry from their TLB and acknowledge the request. The initiating core needs to wait for their replies. If one of the other cores accesses the same page again, it is not in its TLB and therefore it loads the page table entry with the new rights from memory into its TLB. This way consistency among cores is ensured. For this thesis I refer to the process of notifying other cores about changed page table mapping by the term *TLB shootdown*.

The TLB shootdown operation is one of the simplest, yet important operations of the operating system. Invalidating a single TLB is a fast operation, taking about 95 to 320 cycles on current x86_64 machines. The complete TLB shootdown operation is a latency-critical operation, because the initiator needs to wait for all other cores before it can complete the operation locally.

Every multicore operating system needs a TLB shootdown operation. Windows and Linux use a known location to store the page manipula-

tion operation and inter-processor interrupts (IPIs) to notify all other cores about the newly arrived operation. A core which changes the mapping of a page writes the operation to the known location and sends an interrupt to every core which might have a mapping in its TLB. Every core takes the interrupt. It invalidates its TLB and acknowledges the interrupt by writing to a shared variable. Finally, the core resumes to user space. While this has low latency, it can be disruptive. The cost of taking an interrupt is about 800 cycles.

The TLB shootdown operation is an example that shows the necessity for efficient multicast communication. Therefore, the thesis uses this example to evaluate the efficiency and scalability of multicast communication trees that are derived at runtime on a wide range of hardware, which is not known in advance. The TLB shootdown provides a baseline protocol. It is simple in the sense that participating CPU drivers always acknowledge it. More involved protocols are general two-phase commit protocols. They also need multicast messaging, but participating cores might "abort" an operation. The focus of this thesis is building hardware-aware multicast trees and therefore does not evaluate different protocols that build on the basic multicast mechanism.

7.2.3 Summary

This section pointed out the most important reasons for multicast messaging within an operating system. By now it should be clear that a suitable multicast tree needs latency measurements as well as hardware topology knowledge and therefore regular tree construction algorithms from the distributed field cannot be applied directly.

The chapter uses the TLB shootdown operation as an example, because it needs global coordination, which can be achieved by multicast communication. Furthermore, it is a common operation in multicore operating systems.

The next section describes how to build a hardware-topology-aware multicast messaging tree in an operating system, such that it adapts to the current underlying hardware and such that it also leads to optimal performance when sending multicast messages.

7.3 Design

This section describes the design of the multicast tree construction algorithm used to achieve a hardware-topology-aware tree. Before explaining the actual algorithm in section 7.3.2, I explain the most important design principles in the next section 7.3.1.

7.3.1 Design principles

I derived a number of properties which should be met by the multicast tree on every possible underlying hardware and summarizes them in the paragraphs below.

Policy/mechanism separation is one main goal of this thesis. This is especially important for multicast messaging, because the mechanism of sending multicast messages is on the system's critical fast-path. The multicast mechanism code uses a forwarding table which it just looks up in order to send the next message. The entries in the forwarding tables define the next core ID to which a message should be sent to. The policy code in the SKB derives the values to be inserted in the routing tables of all cores. If the configuration changes (changes in the environment for some reasons, CPU hotplug, CPU power-save mode), and the multicast tree needs to be recomputed, the policy code can be invoked again and the results can be applied to the forwarding tables. While the SKB derives the new tree, the current entries in the forwarding table remain valid.

Adaptability to hardware is ensured by looking at hardware discovery data in the SKB and by deriving the forwarding table entries at runtime. As long as correct hardware information is available, a multicast tree can be derived. The tree construction code is independent of the number of CPU packages, number of cores per package or interconnect topology. It is purely based on facts describing CPU packages and cores and online measured message latencies.

Portability and maintainability are further goals of this thesis. As long as the CPU package description facts can be generated and message latencies can be measured, the multicast algorithm does not need to change at all. The mechanism of sending multicast messages is based on forwarding tables and the underlying message channels. The forwarding tables are independent of the underlying architecture and porting the code does not involve changing the mechanism. The only two parts of he code which potentially need to change are the datagathering mechanism to query CPU packages and the underlying message transportation implementation.

Keeping the code simple improves maintainability and understandability. Still, the multicast policy code should derive a reasonably good multicast tree to allow for fast multicast messaging. By using hardware knowledge and by exploiting hardware characteristics gained from this knowledge, the code produces good multicast trees at a low code complexity in terms of lines of code needed to implement the policy.

7.3.2 Hardware-aware multicast tree

The goal of the multicast tree is to maximize parallelism and minimize interconnect usage. It is the task of the policy code to find the optimal point. This is reached, when more parallelism would increase latency due to congestion on the cache-coherency protocol and more point-to-point messaging would increase latency due to too much sequential processing.

The cache-coherency protocol on x86 machines is often a broadcast protocol. Invalidations of cache lines are typically sent to all cores. However, many modern CPU packages have a shared cache (typically a shared L3 cache) and appear as a single node regarding the cache-coherency protocol. The message passing between cores on the same CPU socket remains local in the shared L3 cache and does not involve interconnect transactions. It is preferable, that one core of the package forwards a multicast message to all other cores on the same package. Only one message should be sent through the interconnect to notify a remote package. This reduces the interconnect transactions and therefore the number of effective broadcast messages imposed by the hardware.

In general, the overall latency cannot be computed statically. To find the optimal multicast tree, all variants would need to be measured (including a varying number of children for each node). This is however not practical at every startup of the OS. Instead, the policy code exploits the fact, that message passing within the same CPU packages relies on a shared L3 cache or otherwise fast local communication between cores. It choses one core per CPU package as the multicast aggregation node. This core is responsible to forward the message to all cores on the same CPU package and also to forward the message to the next CPU packages, if it is not a leaf of the tree. Forwarding messages to the next CPU packages happens along dedicated point-to-point message channels. The order of packages, to which the core forwards the message, is defined by the policy code. The policy code sorts the next level CPU packages by point-to-point latency between each package and the sending core in decreasing order. This ensures, that the core forwards the message first to the remote package with the highest latency. This technique hides the latency and improves parallelism. While the message with the highest latency is still in transit, the core spends time to send the message to the next package. The cost of sending the message plus the smaller latency overlaps with the first message with higher latency. After sending the message to all remote packages, the core forwards it also to all other cores on the same package.

Similarly, within a package, the message is sent to all other cores. Again the latency can be hidden. Local cores start processing the message in parallel while the messages to the remote CPU packages are in transit. Because the locally shared cache lines between the aggregation core and all cores on the same node do not cause invalidation messages on the interconnect, the second message, within the package, is much cheaper. As a further optimization, the algorithm takes the NUMA knowledge into account. It allocates message buffers in the NUMA node of the multicast aggregation core. The evaluation (section 7.5) in this chapter shows that this algorithm performs best on big NUMA architectures. The optimal tree constructed this way is different for every initiating node. Every root node needs to compute its own multicast tree. At startup time, it invokes the policy code in the SKB and passes itself as the root of the tree. The SKB then returns the multicast tree it should use. As long as the hardware does

not change (due to hotplug for example), the tree can remain as initially derived. The algorithm can be summarized into the following steps:

1. Choose root node according to argument passed to policy code
2. Choose one core per CPU package as multicast aggregation core
3. Sort multicast aggregation cores by latency to root core in decreasing order
4. Send message to remote cores
5. Send message to local cores
6. All remote cores send message to local cores
7. Process message
8. Send replies to aggregation node
9. The aggregation node waits for replies and sends itself a reply to the root node
10. Once all replies are received at the root node, it knows that it has been processed by every core

Figure 7.1 shows a simple four packages quadcore machine on which a multicast tree over all cores (in fact a broadcast tree) was constructed. The root node first sends the message to the furthest core (1) and then to one core of the other CPU packages (2 and 3). Only after sending the messages to the remote cores, it forwards the message also to all local cores, first to the one with highest latency (4) and finally to the one with the second highest latency (5) and then to the closest one (6). In parallel, all other inner nodes (on the other CPU packages) forward the message to the local cores, again first to the one with highest latency (4) and finally to the other ones (5 and 6).

Cores send acknowledgements back along the same tree. Inner nodes aggregate all acknowledgements. As soon as all acknowledgements arrive,

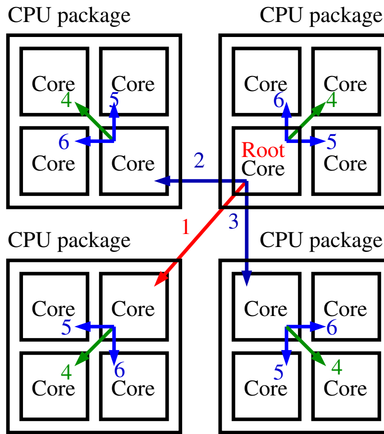


Figure 7.1: Multicast tree on a four CPU packages, quadcore machine.

inner nodes send an acknowledgement back to the root node. The multicast operation is done, as soon as the root node received the acknowledgements of all inner nodes.

This algorithm can be translated directly to CLP code based solely on CPU package knowledge and online latency measurements between the cores. The next section explains how the algorithm is expressed in ECLiPSe.

7.4 Implementation

Following the design presented in the previous section, I implemented the policy code to generate the multicast message tree construction completely in ECLiPSe CLP in the SKB. Below I explain each rule in detail.

Main goal

Each core calls the main goal and passes its own core id as parameter `StartCore` which will be the root node of the multicast tree for this core. `SendList` outputs the list of links to be used to forward the multicast messages initiated at the root node. The list contains tuples of the form `sendto (SrcCore, DstCore, LinkLatency)`.

The main goal choses the tree with the lowest cost. It considers different trees and uses the built-in `minimize/2` goal to chose the tree with the lowest cost.

```
% goal to be called.
% Construct a list of sendto/3 goals, sort them by latency
% in decreasing order and
% minimize the value of the longest latency

multicast_tree(StartCore,SendList) :-
   minimize(multicast_tree_cost(StartCore, SendList, Cost),
            Cost).
```

Tree construction

The following rule creates a list of `sendto/3` tuples and associates costs with each link. The costs are computed as RTT values per link. After sanity checking, the rule creates a list of all available CPU packages. The first step is creating `sendto/3` tuples to all other packages. After that it creates `sendto/3` tuples between the root node of the start package and all cores which are within the same package. Finally, the `sendto/3` tuples get sorted in decreasing order by RTT.

```
% constructs the send list starting at StartCore

multicast_tree_cost(StartCore,[SendH|SendList], Cost) :-
    multicast_sanity_check,
    % determine package of start core
    cpu_thread(StartCore, StartPackage, _, _),
    % construct list of other packages
    findall(X, (cpu_thread(_,X,_,_), X =\= StartPackage), L),
    filter(L,PackageList),
    % compute possible links to those packages as SendList1
    sends(StartCore, PackageList, SendList1),
    % compute links from start core to its neighbors
    sendNeighbors(StartCore, Neighbors),
    append(SendList1, Neighbors, SendList2),
    % annotate with RTT of each link
    annotate_rtt(SendList2, SendList3),
    % sort by decreasing RTT
    sort(3, >=, SendList3, [SendH|SendList]),
    % determine cost as maximum single-link RTT
    sendto(_,_,Cost) = SendH.
```

Links to other packages

This rule creates a sendto/3 tuple from the start core to one core of every available package (except the package of the start core). Starting from the chosen core per package, the rule creates sendto/3 tuples to all the cores within the same package as the chosen core by applying the rule sendNeighbors/2.

```
% creates a list with sendto(SrcCore, DstCore) to define which
    core should send
% to which other core

% sends(+StartCore, +PackageList, -SendsList)
sends(_, [],[]).
sends(StartAPIC_ID, [H|T],[HS|Sends]) :-
   % find the lowest APIC ID on the package as APIC_ID
   findall(X, cpu_thread(X, H, _, _), APICIDs),
   sort(APICIDs, [APIC_ID|_]),
   % construct a link to it from the start ID
   HS = sendto(StartAPIC_ID, APIC_ID),
   % recurse on other packages
   sends(StartAPIC_ID, T, Sends2),
   % find all the cores on the same package as APIC_ID, and
       add pairs for them
   sendNeighbors(APIC_ID, M),
   append(Sends2,M,Sends).
```

Links to neighbor cores

This rule constructs links between cores on the same package. It retrieves all `APIC_ID`s of the same package and creates a `sendto/2` tuple for all of them.

```
% construct links to all the neighbors of APIC_ID on the
% same package as it
sendNeighbors(APIC_ID, Sends) :-
    % find package containing APIC_ID
    cpu_thread(APIC_ID, Package, _, _),
    % construct links to all my neighbors
    findall(sendto(APIC_ID,X),
            (cpu_thread(X,Package,_,_),X =\= APIC_ID),
            Sends).
```

Compute RTT

This helper function computes the round-trip time (RTT) for each link by adding the measured one-way latencies of both directions. Measurements showed that the latencies for both directions on a specific link do not necessarily need to be the same.

```
% add the rtt number to every sendto tuple

annotate_rtt([],[]).
annotate_rtt([sendto(Src,Dst)|T1],[sendto(Src,Dst,Lat)|T2]) :-
    message_rtt(Src,Dst,Lat1,_,_,_),
    message_rtt(Dst,Src,Lat2,_,_,_),
    Lat is Lat1 + Lat2,
    annotate_rtt(T1,T2).
```

Sanity checks

It is important that all necessary information is available in the SKB before starting to construct the multicast tree. Therefore, this helper function checks whether all information is available. If this is not the case, it returns `No.`, causing the main goal to return this answer to the calling C function.

```
% sanity check:  Check first that we have all the
% necessary information
multicast_sanity_check :-
   is_predicate(nr_running_cores/1),
   is_predicate(cpu_thread/4),
   is_predicate(message_rtt/6),
   nr_running_cores(NrRunningCores),
   findall(X,cpu_thread(X,_,_,_),L),
   length(L,NrRunningCores),
   ExpectedNrRTTMeasurements is
      NrRunningCores * (NrRunningCores - 1),
   findall(X, message_rtt(X,_,_,_,_,_), L2),
   length(L2, ExpectedNrRTTMeasurements).
```

7.5 Evaluation

While the final result should lead to higher performance of sending multicast messages, it is equally important for this thesis to prove that few lines of high-level declarative CLP code derives suitable, adaptable policies for forwarding multicast messages. Furthermore, the thesis shows, that the code complexity in terms of lines of ECL^{i}PS^e code is low. In order to cover all the mentioned evaluation goals, the evaluation in this section focuses on three different dimensions:

1. Adaptability to current underlying hardware
2. Code complexity of the ECL^{i}PS^e code
3. Performance of sending multicast messages along the constructed tree

Reducing the code complexity while being automatically adaptive to the underlying hardware without prior knowledge is one of the most important goals of this thesis. Obviously, the result of the policy code (the multicast tree construction) should allow the mechanism of sending multicast messages to be as performant as possible. In the following subsections I show the evaluations of these three dimensions mentioned above.

Function class	LOCs
Tree construction	40
Helper functions	8
Total	48

Table 7.1: Lines of code to construct the multicast tree

7.5.1 Adaptability

As stated in section 1.1.1, every machine looks different nowadays. This also means that the multicast tree needs to look different on every machine. The code must not assume any knowledge about the hardware topology, but instead, it should use the detailed, discovered hardware information stored in the SKB.

As we can see from the rules in section 7.4, the rules use the `cpu_thread/4` to learn about cores and packages as well as the `message_rtt/6` which is measured at runtime and provides latency knowledge of the current machine. None of the rules assume any knowledge in the code. As long as these two facts are available, the policy code derives a valid multicast tree which fits on the current underlying hardware.

7.5.2 Code complexity

It is important, that the code complexity is reduced for programmers. Code complexity is a qualitative metric, expressed by counting the number of lines of code of the shown rules above, without counting comment lines or white lines. With a small number of only 48 LOCs I was able to implement a completely adaptive code which constructs a correct multicast tree on all of our test machines. Maintaining this code, if necessary, is simple as well.

7.5.3 Execution time

The most important metric resource-wise is the execution time required to construct the multicast tree per core. The measurements show that the

execution time is less than 3ms on our test machines. This tree is constructed once when the system starts up and can be reused for every multicast message sent afterwards. The big gain of adaptability and reduced code complexity compared to the execution time forms a good tradeoff.

7.5.4 Effective multicast performance

The ultimate goal of the multicast tree is reaching high performance of sending multicast messages in the system. To evaluate the performance, four different variants of TLB shootdown implementations were compared in terms of latency. The measurements in figure 7.2 show the multicast messaging latencies, without actually performing the TLB shootdown locally, on a 8x4 core AMD Barcelona system[3].

The *Broadcast* protocol uses one single message channel to send TLB shootdown requests to all CPU cores. Each core polls the same cache line and waits for changes by the requesting core. On a TLB shootdown request, each core performs the TLB shootdown and then sends an acknowledgement on an individual message channel back to the requester. When the requesting core updates the shared cache line, it is invalidated in all other core's caches[4, section 7.3]. When N cores are polling the single cache line and an update of it by the requesting core invalidates it in all N cores, the data transfers the interconnect N times. The latency therefore grows linearly with the number of cores.

The *Unicast* protocol uses a separate message channel between the requesting core and every other core. A cache line is therefore only shared between two cores. The requesting core sends individual TLB shootdown requests to every single core leading to sequential processing. As the figure shows, this protocol is better than the broadcast protocol, but still scales linearly with the number of cores.

The *Multicast* tree based approach performs much better compared to the first two approaches. The AMD Barcelona CPU packages have a shared L3 cache which allows having fast local messaging based on cache

[3]The experiment was conducted together with Andrew Baumann, Simon Peter and Akhilesh Singhania.

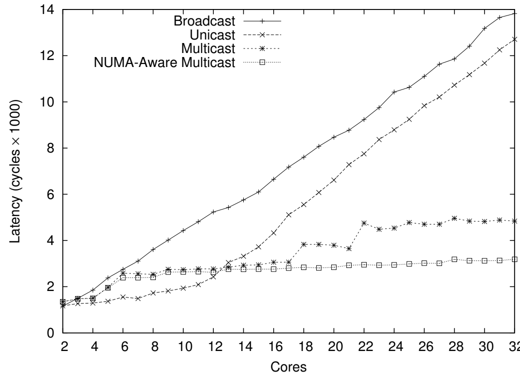


Figure 7.2: TLB shootdown latencies of four different implementations.

lines solely shared by cores on the same package. If only cores on the same package poll a shared cache line, an update sends invalidation messages only within the CPU packages and not over the complete interconnect. This makes local broadcast much cheaper and furthermore it allows CPU packages to work in parallel without sending messages over the interconnect.

Finally, the *NUMA-Aware Multicast* protocol allocates message buffers on the aggregation node's local memory. This protocol performs best as shown in figure 7.2. Because the SKB easily provides NUMA information, the message buffers can be placed easily on the NUMA nodes belonging to the aggregation nodes chosen by the multicast tree algorithm.

This experiment shows that with a small effort it is possible to adapt mechanisms to the current underlying hardware to improve performance drastically.

7.6 Summary

Efficient multicast messaging is an important mechanism used to perform distributed operations within the operating system. The goal is to minimize the waiting time of receiving the acknowledgements of all participating cores. The multicast forwarding tree should be optimal on every machine, even if the topology is not known in advance.

With this example I was able to show that a general algorithm can be implemented with a few lines of code and only based on high-level facts discovered and stored in the SKB at runtime. The algorithm has low code complexity in terms of lines of code and is easily maintainable. It does not assume a priori knowledge, but only uses knowledge gathered at runtime.

The outcome of this algorithm leads to much higher performance of the mechanism using the algorithm's values. Because the configuration of the CPU packages changes rarely, the algorithm itself has to be invoked rarely as well. The mechanisms rely on a fast lookup in a low-level table data structure and to not need to wait for a slow policy algorithm. The clear policy/mechanism separation allows even the system's fast path to use a high-level model of the policies in the SKB to derive an optimal configuration.

Chapter 8

Global Resource Management

Managing hardware resources in modern machines is increasingly complex. The examples in the previous two chapters show how complex hardware can be configured and used efficiently by means of a high-level declarative language. However, hardware resource management has further dimensions, one of which is allocating computation resources to applications.

This chapter shows how to allocate CPU cores to applications, such that their requirements on the resources on the current hardware are satisfied. The allocation code creates a global view and matches application requirements with actual hardware topology knowledge to derive a global allocation which satisfies all application requirements. A high-level model of the hardware in CLP significantly reduces the complexity of incorporating detailed hardware knowledge.

Further, the chapter presents a simple library, which, linked to the application, takes care of interacting with the SKB. Through a few function calls, the application registers hardware requirements. The library takes care of creating and destroying application threads. A simple example in

this chapter shows how easily the application uses the library.

8.1 Introduction

Modern, heterogeneous, many-core architectures require smart management of their resources to yield optimal performance. The smart management of the resources of such systems is important to fully utilize their potential and yield the best throughput of individual applications as well as the optimal overall system performance and utilization. Nowadays, applications are programmed against popular interfaces like POSIX, the Portable Operating System Interface for Unix[132], or Windows but of late also against the paravirtualized interface (PV). These interfaces are highly abstract models and hide underlying hardware differences such as, for example, memory hierarchy, interconnect between cores or the total number of hardware execution contexts. Additionally to hiding hardware differences, these interfaces also completely hide internal allocation policies used by the operating system

Due of this abstract interface, the operating system allocates hardware resources, such as memory or CPU time, to applications in a best-effort way. Traditionally, an application requests only a certain *amount* of resources from the operating system, without specifying any desired resource *properties* in detail. The operating system keeps track of free and allocated resources, such that on new resource requests it can allocate a portion from the free resources. It is however not specified by the application, *which* of the free portions would be best for the application. In this chapter, I argue that applications should in fact not themselves specify, *which* part of the free resources they need, but they should specify *what properties* the resources should have, such that the operating system knows which part would be best to allocate. In section 8.3 I show how applications can specify properties and how they are taken into account by the allocation code.

For certain applications (e.g., short running applications) it is sufficient to get the right amount of resources, even if the location of the resource might not be optimal. However there are applications which not only know

the necessary amount of resources they need, but also what properties they would like to have. As an example, many applications perform better, if memory is allocated in a NUMA-aware manner (an example in this thesis, which clearly performs better, if the memory is allocated in a NUMA-aware manner, is the message buffer of the NUMA-aware multicast tree presented in section 7.5.4).

In general, topology-aware resource management becomes increasingly important to yield the best performance on modern hardware. The NUMA-aware placement of threads and the memory they access is an important example. Basic properties, like NUMA-awareness, are derived implicitly by the operating system. Linux derives the application's needs based on limited monitoring[21]. The default policy is to allocate memory from the NUMA node, which is local to the CPU core on which the process is executing. The *scheduling domains* in Linux allow processes to have affinities with sets of CPU cores and their caches and associated memory. In this case, the application relies on the scheduler to consider NUMA-affinity. It is however not always desired, that the NUMA region of the allocating core is being used. This core might only serve as a coordinating core of the application, while the memory region might be accessed heavily by a working core on a different NUMA region. To account for these cases, *memory allocation policies*[21] can be created in the kernel through additional interfaces such as libNUMA in Linux or lgroup in Solaris to explicitly instruct the kernel about affinities. These interfaces provide a basic mechanism to let an application chose a certain NUMA region for future memory allocations.

However, it is difficult for a single application, and in particular for a developer, to explicitly manage hardware resources for several reasons. First, the abstracting interfaces of today's OSs make it difficult to smartly manage resources at the application level. It quickly becomes difficult for programmers to decide on a good NUMA region. This property changes from machine to machine and would therefore need a deep hardware knowledge gathered somehow by the application. As the thesis already mentioned in section 1.1, no two machines look the same. They might come with a different amount of NUMA regions and with a different scheme of which cores belong to which NUMA regions. Second, interfaces like lib-

NUMA do not provide any means of incorporating the complete memory hierarchy including caches. An application can therefore not use these interfaces to allocate two collaborating threads on the same cache and likewise it cannot directly avoid false sharing by means of these interfaces. Third, these interfaces imperatively set affinity of threads to absolute processor IDs and memory allocation to an absolute NUMA domain. They are thus only a local optimization – based on a purely local view – that might even conflict with other processes running on the same machine. Several concurrently running applications might decide to allocate memory on the same NUMA regions while another NUMA region might be completely empty. As a consequence, these applications most probably pin threads on a small number of cores rather than distributing them over several ones. Applications sharing a machine would need to coordinate their NUMA-affinity but there are no standardized interfaces. Finally, if every applications decides on its own, the same complexity of trying to make sense of hardware information and taking decisions is found in every application.

The lack of information flow between the application and the operating system causes this problem of applications trying to optimize on a given system on their own and with a purely local view. However, if there were a wider interface between the operating system and the applicaiton, a global view of resource preferences could be created in the operating system. It is already the case that the applications know what they need in terms of resources. Likewise, the operating system already has a deep hardware knowledge. It learns about hardware through resource discovery at startup. The knowledge not only includes the amount of resources, but also topology knowledge. In the case of Barrelfish, the SKB contains a detailed description of available hardware such as RAM or CPU cores. The only missing part is an extended interface, which allows combining the two rich pieces of information.

Many proposals exist to improve the information flow between the OS and the applications running on top, but they are seldom used in practice because they push a great deal of complexity towards the developer. So far, the effort of implementing and using them has not paid off because systems were homogeneous and only had few cores. As hardware is get-

ting more complex and heterogeneous nowadays, it is worth rethinking the OS–application interface. The SKB facilitates matching resource property requirements with actual hardware knowledge. In this thesis I am showing how an extended interface can be built such that resource needs of applications can be communicated to the OS. This extended interface provides a global view over different needs of all applications and allows matching individual needs with available hardware resources in a coordinated way. This declarative interface is a radical new approach of unifying resource needs and available resources.

8.2 Background and related work

This section gives an overview of the most popular application programming interfaces and of existing approaches to overcome the missing information flow of current operating system interfaces.

POSIX, the Portable Operating System Interface for Unix[132], is a highly abstract model and hides underlying hardware differences (e.g., memory hierarchy, interconnect between cores, total number of hardware execution contexts) as well as how the operating system works (e.g., resource allocation policies) from the application. It provides the illusion of a complete, homogeneous machine for one application. Consequently, POSIX only presents few opportunities to provide information to the OS (e.g., `madvise`, which is rarely used). This lack of information flow can lead to a suboptimal utilization of hardware resources and to lower application throughput.

Windows' API provides applications a wealth of functionality to choose from[91], including hinting for resource allocation (e.g., NUMA-aware memory allocation) or scheduler activations[7] like threading[93]. Despite the broad API there are only few possibilities for applications to provide feedback to the OS to improve global knowledge and global optimization.

An emerging interface is the paravirtualized interface provided by virtual machine monitors (VMMs). Applications can run directly as domains on a VMM (e.g., Maxine Virtual Edition[103]) Despite leaving the domains a lot of freedom (e.g., own scheduler, memory management), the

VMM provides a homogeneous abstraction of the underlying, heterogeneous hardware. It cannot globally optimize resource utilization as there is no information flow between the domains and the VMM. The shortcomings of homogeneous abstraction and lack of information flow in virtualized environments is, for example, tackled by the symbiotic interface[78]. It widens the interface between the VMM and the guests enabling them to exchange more information.

To overcome the limitations of these interfaces, many ad-hoc fixes have been implemented. Some problems are worked around in user-space in every single application. Examples are own memory management in managed language runtimes such as Java, Haskell, or Prolog and in libraries (e.g., OpenSSL), own buffer caches in DBMSs, or thread pinning for better cache utilization [44, 70, 73, 74, 135]. There are also feedback-oriented improvements in existing systems. Solaris' preemption control[126] allows a process to indicate that it is currently holding a lock and making good progress and thus should not be descheduled. Apple's iOS sends "Memory Warnings" to applications asking them to free up memory[8]. Daemons like VeryNice[60] and the auto nice daemon (AND)[124] renice processes according to configuration files to ensure that they get the necessary amount of CPU time but not more than they need. Some optimizations are potentially dangerous and require emergency functions to keep the system running. With memory overcommitment, for example, the OS assumes that not all processes will concurrently access the complete allocated virtual memory and it grants more memory than it actually has. If processes actually use all available memory, the OS resorts to the out-of-memory (OOM) killer, which kills a random process. Similarly, the VeryNice[60] daemon and the auto nice daemon (AND)[124] can renice processes to a much lower priority or even kill processes if they use too many CPU cycles.

The actors project[1] combines resource reservation with feedback control. The difference between desired resource reservations and actual used resources guides the resource allocator to decide how resources should be allocated to other tasks. The feedback control also allows deciding whether desired resources should fully be granted or whether it is too much of overprovisioning.

Many of these solutions are ad-hoc fixes which try to open existing interfaces to some extent. In the next section I am going to present a more general approach which derives allocation policies globally. The policies can be used by the actual allocation mechanisms.

8.3 Model hardware and global allocation

This section explains how policies for global allocation can be derived such that hardware knowledge is taken into account and application requirements are satisfied. The section explains, how the complexity can be handled by modeling the global allocation in a CLP program.

The first step to create a global view is to create a high-level model of the available hardware. This model includes available resources as well as their topology. Because the SKB already has a deep knowledge of hardware and its topology, these facts can be used to create the hardware model. The second step is allowing the model to represent applications which are using the available hardware. Applications should have a way to register themselves with the SKB. They should be able to upload requirements on the hardware resources. Based on this information, the application part of the model can be constructed. The final step is running a high-level allocation algorithm such that application requirements are met as much as they can. The algorithm uses decision variables within the model which finally show the concrete hardware resource allocation to applications. These steps are explained in more detail in the following subsections.

8.3.1 Hardware model

The hardware model needs to include as many details as possible. The topology model in this thesis includes HyperThreads, cores, shared caches as well as NUMA nodes. The hardware-to-application allocation is modeled as a matrix, where columns are hardware properties and rows represent tasks. A single column represents a single HyperThread. Additional facts in the SKB provide the knowledge of which HyperThreads

Core	0	1	2	3	4	5	6	7
Task 1	X=1	X=0	X=0	X=0	X=0	X=0	X=0	X=0
Task 2	X=0	X=1	X=0	X=0	X=0	X=0	X=0	X=0
Task 3	X=0	X=0	X=1	X=1	X=0	X=0	X=0	X=0
Task 4	X=0	X=0	X=0	X=0	X=1	X=1	X=1	X=1
Shared L3	Cache 0		Cache 1		Cache 2		Cache 3	
NUMA	Node 0				Node 1			

Figure 8.1: Matrix used for global allocation

(i.e., which columns) belong to the same core. Similarly, additional facts about the cache hierarchy provide knowledge of which cores share a cache and therefore which columns or groups of columns belong together in the matrix model. Finally, a group of cores share one NUMA node and therefore an even larger group of columns belong together in terms of NUMA sharing.

8.3.2 Application model

An application is modeled as a set of threads or tasks, each of which is executing on a specific hardware execution context. Every application is represented as a complete row in the matrix. Every field contains a decision variable. Initially, these variables do not have a concrete value assigned. It is a nice feature of ECL^{i}PS^{e} CLP, that it allows constructing data structures with variables without concrete values, as already mentioned in section 2.1.4. Later on, an allocation algorithm assigns concrete values to all decision variables. The concrete value defines, whether a hardware execution context is assigned to a task or not. Whenever a variable holds the value “1”, the task having this value is allowed to execute on that specific hardware execution context, otherwise it is not.

The example in figure 8.1 shows an 8-core machine where a pair of cores shares a cache and 4 cores share one NUMA node. In this example,

every core is explicitly allocated to one task. Task 1 has core 0 allocated, task 1 gets core 1, task 3's allocation includes cores 3 and 4 and finally, cores 4, 5, 6 and 7 are allocated to task 4.

8.3.3 Application requirements

So far, the matrix is able to represent core-to-application allocations. It does not, however, take application requirements into account. There needs to be a way of attaching application requirement to the model. The first step is identifying the most important basic requirements. In a second step, these requirements have to be attached to the model in a reasonable way. For this thesis, I identified the following basic application properties.

Exclusive core allocation might be a requirement of an application. An exclusive allocation of cores to an application provides at least two benefits. First, all threads run at the exact same speed. This is a property important to applications which synchronize threads at the end of each round. Second, the execution is highly predictable. There is no time multiplexing with other tasks, no scheduling is necessary for exclusively allocated cores and there are no cache effects caused by running other tasks in between.

Compute bound applications are limited by the execution of instructions on the CPU core, rather than by memory latencies or I/O. If this is the case, the same task should not run two of its threads on the same core. It therefore ensures that all decision variables belonging to this task are at most '1'. Additionally, explicit allocations of cores to the application would be beneficial, but this is another property (see above) not enforced by the "compute bound" property. If the application does not restrict the maximum number of cores it can use, the allocation assumes that a higher number of cores is beneficial to parallelize and finish the compute-bound task as quickly as possible. The concrete location of the allocated cores is not highly important, as a compute-bound task mostly operates on a small amount of data in the cache or even in registers.

Memory bound is an application property saying that the application is limited by memory latency times rather than by the instructions executed on the CPU core. This is mostly the case for applications, which scan big amounts of memory. These applications might benefit from parallelizing memory accesses. If an application is memory bound, the policy code tries to allocate cores from several NUMA nodes. If it is the only application running at a moment, the code still allocates all cores, unless the application explicitly restricts the maximum amount of cores to be used.

Working set size defines, how much data will be accessed by one worker thread of the application. The data needs to be loaded into RAM, possibly into several NUMA nodes. It needs to be ensured, that at most as many threads are placed on a NUMA node, that the sum of their working set sizes fits onto the NUMA node. Otherwise, the data needs to be placed on other NUMA nodes as well. To ensure, that cores still perform local memory access, it is necessary to allocate cores also from those other NUMA nodes, even if this would not be necessitated by other requirements. As an example, if four cores should be allocated to a compute-bound application on the machine of figure 8.1 and the data size is 10GB, but the NUMA node size is only 8GB, it would be best to allocate two cores of NUMA node 0 and two other cores of NUMA node 1 and distribute the data to both NUMA nodes.

Maximum number of cores restricts how many cores will be allocated to the application at most. Due to internal data structure restrictions, certain applications do not get faster, if more cores are allocated. Collaboratively telling the global allocation code that it is not worth allocating more cores to this application, allows allocating the remaining cores to another application, which might well benefit from having more cores allocated to it.

Minimal number of cores defines, how many cores an application needs at least. An application might want to define this to meet certain service level agreements, which it cannot, if it has a lower number of cores. This

is not always feasible, for example, if the sum of all requested minimal number of cores is greater than the available number of cores. It is also problematic, because an application might request exactly the number of installed cores, leaving no free core for other applications. This would work from a global point of view if no other application defines a minimum number of cores to allocate. If the request cannot be fulfilled, the application gets a smaller number of cores. It always knows how many it finally got.

Cache sharing can be exploited by two cores, if they operate on the same data structure. If an application already knows, that one thread preloads data items and a second threads performs operations on the same data items, the two threads should be allocated on two cores sharing a cache. Alternatively, two HyperThreads on the same core might be used. Experiments have shown, that in some cases it is even worth to run a helper thread which preloads data according to a work-ahead set constructed by the main thread[145]. On the other hand, an application might want to avoid cache sharing or using two HyperThreads on the same core, if it knows that they are accessing completely different data items. This way it can avoid trashing of the cache contents.

8.3.4 Translating requirements to constraints

Now, as some possible properties are defined, they need to be expressed as constraints on the decision variables of the matrix. These constraints are applied even before a concrete instantiation of values is done by the ECL^{i}PS^{e} CLP solver. Attaching constraints of several applications at the same time creates a global view of requirements on the allocation. If conflicting constraints are applied, it is unfeasible to find a valid solution meeting all the constraints. Therefore, the policy code performs sanity checks on the constraints, before applying them. In case that the code finds conflicting constraints, it weakens some of them in a predefined way and applies the modified versions of them. Modifying constraints is much easier than actually finding a complete solution. Therefore, the allocation

code only modifies them, but still relies on the ECL^{i}PS^{e} CLP solver to find a valid solution. In the following paragraphs I show how some of the simpler requirements translate to ECL^{i}PS^{e} CLP constraints at a high-level.

Exclusive core allocation translates to the sum of all decision variables belonging to the same core (i.e. column) is at most “1”. This means, at most one task runs on this core. It is however not necessary, that the core is being allocated at all.

Maximum number of cores translates to the sum of all decision variables belonging to the same task (i.e. row) has to be less or equal to the given value.

Compute bound constrains all decision variables of a task to at most ’‘1”. It does however not need to define *which* cores are suitable, because compute-bound tasks are assumed to work on local cache, not on memory. Therefore NUMA-topology-aware allocation of cores is not necessary. The default allocation tries to allocate as many cores as possible to each task in a fair way. That means that all tasks will get the same amount of cores, if no further restrictions are applied. This property does not prevent the allocation from placing tasks of other applications on the same cores. To guarantee that, the exclusive core allocation property needs to be used (which in many cases makes sense).

Memory bound tries to allocate cores from every NUMA domain in a balanced way. This property translates to the constraint that the sums of all decision variables per NUMA domain should not differ by more than “1”. This means that either the same number of cores is allocated on every NUMA domain or that some NUMA domains contain at most one additional core or at most one core less than the other NUMA domains. It might still be that cores from all NUMA-domains are being allocated, for example if this task is the only one, or if a second task only needs one core, or if there is a second memory-bound task which gets one core per NUMA domain.

8.3.5 Decision variables and concrete topology-aware allocation

The final step is instantiating all decision variables with concrete values. ECLiPSe CLP provides a built-in predicate `labeling/1` to trigger the solver to assign concrete values to all passed variables. After that, the final result contains the values plus additional topology information. Along the core numbers to be used by the different tasks, NUMA node information is passed back to the application as well. This avoids, that the application needs to do a second query of NUMA node affinities to cores. Similarly, the result passed back to the applications contains the knowledge of which cores share a cache and which ones are HyperThreads on the same core. According to this information, the application creates or destroys threads and assigns data items in a NUMA-aware manner according to the information returned to it. This ensures, that each core accesses local memory.

The result is effectively passed to applications by means of upcalls. Upcalls arrive asynchronously to the application, potentially at any time. As section 8.4 explains, the resource manager takes care of parsing the result from the SKB and generating upcalls. To remove complexity from the application and to avoid code duplication, a small framework takes care of interacting with the resource manager and of creating and destroying threads for the application. This framework is explained in section 8.5.

8.4 Resource manager

The SKB is purely reactive (see section 3.4) and therefore does not recompute the global allocation by itself if the set of tasks or their properties change. It is therefore necessary to have an external component to trigger the recomputation in changes of task properties or the number of tasks.

The global allocation framework not only provides the allocation logic in the SKB, but also a *resource manager*, which is a user-space service interacting with the SKB. The resource manager is the mediator between applications and the SKB. Applications register their tasks and the tasks' properties with the resource manager. The resource manager adds, modi-

fies or deletes facts in the SKB and calls the allocation algorithm, whenever it modified something. The re-evaluation potentially affects all applications, even if only one application changes its requirements. The new allocation plan contains the difference between the old allocation and the new one. This means, that only affected tasks are contained in the result returned to the resource manager. The resource manager reads the new allocation, parses it and sends upcalls to every application contained in the result. From the upcalls, the applications learn whether they got more cores or whether they lost cores.

The resource manager does not decide anything by itself. The complete knowledge comes from the algorithm in the SKB. Also, the knowledge to identify applications and to know, where upcalls should be sent, comes from the SKB. The resource manager therefore only serves as an intermediate point which is able to trigger a re-computation of the allocation and which is able to send upcalls to applications.

8.5 Framework to register parallel functions

This section explains how to easily make use of the global allocation facility in an application. Only a few steps are necessary and suddenly, the whole system benefits from it.

To facilitate the programming of parallel applications using the global allocation matrix in the SKB, I implemented a simple framework which allows applications to register parallel functions with the SKB. The framework provides functions to register properties together with the functions. Many applications contain several tasks which can be executed in parallel. These tasks can all be registered as parallel functions with the SKB. They might even have different properties.

A parallel function refers to a C function which can be executed by one or several threads. These functions should have the property that more threads can be created at runtime by the framework, such that several of them execute at the same time. This has especially implications in the data structures used as input and output of a single function, as potentially many functions try to read input and try to generate output, if the function

gets parallelized by the framework.

The framework can be linked as a library to the application. It provides functionality to register and deregister functions as parallel functions. It also allows registering properties with the functions, like compute-bound or the maximal number of cores to be used, for example. The framework takes care of creating and destroying threads for every parallel function without interaction with the application. To synchronize between several threads running the same function, the framework provides some mechanisms explained further in section 8.5.1. Because the application does not need to know how many threads are executing the same function, it also cannot synchronize them by means of the thread-based synchronization primitives. This is the main reason why the framework provides the necessary mechanisms.

The framework interacts with the resource manager. It is responsible to forward information about the functions and its properties to the resource manager, whenever an application registers or deregisters parallel functions with the framework. The resource manager will then take care of the facts and will call the allocation algorithm, as described in section 8.4. The framework also registers with the resource manager to get upcalls for the application. The callback function reads the passed information and based on that creates or destroys threads. It keeps track of the allocated cores and the current NUMA nodes in use. The application can register "constructors" and "destructors" for input and output data structure, such that the framework can call these, whenever a new NUMA domain is assigned to the application or whenever the application lost a NUMA domain. The framework will pass the per NUMA-domain data structure to every new thread it created. This ensures that every thread accesses local memory.

The application can decide to only use one global input and one global output data structure, in which case the framework always passes the same instance as a parameter to every thread.

Because the framework comes as a library which *can* be linked to the application, it is not a requirement, that the application uses the library. It can directly interact with the resource manager and therefore has full control over upcalls and what it wants to do on every upcall. This gives a completely direct control of thread creation and deletion on upcalls and

also of which data structures should be processed by which thread. The framework should only serve as a tool to facilitate the interaction with the resource manager and to deal with threads for applications which do not wish to do that by themselves.

8.5.1 Using the framework

The framework provides a simple interface to register parallel functions. It keeps some internal state and takes care of creating and destroying threads. It also interacts with the resource manager. The example below shows pseudo code of an application which uses worker threads registered as parallel functions with the framework.

The pseudo code below shows a skeleton of a worker thread used in the application, for which the framework dynamically creates and destroys threads:

```
parallel_worker(void *arg) {

  //Optional, if tracing used: Initial data point
  tracing_add_bucket_entry_per_thread(0);

  while(!parallel_should_terminate()) {
    do work, use data structure passed in arg

    //Optional, if tracing used: How much work done so far?
    tracing_add_bucket_entry_per_thread(some value);

    if (producer done && data structure empty) {
      break;
    }
  }
}
```

The worker thread gets the data structure associated with it passed as an argument. The actual work is performed in the while loop. As long as the framework does not ask the thread to terminate, the while loop continues. The worker thread reads data items from the data structure, does some work on them and produces output, which it puts back to the

output part of the data structure. For performance tracing, each worker thread may optionally indicate how much work it has processed so far by adding an entry in every loop. The tracing framework stores the amount together with a time stamp, such that later the main function can output the work amount/timestamp pairs to a file. If it is used at all, an initial timestamp should be created by the thread.

There are two reasons why a thread should terminate. First, if the application loses a core, the framework gets notified by the resource manager and asks the thread to terminate by means of the `parallel_should_terminate()` function (see section 8.5.2 for a discussion). Second, if the input data structure is empty and if it is known, that the data producer is done, the worker thread can terminate, as no more data needs to be processed.

The main function is also given as pseudo code below.

```
main() {

  // Initialize the framework
  parallel_init(...);

  // Optional: use tracing
  tracing_init_buckets("somename");



  // Register parallel_worker() as a function,
  // which can be run by a varying number of
  // threads
  parallelfunction(parallel_worker,
                   global data structure
                   or function to create and destroy
                   NUMA-aware data structures, ...);
```

```
  while (more data to be produced) {
    if (one global data structure) {
      add item to data structure;
    } else {
      chose next NUMA domain;
      // libnuma call
      numa_set_preferred(numa_domain);
      // Get data structure on this NUMA domain
      parallel_get_datastr_numa(parallel_worker,
                                numadomain);
      add item to this data stucture;
    }
  }

  signal producer done;

  // Wait for all threads of this function
  // to terminate
  parallelfunction_wait_terminated(parallel_worker);

  // Optional, if tracing used
  tracing_write_buckets();
  tracing_close();
}
```

The main function of the application initializes the simple framework. Optionally, the application might use the tracing framework to produce performance graphs. The name passed as argument to the "tracing init" function is part of the output file name.

Next, the main function registers the worker thread function as a parallel function. From this point on, the framework starts creating threads executing this function. During runtime, the framework may create more threads or destroy threads respectively indicate, that a thread should terminate by means of the `parallel_should_terminate()` function. Section 8.5.2 discusses destroying threads in more detail.

The main function produces data items to be processed by the worker threads. Depending on the application's architecture, it might either use a single global data structure for input and output for all worker threads or it might want to use a per NUMA-domain data structure. If it uses one global data structure, it adds every data item to be processed to this data

structure. Otherwise, it selects one NUMA-domain, gets the input/output data structure associated with it and adds the data item to this data structure. The worker threads do not need to know explicitly, whether one data structure or a data structure per NUMA node is being used. They get the "right" data structure passed as an argument in any case.

After the main function has produced all data items, it signals that it is done, for example by setting a flag. This is, of course, application-specific and may be implemented differently.

Finally, the main function waits for all worker threads to terminate. The function `parallelfunction_wait_terminated()` is similar to the `pthread_join()` function. It waits for all threads to terminate, but the application does not need to know how many threads there are at that moment. If tracing was used, the results can be written to a file at the end.

8.5.2 Terminating threads

The goal of the global resource allocation code is to ensure that resources are allocated such that the requirements of the applications can be satisfied as much as possible. The global allocation code in the SKB is pure policy code and as such cannot enforce anything. This is especially "problematic" for removing resources or for avoiding that an application creates threads, even if the cores is not assigned to it.

The current implementation relies on the application, to destroy threads by using the `parallel_should_terminate()` function in the thread's while loop.

To actually ensure, that an application destroys threads, the operating system could kill it. This is however problematic as well. The thread might just have acquired a lock on the output data structure when it gets killed.

At least three solutions or a combination of them could solve the problem in a less "destructive" way. The solutions are listed below:

1. Notify the application that it should destroy a thread. Give it some time. If it does not terminate the thread, kill it.

2. Notify the application that it should destroy a thread. Let the OS move the thread to another core, which still belongs to the application. Possibly reduce the priority of this thread.

3. Notify the application that it should destroy a thread. Move threads, which should have been terminated, to a dedicated core, which runs all threads of all applications, which did not terminate threads by themselves.

The current implementation does not perform any of these solutions. At the moment, Barrelfish does not have any access control to cores, which means, that every application can create threads on every core at any time. The research in this thesis is about the feasibility of a declarative global allocation and a framework, which handles thread creation and deletion. It is not about security or enforcing the policies. It would, however, be interesting future work to experiment with the three solutions mentioned above and see, how well application resources, especially cores, can be managed this way.

Even though the thesis is not about enforcing policies, I have some thoughts about how security could be enforced. A “CPU core” capability (or some other mechanism) could be used to control on which cores an application can create threads. There are a number of interesting questions, which arise. First, how should the operating system ensure that an application cannot use the core anymore, if it just lost it? Second, how should the protocol look like, when applications lose a core? Threads need to be terminated by the application or moved by the OS, before the capability gets deleted. The exact way of the “remove core” operation needs to be determined.

8.5.3 Overall architecture

The overall global allocation framework consists of three interacting parts, shown in figure 8.2. First, there is the policy code in the SKB, which decides on the number of cores and concrete IDs for each task. Second, there is the resource manager, which interacts with the SKB by adding, modifying and deleting facts and by calling the allocation algorithm. It also

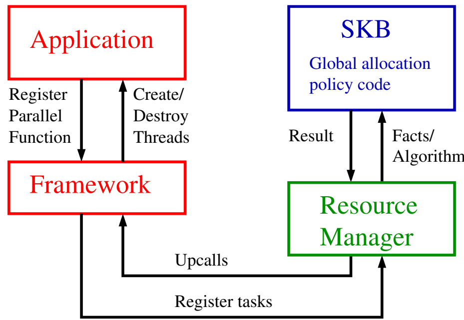


Figure 8.2: Interactions between the application, the framework, the resource manager and the SKB.

interacts with the framework linked to each application. Finally, a simple framework handles thread creation and deletion within the application and interacts with the resource manager.

8.5.4 Use-cases

To validate the framework and the declarative global allocation, I used two real applications as use-cases. Both applications were existing, but needed small modifications in order to make use of the simple framework to register parallel functions. These two use-cases are presented in section 8.6 and section 8.7 respectively.

8.6 Use case 1: pbzip2

The first use-case is pbzip2, a compression tool which compresses or decompresses data in parallel. It is compute bound and has a simple structure, the next section will show.

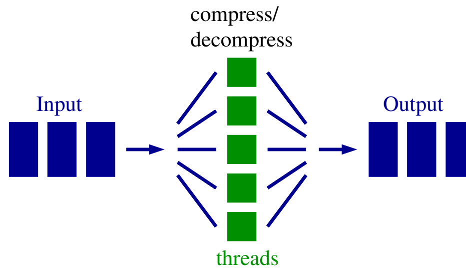


Figure 8.3: pbzip2's architecture

8.6.1 Architecture

pbzip2[50] is a parallel version of the bzip2[119] compression tool. pbzip2 has two parallel phases: compressing data or decompressing data. Several threads running on different cores can therefore execute either the compression or the decompression function in parallel. Each thread compresses or decompresses a different data block. The input data file gets partitioned into several data blocks. pbzip2 enqueues each input block into a common input queue. Each instance of the compression or decompression function reads one input data block at a time, processes it and writes the result data block into a common output queue. The file writer thread writes the result data blocks in the same order as the input blocks to the specified output file. The ordering of the blocks is ensured by block sequence numbers. Figure 8.3 shows the architecture of pbzip2.

The compression and decompression functions are compute-bound and operate on a block size of 900kB. This working set size still fits into L2 cache. Most of the memory accesses can therefore be handled at least by the L2 cache. The property "compute-bound" rather than "memory-bound" is the appropriate one.

Internal data structure limitations allow for a maximum of 4096 threads. Because the execution of the compression or decompression function is completely on a per block basis, pbzip2 benefits from a high number of

cores, ideally all available cores, if it is the only running application.

Deciding on the number of cores

When the user starts the original pbzip2 implementation, it reads the load average queue and applies some heuristics. The output is the number of pthreads to be created. This number remains the same during the complete execution of pbzip2. pbzip2 decides on the number of threads based on a purely local view. It does not know, if one application is about to terminate or if another application is starting at the same time. In both cases, pbzip2 will create a suboptimal number of threads. In the first case, it creates only a few threads and compresses or decompresses a potentially large file with the threads created once at startup time, even if most of the machine becomes idle. In the latter case, pbzip2 creates one thread per available core. Because another application is about to start as well, the cores will be time-multiplexed between the two applications, which again, is suboptimal.

For this thesis, I modified pbzip2 such that it uses the simple framework to register the compression or decompression function as a parallel function with the SKB. It attaches the properties “compute-bound” and “maximum of 4096 threads” to the parallel function. The SKB runs the global allocation code and sends an upcall to pbzip2 containing the information of how many and which cores it can use to run the function. The decision taken in the SKB is based on global knowledge. Therefore, the decision is much more informed. Furthermore, the number of allocated cores can change during runtime. Whenever the allocation changes, the simple framework sends an upcall to pbzip2 again. Based on the new plan, pbzip2 either creates more threads executing the compression or decompression function, or it terminates the threads which were running on cores which it just lost.

8.6.2 Evaluation

The evaluation in this section is specific to pbzip2 and demonstrates that only few changes are necessary to benefit from the global allocation facil-

ity.

First, I evaluate, how many lines of code need to be changed to make an existing application SKB-aware. The goal is to show, whether it is feasible at all to push the responsibility of allocating threads into a framework and finally into the SKB. This is a qualitative metric and depends heavily on the exact application. Second, I evaluate the performance benefit when the application uses the framework, and especially the global knowledge by the SKB, to decide on the core allocation.

LOCs changed

pbzip2 has a specific place where compression or decompression threads are explicitly created. Also, there are specific places where all threads terminate and where the main thread waits for all worker threads to terminate. This structure allowed me to modify pbzip2 at these places in a way, that the functions get registered as parallel functions, instead of explicitly creating threads. Modifying only 25 lines of code made the pbzip2 implementation SKB-aware. Therefore, the simple framework provides a rich enough interface, such that applications which explicitly create a number of threads can be modified in only a few lines of code. They immediately benefit from the global allocation.

Dynamic number of threads

One goal of having a global allocation is adaptability to a changing set of running applications. This means, that pbzip2 should react in terms of number of threads according to new allocations received through upcalls.

In an experiment, I started a pbzip2 instance on a 24 core machine. The instance compressed a 4GB input file. After 60 seconds I started a second pbzip2 instance on the same machine. This instance compressed a 2GB input file. Because it was twice the exact same program, both instances had the same properties. The expected behavior was that the first instance gets all the 24 cores at the beginning. As soon as the second instance starts, the first instance should give up half of the cores (because they have the

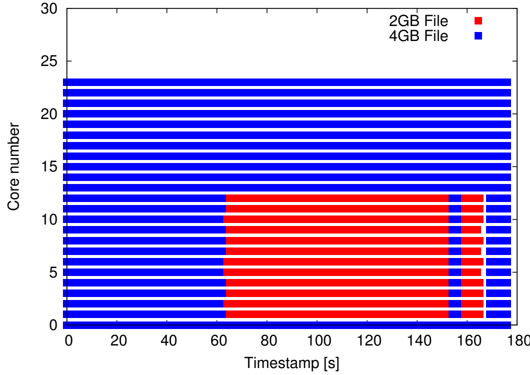


Figure 8.4: Changing number of allocated core per instance

same properties, they are being handled the same). The cores should be allocated to the second instance.

Figure 8.4 shows the core allocation of both instances. The x-axis shows the elapsed time. The y-axis shows the core number allocated to one of the two instances. The diagram shows that after 60 seconds the first instance gives up half of the cores. The second instance gets these cores and compresses the input file for about 90 seconds. After a second short parallel phase, the second instance terminates and releases all cores. The first instance again runs on all 24 cores.

Performance

The performance evaluation shows to what extent performance can be improved, or the overall system throughput can be kept at the same level, when the global allocation code allocates resources in a conflict-free manner.

Figure 8.5 compares the original implementation with the modified one, both on Linux. Both instances compress the same input files. The

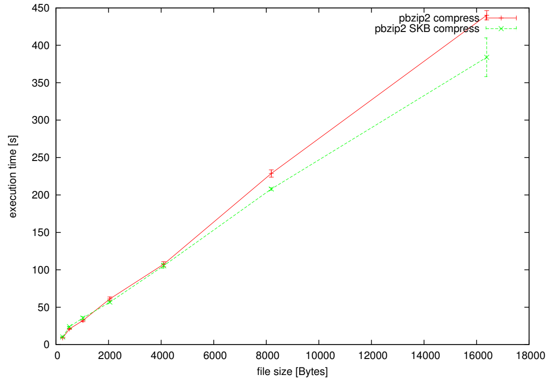


Figure 8.5: Compressing input file: Original vs. SKB-aware version

x-axis shows the input file size and the y-axis the execution time. The modified version performs slightly better. The reason is that the modified version creates threads exactly on the allocated cores and therefore pins the threads explicitly to one core. The original implementation on Linux does not do thread pinning. The threads actually move which causes some performance drop. Figure 8.6 shows a similar behavior, but for decompression.

More interesting is the total throughput of the system. In this experiment, one pbzip2 instance executes in the system for 60 seconds. After 60 seconds, a second instance starts and runs concurrently. As figure 8.4 already shows, both instances run on distinct cores. The sum of the throughputs of both instances is about the same as the throughput of one single application, as figure 8.7 shows. This means, the throughput between second 0 and second 60 is the same as between second 60 and second 140. At the right side of the graph, there is some cleanup noise.

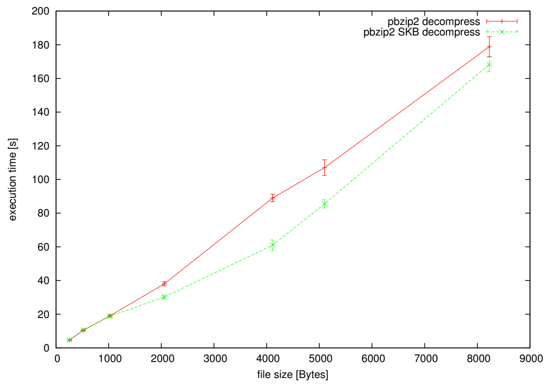


Figure 8.6: Decompressing input file: Original vs. SKB-aware version

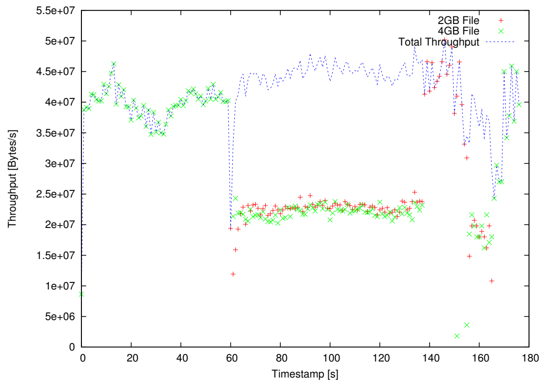


Figure 8.7: Total throughput, if two instances are running

8.6.3 Summary

The framework provides enough mechanisms and a rich enough interface to easily modify an application like pbzip2, which already uses worker threads to process data items. Few lines of code needed to be changed to make it SKB-aware. The global allocation code handles core allocation well, such that the overall system performance can be increased slightly. An important benefit is that by only changing a few lines of code, the modified pbzip2 version can adapt to a changing number of allocated cores at runtime.

The use-case in the next section, a column store, shows that even a more complex and more performance-critical application benefits from the global allocation code.

8.7 Use case 2: Column store

The second application is a modified version of a column store developed in our group[3]. The column store engine executes one scanning thread per core which constantly scans a part of the data in a column oriented way. The column store partitions the data such that all scanning threads scan the same amount. Because the data set is large, NUMA-awareness is important. Still, executing queries on single data items is compute-bound. Due to the synchronization point after one scan, it is important, that the scanning threads run at the same speed. The total latency is determined by the slowest scanning thread.

The column store was modified to become SKB-aware. It uses the framework to register scanning threads with the resource manager and finally with the SKB. Based on upcalls, it creates and destroys scanning threads, as explained further in this section. The work presented in this section has been published recently[48].

8.7.1 Problem

Similarly to pbzip2, the column store needs to decide how many threads it needs to create. Additionally, it needs to decide how it should partition and distribute the data.

Typically, a data base engine (a classical one or a column store engine) has a deep knowledge about its data organizations and its algorithms to process the data. Therefore, it also has a clear knowledge of its requirements from the operating system and especially from the hardware resources. Traditionally, database systems tried to incorporate all system-level information and derive policies by themselves. To do so, database systems tried to circumvent the operating system's policies. Typically the assumption is, that the database system is the only running application and that there is no interference with other applications. This, together with the assumption that the machine configuration does not change, is also one reason, why a purely local policy decision typically worked well.

Nowadays, as machines are getting bigger and bigger in terms of resources (e.g., number of cores, main memory), the machine can be shared by a database system and some other applications. However, in this case, a purely local decision of how resources should be used is not the right way anymore. Instead, the global allocation should be performed. A rich interface should however incorporate as many requirements of the database system as possible to guarantee nice behavior, even if the machine is shared.

Of course, an administrator could statically partition big machines to a fixed number of applications, but this would mean that the set of applications should not change and also, that a big enough portion of the machine would get allocated to each application to handle peaks well. This would lead to overprovisioning which, in most cases, is a waste of resources. A more dynamic allocation of resources based on the current workload and application properties, which should be met, is advantageous. First, the database does not need to be overprovisioned. It can be run together with another application. Still, both applications perform well, if the right set of resources is allocated to them. Second, the database does not need to deal with low-level system resource knowledge, if the system allocates the

right set of resources based on property specifications.

8.7.2 Internal knowledge

The column store has two sets of properties. First, there are the generic properties described in section 8.3.3. These are system properties taken into account during the global allocation. Second, the column store has application-specific properties. The user or database administrator can specify a time, within which queries have to be answered. The column store therefore has to meet a service-level agreement (SLA). Based on the SLA value, the column store decides how many cores are necessary to fulfill the maximum response time. The time depends on the current underlying hardware. The column store uploads an application-specific function to the SKB to compute the minimal number of cores to be used, based on the SLA value and either the available hardware knowledge in the SKB or online measurements performed by the column store. The function decides the minimal number of cores necessary to fulfill the SLA agreement. This value needs to be taken into account by the global allocation code. The column store adds the requirement of a minimal number of cores to the SKB, such that this requirement can be taken into account.

The data size is another important property. The goal is to not overload a NUMA node with data. More correctly, it is actually not possible to overload a NUMA node. Instead, a non-careful allocation would assign data to another NUMA node. However, it is better to explicitly know and control which NUMA nodes contain data and on which NUMA nodes there should be threads processing the data.

The actual minimal number of cores is therefore determined by two metrics. The first one determines how many cores are necessary to fulfill the SLA agreement, i.e., the maximal response time. The second one determines how many NUMA nodes are necessary such that data can be distributed in chunks of at most NUMA node sizes. Each NUMA node should contain at least one core allocated to the column store. This is the minimum number of cores to allocate based on data size and NUMA node sizes. The greater value determines the actual minimal number of cores to be used. The column store passes this value to the global allocation

framework.

8.7.3 Registering scanning function

The column store uses the simple framework presented in section 8.5 to register scanning threads as flexible parallel functions. The properties, which it registers along with the function, are first, that it is compute-bound, second, that it needs an exclusive core allocation, and third, that a minimal number of cores is requested. As more cores are not necessary, the column store will not use additional ones, even if more would be assigned to it. Therefore it restricts the maximum number of cores to be allocated to the same number as the minimal cores to be allocated.

Like every application, the resource manager upcalls the column store and provides the concrete allocated cores together with NUMA domain information. The column store uses the result to create threads on the respective cores and starts distributing data. Finally, it starts the scanning threads on these cores and processes the queries.

8.7.4 Evaluation

Deployment on different systems

As mentioned in section 1.1, no two machines look the same. Each machine has a different amount of CPU cores and a different amount of NUMA nodes. Also, NUMA nodes are of different sizes. Still, the column store has to fulfill the SLA agreement on every machine and the data should be partitioned such that no NUMA node gets overloaded.

In this experiment we run the modified column store on four different machines[1]. The user data size is 8GB. An additional 1GB of metadata has to be added. This leads to a total data size of 9GB. Table 8.1 shows the number of allocated cores on every system as well as the actual execution time. The column store violates the SLA agreement once. This is due to the fact that the application-specific function, computing the minimal

[1] The experiment was conducted together with Tudor Salomie and Jana Giceva.

Hardware characteristics	SLA req.	#Cores by SLA	#Cores by NUMA	Final #cores	Size of partition	Actual mean response time
32 cores	2s	8	1	8	1GB	1.66s
128GB RAM	4s	4	1	4	2GB	3.27s
32GB/node	8s	2	1	2	4GB	6.54s
32 cores	2s	8	5	8	1GB	2.18s
16GB RAM	4s	4	5	5	1.6GB	3.55s
2GB/node	8s	2	5	5	1.6GB	3.55s
16 cores	2s	8	3	8	1GB	1.68s
16GB RAM	4s	4	3	4	2GB	3.25s
4GB/node	8s	2	3	3	2.67GB	4.33s
48 cores	2s	8	1	8	1GB	1.87s
128GB RAM	4s	4	1	4	2GB	3.71s
16GB/node	8s	2	1	2	4GB	7.37s

Table 8.1: Deployment of the SKB-aware column store on different machines.

number of cores, is derived from an average measurement and has not yet been updated on this concrete machine.

For the second machine it is necessary to use 5 NUMA nodes to distribute the data size of 9GB to different NUMA nodes. Even though 4 or 2 cores would be sufficient to meet the SLA requirement of 4s and 8s respectively, at least 5 cores will be allocated. This is because data should be processed by a local core on every NUMA node. 5 NUMA nodes are allocated to the column store, therefore at least 5 cores (at least one per NUMA node) have to be allocated to the column store.

Running the column store and another application concurrently

The column store is a latency critical application. After every round of scanning data, all threads synchronize. The overall latency is therefore determined by the slowest scanning thread. Ideally, each thread executes at the same speed. This means, that there should be no interference between the column store's threads and other applications.

By registering the scanning threads with the SKB, the global allocation code ensures exclusive allocation of cores to scanning threads. Figure 8.8

shows the performance implications of the column store running concurrently with another compute-bound application. The baseline shows the column store running as the only application. The second experiments shows the performance when an application runs on one of the cores allocated to the column store. The third scenario uses the SKB. The global allocation code removes the core, on which the other application runs, from the column store. Therefore the column store runs on one core less, but does not interfere with the other application.

Figure 8.8 shows that the SKB-aware version of the column store is running with only 47 cores instead of 48 cores, but, without interfering with the other application, performs better than the 48-core version which interferes with the other application. It is therefore important that a global view decides on which cores should be assigned to which application.

This information cannot easily be derived by the column store. It depends on the scheduling state of the OS and on the knowledge of the other applications. It does not make much sense for the column store to try to take decisions based on local knowledge.

Still, the column-store specific information is taken into account by the operating system, because the column store pushes its resource requirements to the SKB. This creates a global view of system state and application specific requirements in the SKB and both can be taken into account.

In a second experiment[2], the column store is the only running task initially. After every 5 minutes, a new compute-bound application starts. In a naive setting, every application decides on its own on which core it wants to run, based on purely local knowledge. The experiment shows the performance drop which happens, if all naive applications decide to start on core 0. Figure 8.9 shows the results.

The naive column store engine has a dramatically higher response time. Already after the first new application enters the system, the column store cannot meet the SLA agreement anymore, because its scanning thread on core 0 runs at half of the speed of the other scanning threads. The SKB-aware column store gets upcalled and informed that it has lost

[2]The experiment was conducted together with Tudor Salomie and Jana Giceva.

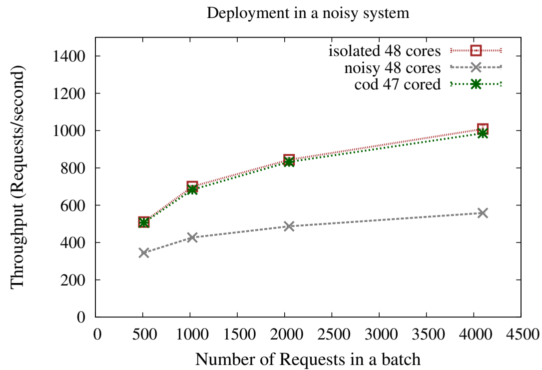


Figure 8.8: CSCS performance when deployed in a noisy system

a core. The column store engine redistributes the data to other cores and continues executing queries using the remaining threads, which, however, run all at the same speed.

In the naive scenario, the next application starting on core 0 causes the next performance drop, while the SKB-aware version gets another upcall telling the column store that it has lost a second core. Even though the column store has lost two cores now, it can still meet the SLA agreement, because all remaining threads run at the same speed.

8.7.5 Summary

The modified version of the column store uses the same framework as pbzip2. This shows, that the framework is general enough to handle even more complex applications like, for instance, the column store.

The column store registers the scan function as a parallel function with the resource manager and finally the SKB and gets upcalls telling it, which cores it can use now or which ones it just lost. This makes the column store adaptive to a changed environment, especially when new applications enter or leave the system. Also, the high-level requirements on hardware

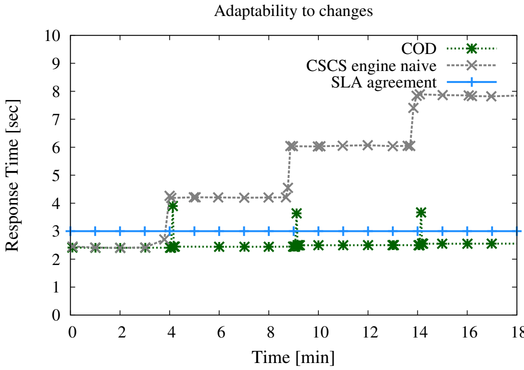


Figure 8.9: Adaptability to changes in the system

resources facilitate the column store, because it does not need to learn about specific hardware knowledge by itself. The allocation algorithm in the SKB incorporates the column store's properties and returns suitable hardware resources according to these properties.

8.8 Evaluation of the allocation policy code

The previous sections showed that the global allocation is easy to use in applications. They also evaluated the benefits of the specific applications. This section evaluates the global allocation code itself. The focus is on code complexity and maintainability of the policy code.

The section also summarizes the execution time of the algorithm. While, as already mentioned earlier, it is not the main focus of this thesis to produce the most performant allocation code itself, at least reasonable performance is necessary to claim that it is a useful framework.

Class	LOCs
Matrix	39
Algorithm	23
Knowledge access and constraints	153
Interface	77
Output	52
Sanity-checking	20
Misc	20
Total	384

Table 8.2: LOCs to implement global allocation in the SKB.

8.8.1 Code complexity

Table 8.2 lists the lines of code needed to implement the global allocation in the SKB. This code consists of several parts. First, the construction of the decision matrix based on hardware information is a core piece of code. Second, interface functions prepare and store the passed facts. The resource manager calls these interface functions and passes application properties as input parameters to the interface functions written in ECL^{i}PS^e. Third, sanity-checking functions ensure that there are no conflicting properties. This is important, otherwise ECL^{i}PS^e would not be able to find a solution and would simply output "No.". Finally, there are helper functions enabling all the main functions to access the necessary knowledge in an easy way.

As table 8.2 shows, the core functionality is implemented in a few lines of code. 20 lines of CLP code are used to construct the decision matrix and 23 lines of CLP code are used to call the various functions which attach constraints to the matrix. The "knowledge access and constraints" functions interpret application properties and hardware knowledge and derive and attach constraints such that application requirements are handled the correct way. This category needs a relatively large amount of code. The interface consists of several high-level functions which create the right facts or, in the case of removing a task, ensure that all associated facts are

deleted and the task's state in the SKB is cleaned up. The "output" category includes goals to compute the difference between the new and the old allocation (see section 8.4) and to transform the algorithm's result into a suitable output list. Finally, there is a small amount of sanity-checking code and various small helper functions, with 20 lines of CLP code each.

The total amount of 384 lines of code is well maintainable. Furthermore, it is easily extensible. If new requirements have to be modeled, it is sufficient to add a new small goal which transforms the requirement based on the necessary hardware knowledge and/or application properties into an additional constraint on the matrix.

The model follows a clear policy/mechanism separation as all other parts relying on the SKB. For this part of the codes, it means that the allocation policies are completely separate from the resource manager and from the framework which creates and destroys threads. As long as the interface does not change, the policy code can be changed without needing to change any line of the framework or the resource manager. This gives a lot of freedom to experiment with new policies. Applications immediately follow the new policies.

8.8.2 Execution time

It is difficult to measure execution time in the general case. The execution time heavily depends on the number of concurrently registered applications and on the amount and type of their resource requirements. The more detailed the requirements are, the more complex constraints have to be applied to the matrix, thus resulting in a higher execution time in the ECL^{i}PS^{e} solver. The thesis does therefore not provide a graph of the execution time, but instead a table for selected configurations with actual execution time measurements. The goal is to show that the execution time for re-evaluating the global allocation is reasonable. The results are shown in table 8.3.

Applications	Execution time
Column store	5ms
Column store + compute-bound application	13ms
1 pbzip2 instance	5ms
2 pbzip2 instances	13ms

Table 8.3: Execution time for different configurations

8.9 Summary and future work

Global resource allocation is getting increasingly important on modern machines. Modern machines can be spatially shared among many applications. Even database engines can run concurrently to other applications and still perform well. To ensure good performance, a global view over all running applications and their requirements in terms of hardware resources is critical in order to derive smart resource allocation policies.

Using a model which unifies hardware knowledge with application requirements provides a global view and allows deriving allocation policies which improve the overall system performance. The high-level declarative nature of the implementation reduces the code complexity. Only a few simple functions written in ECL^{i}PS^{e} achieve a good result. The simple functions are well maintainable and easy to change and adapt to future needs, if necessary.

The clear policy/mechanism separation allows evolving the policy code without changing the applications, the framework or the resource manager. Changing the policy code immediately impacts application behavior. It can be validated immediately, whether new policy code actually leads to desired behavior.

The global allocation framework can be easily extended to managed language runtimes. At the lower end of the software stack, the managed language runtime could interact with the framework and register parallel functions and properties and receive callbacks with allocation plans. In this scenario, it might even make sense, if the managed language runtime directly interacts with the resource manager. The managed language run-

time would get callbacks directly and could explicitly manage its threads. At the upper half of the software stack, there are applications written in the managed language runtime. The runtime has deep knowledge of the application code. Not only does it see the code, but there is still semantic information available which might help the runtime to derive the right application properties in an adaptive way at runtime.

A high-level functional language has the freedom to parallelize certain operations, like, for example, a `map` function which applies an operation to a list. It is not important for the programmer to known the number of actual threads assigned to the program, because the final result is the same, independently of whether one or several threads worked in parallel on portions of the list. Combining a high-level language with this framework is an interesting future work.

Chapter 9

Conclusion

9.1 Summary

The hypothesis of this thesis was that if the operating system had a facility to reason about the underlying hardware, it can better adapt to it and make use of the available hardware. Further, code complexity can be taken out of the operating system's mechanisms, making the mechanisms much simpler.

In this thesis I was able to show that applying high-level declarative language techniques allows dealing with the increased hardware complexity found in current machines. Adaptability to the underlying hardware can easily be expressed by means of declarative algorithms that describe *what* goal is to be achieved, but not *how* to get there. Because the algorithms are based on high-level knowledge, which is abstracted from the hardware, they work on new machines, even if they are not known at the design time of the algorithm. The high-level reasoning enables the system to derive new knowledge in the future by combining facts in an unforeseen way at the time of designing the algorithm.

By using high-level language techniques for reasoning about the system, the complexity typically involved in mechanism code can be taken

out. This greatly simplifies mechanism code, because it only needs to apply the policy parameters derived outside the mechanism code. The mechanism code does not need to decide anything itself and especially, it does not need to handle special cases, independent of the underlying hardware.

The use-cases presented in this thesis prove that the operating system does indeed adapt to the underlying hardware by using high-level descriptions of the goal to be reached, which in turn are based on high-level knowledge of the current underlying hardware. The algorithms used are implemented in relatively few lines of code. This makes them well understandable and easy to maintain. More concretely, the use-cases show how simple algorithms can lead to much better performance on the one hand (for example, in the multicast messaging case) and how difficult hardware configuration problems can be solved (for example, in the PCIe case) with few natural hardware configuration rules.

Obviously, more case studies in the context of operating systems designs could be done to prove the usefulness of applying high-level declarative languages to reason about hardware. The next section sketches ideas for future work.

9.2 Directions for future work

Having the SKB as a basis to implement reasoning algorithms, there are many extensions and further use-cases possible. There are different levels at which it would be interesting to explore to what extent high-level languages can help to reduce complexity.

At the hardware configuration level, algorithms in the SKB can help to deal with more classes of hardware, not presented in this thesis. Examples include USB, which needs configuration on hotplug events, even if it is not at the same level of complexity of PCIe configuration. As history shows, new hardware, which is constantly arising, is likely to complicate hardware management, rather than facilitating it. Furthermore there is a clear trend towards hotlpugging almost everything. While today it is already normal, that USB devices can be hotlpugged on commodity machines, there is a trend towards PCIe hotplugging (today only in bigger server ma-

chines), hotplugging of cores and memory (again, supported by big server machines, but not yet by commodity desktop machines), and hotplugging of new external devices, such as, for example, Thunderbold. The operating system has to deal with hotplug and hot-unplug events at a low-level, but it also has to decide about resource allocations in a much more dynamic way, compared to the almost static hardware configurations found in past and today's commodity systems. Dependencies have to be resolved and ensured and decisions on which resources and how they should be allocated have to be taken. As machines are getting bigger, power-save modes of single pieces of hardware are becoming more important. Deciding which devices to turn off or turn on again and at which time and for how long, becomes more complex. Turning-off times, re-activating times, the amount of energy which can be saved, and the cost of potentially moving running tasks, all have to be considered to derive reasonable power-management policies.

Deciding on computation placement is getting more important, as machines are getting bigger. These decisions become even more complex, once machines become even more heterogeneous than today. The global allocation framework presented in this thesis is a first step towards deciding on core-to-task allocations based on high-level requirements descriptions of the applications. A deeper research in this direction will provide more insight, on how application requirements can be matched with heterogeneous hardware. More fine-grained hardware properties have to be taken into account, for example, whether a core is able to run all the instructions an application wants to execute. If an application needs precise floating point, it needs to run on a core which supports that. If an application makes use of a cryptographic instruction set extension, it needs a core that supports that. In both cases, it may be the case in the future that not all cores support the complete instruction set. Cache-coherency (or the lack of it) may constrain the application on a certain group of cores, if the application needs cache-coherency and the hardware provides islands of coherent cores. While it is not yet totally clear exactly, what future hardware will look like, it is likely that the complexity will grow and that operating systems will have to deal with the examples mentioned above.

With bigger and more complex machines it might be worth implement-

ing applications in high-level managed languages. These languages can derive concurrency and create threads in cases where data can be processed in parallel, without changing the semantics of the program and without the programmer having created threads explicitly. Managed language runtimes have a rich interface to the applications running on top of them. They can monitor the application and derive resource requirements which suit the application best. Additionally, managed language runtimes still have access to semantic information. It is not just machine code, which executes on top of the runtime and accesses some memory addresses. Instead, the runtime knows what functions access what type of objects. It also knows, which threads communicate with each other over which objects. This information provides more insight and allows the runtime to derive better and more detailed requirements, which it can register with the operating system. By extending a managed language runtime and by extending the global allocation code, it is possible to let them collaborate better and to use the available heterogeneous hardware much better. When a managed language runtime gets extended such that it collaborates with the operating system in terms of resource management, suddenly all applications benefit from the global allocation framework. The effort of modifying a managed language runtime has to be taken only once. Unmodified legacy applications benefit immediately from global resource allocation decisions. This is much better than modifying many legacy applications written in a language like C.

Finally, it would be worth to explore other constraint logic programming engines. ECL^{i}PS^{e} is extremely expressive and allows experimenting almost with no limitations. On the other hand, it is not the fastest language. As mentioned in section 3.8.2, more modern solvers might improve performance.

Bibliography

[1] The ACTORS project. http://www.actors-project.eu/.

[2] Aït-Kaci, H. *Warren's abstract machine: a tutorial reconstruction*. MIT Press, Cambridge, MA, USA, 1991.

[3] Alonso, G., Kossmann, D., Salomie, T.-I., and Schmidt, A. Shared Scans on Main Memory Column Stores. Tech. Rep. 769, ETH Zürich, Systems Group, Department of Computer Science, Zürich, Switzerland, July 2012.

[4] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Sept. 2007. Publication number 24593.

[5] AMD. *CPUID Specification*, September 2010.

[6] Anderson, E., Hobbs, M., Keeton, K., Spence, S., Uysal, M., and Veitch, A. Hippodrome: Running Circles Around Storage Administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST '02.

[7] Anderson, T. E., Bershad, B. N., Lazoswka, E. D., and Levy, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Threads. *ACM Transactions on Computer Systems 10* (1992), 53–79.

[8] APPLE. iOS Application Programming Guide. http://developer.apple.com/library/ios/DOCUMENTATION/iPhone/Conceptual/iPhoneOSProgrammingGuide/Performance/Performance.html.

[9] APPLE. Apple Support. http://kbase.info.apple.com/, Oktober 2012.

[10] APT, K. R., AND WALLACE, M. G. *Constraint Logic Programming using ECL^{i}PS^e*. Cambridge University Press, 2007.

[11] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., BURNETT, N. C., DENEHY, T. E., ENGLE, T. J., GUNAWI, H. S., NUGENT, J. A., AND POPOVICI, F. I. Transforming policies into mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating System Principles* (2003), SOSP '03, pp. 90–105.

[12] BALDWIN, J. H. Multiple passes of the FreeBSD device tree. In *BSDCan Conference* (May 2009). http://www.bsdcan.org/2009/schedule/attachments/83_article.pdf.

[13] BALDWIN, J. H. About hot-plugging support in FreeBSD. http://www.mavetju.org/mail/view_message.php?list=freebsd-arch&id=3106757, Feb. 2010.

[14] BARKER, V. E., O'CONNOR, D. E., BACHANT, J., AND SOLOWAY, E. Expert systems for configuration at Digital: XCON and beyond. *Commun. ACM 32* (March 1989), 298–318.

[15] BARRELFISH PROJECT. The Barrelfish Research Operating System. http://www.barrelfish.org/, May 2012.

[16] BARRELFISH PROJECT. The Barrelfish Wiki. http://wiki.barrelfish.org/, May 2012.

[17] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating System Principles* (Oct. 2009), SOSP'09.

[18] BAUMANN, A., PETER, S., SCHÜPBACH, A., SINGHANIA, A., ROSCOE, T., BARHAM, P., AND ISAACS, R. Your computer is already a distributed system. why isn't your OS? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems* (2009), HotOS'09, pp. 12–12.

[19] BAXTER, S. GPU Performance. http://www.moderngpu.com/intro/performance.html, 2011.

[20] Bison: GNU parser generator. http://www.gnu.org/software/bison/.

[21] BLIGH, M. J., DOBSON, M., HART, D., AND HUIZENGA, G. Linux on NUMA Systems. In *Proceedings of the 2004 Ottawa Linux Symposium* (July 2004), pp. 89–101.

[22] BORKAR, S. Thousand core chips: a technology perspective. In *Proceedings of the 44th Annual Design Automation Conference* (2007), pp. 746–749.

[23] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation* (Dec. 2008), OSDI'08, pp. 43–57.

[24] BRATKO, I. *Prolog programming for artificial intelligence*. Adison-Wesley, Harlow, England, 2001.

[25] BUDRUK, R., ANDERSON, D., AND SHANLEY, T. *PCI Express System Architecture*. Addison Wesley, 2004.

[26] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation* (2006), OSDI'06.

[27] CARLSSON, M., OTTOSSON, G., AND CARLSON, B. An open-ended finite domain constraint solver. In *Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics,*

and Programs: Including a Special Track on Declarative Programming Languages in Education (1997), PLILP '97, pp. 191–206.

[28] CERI, S., GOTTLOB, G., AND TANCA, L. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. on Knowl. and Data Eng. 1*, 1 (Mar. 1989), 146–166.

[29] CHEN, X., MAO, Y., MAO, Z. M., AND VAN DER MERWE, J. Declarative configuration management for complex and dynamic networks. In *Proceedings of the 6th International Conference on emerging Networking EXperiments and Technologies* (2010), Co-NEXT '10, pp. 6:1–6:12.

[30] CISCO. ECL^{i}PS^{e}. http://www.eclipse-clp.org.

[31] CONWAY, P., AND HUGHES, B. The AMD Opteron northbridge architecture. *IEEE Micro 27*, 2 (2007), 10–21.

[32] DAGAND, P.-E., BAUMANN, A., AND ROSCOE, T. Filet-o-Fish: practical and dependable domain-specific languages for OS development. In *Proceedings of the Fifth Workshop on Programming Languages and Operating Systems* (2009), PLOS '09, pp. 5:1–5:5.

[33] DAGAND, P.-E., BAUMANN, A., AND ROSCOE, T. Filet-o-fish: practical and dependable domain-specific languages for os development. *SIGOPS Oper. Syst. Rev. 43*, 4 (Jan. 2010), 35–39.

[34] DE MOURA, L., AND BJRNER, N. Z3: An Efficient SMT Solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), TACAS'08.

[35] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating System Principles* (2007), SOSP '07, pp. 205–220.

[36] DIAZ, D. GNU Prolog. http://www.gprolog.org/, 2011.

[37] DISTRIBUTED MANAGEMENT TASK FORCE, INC. *Common Information Model (CIM) Standards*. Portland, OR, USA, April 2008. http://www.dmtf.org/standards/cim/.

[38] DUNHAM, S. N. Method for allocating system resources in a hierarchical bus structure, July 1998. US patent 5,778,197.

[39] ELKADUWE, D., DERRIN, P., AND ELPHINSTONE, K. Kernel design for isolation and assurance of physical memory. In *Proceedings of the 1st workshop on isolation and integration in embedded systems (IIES '08)* (2008), pp. 35–40.

[40] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, JR., J. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating System Principles* (Dec. 1995), SOSP'95, pp. 251–266.

[41] EUGSTER, P. T., AND GUERRAOUI, R. Content-based publish/subscribe with structural reflection. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems - Volume 6* (2001), COOTS'01, pp. 10–10.

[42] FATAHALIAN, K., AND HOUSTON, M. A closer look at GPUs. *Commun. ACM 51*, 10 (Oct. 2008), 50–57.

[43] FATAHALIAN, K., AND HOUSTON, M. GPUS: A Closer Look. *Queue 6*, 2 (Mar. 2008), 18–28.

[44] FEDOROVA, A., SELTZER, M., SMALL, C., AND NUSSBAUM, D. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proceedings of the 2005 USENIX Annual Technical Conference* (2005), ATC '05, pp. 26–26.

[45] Flex: The Fast Lexical Analyzer. http://flex.sourceforge.net/.

[46] FREEDESKTOP.ORG. D-BUS. http://dbus.freedesktop.org/, March 2012.

[47] Geambasu, R., Levy, A. A., Kohno, T., Krishnamurthy, A., and Levy, H. M. Comet: an active distributed key-value store. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (2010), OSDI'10, pp. 1–13.

[48] Giceva, J., Salomie, T.-I., Schüpbach, A., Alonso, G., and Roscoe, T. Cod: Database / operating system co-design. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research* (January 2013), CIDR'13.

[49] Giceva, J., Schüpbach, A., Alonso, G., and Roscoe, T. Towards database / operating system co-design. In *Proceedings of the 2nd workshop on Systems for Future Multi-core Architectures* (April 2012), SFMA'12.

[50] Gilchrist, J. Parallel Compression with BZIP2. In *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems* (November 2004), PDCS'04, pp. 559–564.

[51] gpgpu.org. Gpgpu. http://gpgpu.org/.

[52] Greenhalgh, P. Big.LITTLE Processing with ARM CortexTM-A15 & Cortex-A7, September 2011.

[53] Gschwind, M. The Cell Broadband Engine: exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming 35*, 3 (2007), 233–262.

[54] Hanus, M. Multi-paradigm declarative languages. In *Proceedings of the 23rd international conference on Logic programming* (Berlin, Heidelberg, 2007), ICLP'07, Springer-Verlag, pp. 45–75.

[55] Harris, T., Abadi, M., Isaacs, R., and McIlroy, R. AC: composable asynchronous IO for native languages. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (2011), OOPSLA '11, pp. 903–920.

[56] HARRIS, T., ABADI, M., ISAACS, R., AND MCILROY, R. AC: composable asynchronous IO for native languages. *SIGPLAN Not. 46*, 10 (Oct. 2011), 903–920.

[57] Haskell. http://www.haskell.org/haskellwiki/Haskell, 2012.

[58] HELD, J., BAUTISTA, J., AND KOEHL, S. From a few cores to many: A tera-scale computing research overview. White paper, Intel, Sept. 2006. ftp://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf.

[59] HEWLETT-PACKARD, INTEL, MICROSOFT, PHOENIX, TOSHIBA. *Advanced Configuration and Power Interface Specification, Rev. 4.0a*, Apr. 2010. http://www.acpi.info/.

[60] HOLLAND, S. VeryNice. http://thermal.cnde.iastate.edu/~sdh4/verynice/.

[61] HOVEL, D. Using Prolog in Windows NT network configuration. In *Proceedings of the Third Annual Conference on the Practical Applications of Prolog* (1995).

[62] HOWARD, J., DIGHE, S., HOSKOTE, Y., VANGAL, S., FINAN, D., RUHL, G., JENKINS, D., WILSON, H., BORKAR, N., SCHROM, G., PAILET, F., JAIN, S., JACOB, T., YADA, S., MARELLA, S., SALIHUNDAM, P., ERRAGUNTLA, V., KONOW, M., RIEPEN, M., DROEGE, G., LINDEMANN, J., GRIES, M., APEL, T., HENRISS, K., LUND-LARSEN, T., STEIBL, S., BORKAR, S., DE, V., VAN DER WIJNGAART, R., AND MATTSON, T. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *International Solid-State Circuits Conference* (Feb. 2010), pp. 108–109.

[63] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference* (2010), USENIX ATC'10.

[64] HYPERTRANSPORT CONSORTIUM. HyperTransport. http://www.hypertransport.org/.

[65] INTEL. *Intel Processor Identification and the CPUID Instruction*, May 2012.

[66] JANIC, M., AND VAN MIEGHEM, P. On properties of multicast routing trees: Research Articles. *Int. J. Commun. Syst. 19*, 1 (Feb. 2006), 95–114.

[67] JSON: JavaScript Object Notation. http://www.json.org/.

[68] KAMP, P.-H. Rethinking/dev and devices in the UNIX kernel. In *Proceedings of the BSD Conference 2002 on BSD Conference* (2002), BSDC'02, pp. 9–9.

[69] KAUER, B. ATARE: ACPI tables and regular expressions. Tech. Rep. TUD-FI09-09, TU Dresden, Faculty of Computer Science, Dresden, Germany, Aug. 2009.

[70] KAZEMPOUR, V., FEDOROVA, A., AND ALAGHEBAND, P. Performance implications of cache affinity on multicore processors. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing* (2008), Euro-Par '08.

[71] KHRONOS OPENCL WORKING GROUP. *The OpenCL Specification. Version 1.2.*, November 2011.

[72] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating System Principles* (Oct. 2009), SOSP'09.

[73] KLUG, T., OTT, M., WEIDENDORFER, J., AND TRINITIS, C. autopin – Automated Optimization of Thread-to-Core Pinning on Multicore Systems. In *Proceedings of the 1st Workshop on Programmability Issues for Multi-Core Computers* (2008), MULTIPROG '08.

[74] KRIEGER, C. D., AND STROUT, M. M. Performance Evaluation of an Irregular Application Parallelized in Java. In *Proceedings of the*

2010 39th International Conference on Parallel Processing Workshops (2010), ICPPW '10, pp. 227–235.

[75] KROAH-HARTMAN, G. udev – A Userspace Implementation of devfs. In *Proceedings of the 2003 Ottawa Linux Symposium* (July 2003).

[76] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (1998), 133–169.

[77] The Linux Assigned Names and Numbers Authority. http://www.lanana.org/.

[78] LANGE, J. R., PEDRETTI, K., DINDA, P., BRIDGES, P. G., BAE, C., SOLTERO, P., AND MERRITT, A. Minimal-overhead virtualization of a large scale supercomputer. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (2011), VEE '11, pp. 169–180.

[79] LEVIN, R., COHEN, E., CORWIN, W., POLLACK, F., AND WULF, W. Policy/Mechanism separation in Hydra. In *Proceedings of the 5th ACM Symposium on Operating System Principles* (Nov. 1975), SOSP'75, pp. 132–140.

[80] LIVNY, M., BASNEY, J., RAMAN, R., AND TANNENBAUM, T. Mechanisms for high throughput computing. *SPEEDUP Journal 11*, 1 (June 1997).

[81] LLOYD, J. W. Practical Advantages of Declarative Programming. In *Joint Conference on Declarative Programming* (1994).

[82] LOO, B. T., HELLERSTEIN, J. M., STOICA, I., AND RAMAKRISHNAN, R. Declarative routing: extensible routing with declarative queries. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications* (2005), SIGCOMM '05, pp. 289–300.

[83] LOSH, M. W. devd: a device configuration daemon. In *Proceedings of the BSD Conference 2003 on BSD Conference* (2003), BSDC'03, pp. 2–2.

[84] MARSH, B. D., SCOTT, M. L., LEBLANC, T. J., AND MARKATOS, E. P. First-class user-level threads. In *Proceedings of the 13th ACM Symposium on Operating System Principles* (Oct. 1991), SOSP'91, pp. 110–121.

[85] MCILROY, R., AND SVENTEK, J. Hera-jvm: a runtime system for heterogeneous multi-core architectures. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications* (2010), OOPSLA '10, pp. 205–222.

[86] MENZI, D. Support for heterogeneous cores for Barrelfish. Master's thesis, ETH Zürich, Zürich, Switzerland, April 2011.

[87] MÉRILLON, F., RÉVEILLÈRE, L., CONSEL, C., MARLET, R., AND MULLER, G. Devil: an IDL for hardware programming. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation* (2000), OSDI'09, pp. 17–30.

[88] MICROSOFT. The Importance of Implementing APIC-Based Interrupt Subsystems on Uniprocessor PCs. http://www.microsoft.com/whdc/archive/apic.mspx, January 2003.

[89] MICROSOFT. PCI multi-level rebalance in Windows Vista. http://www.microsoft.com/whdc/archive/multilevel-rebal.mspx, Nov. 2003.

[90] MICROSOFT. Firmware allocation of PCI device resources in Windows. http://www.microsoft.com/whdc/connect/PCI/pci-rsc.mspx, Oct. 2006.

[91] MICROSOFT. Windows API Reference. http://msdn.microsoft.com/en-us/library/aa383749%28v=vs.85%29.aspx, 2010.

[92] MICROSOFT. Microsoft Support. http://support.microsoft.com/, Oktober 2012.

[93] MICROSOFT. User-mode scheduling. http://msdn.microsoft.com/en-us/library/windows/desktop/dd627187%28v=vs.85%29.aspx, October 2012.

[94] MOCHEL, P. The `sysfs` filesystem. In *Proceedings of the 2005 Ottawa Linux Symposium* (2005).

[95] MOZILLA. Mozilla Support. http://support.mozilla.org/, Oktober 2012.

[96] NETRONOME SYSTEMS, INC. *Netronome Flow Engine, Model i-8000, Reference Guide.*, 2007.

[97] NIEDERLIŃSKI, A. *A Quick and Gentle Guide to Constraint Logic Programming* via *ECL^{i}PS^e*. Jacek Skalmierski Computer Studio, 2011.

[98] NIGHTINGALE, E. B., HODSON, O., MCILROY, R., HAWBLITZEL, C., AND HUNT, G. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the 22nd ACM Symposium on Operating System Principles* (2009), SOSP'09, pp. 221–234.

[99] NOLL, A., GAL, A., AND FRANZ, M. CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor. Tech. Rep. 06-17, School of Information and Computer Science, University of California, Irvine, Nov. 2006.

[100] NVIDIA. CUDA toolkit Documentation. http://docs.nvidia.com/cuda/index.html, Nov. 2012.

[101] OBJECT MANAGEMENT GROUP, INC. *CORBA 3.1 Specification*, Jan. 2008.

[102] OIKAWA, S., AND RAJKUMAR, R. Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior. In *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium* (1999), RTAS '99.

[103] ORACLE LABS. The Maxine Virtual Machine Project. http://labs.oracle.com/projects/maxine/.

[104] PCI-SIG. *PCI Express Base 2.1 Specification*, Mar. 2009. http://www.pcisig.com/.

[105] Peter, S. *Resource Management in a Multicore Operating System*. PhD thesis, Systems Group, Department of Computer Science, ETH Zürich, Sept. 2012.

[106] Peter, S., Schüpbach, A., Barham, P., Baumann, A., Isaacs, R., Harris, T., and Roscoe, T. Design principles for end-to-end multicore schedulers. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism* (Berkeley, CA, USA, 2010), HotPar'10, USENIX Association, pp. 10–10.

[107] Peter, S., Schüpbach, A., Menzi, D., and Roscoe, T. Early experience with the Barrelfish OS and the Single-Chip Cloud Computer. In *3rd Many-core Applications Research Community Symposium* (2011), MARC'11, pp. 35–39.

[108] Pugh, W. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM 33* (June 1990), 668–676.

[109] Rajkumar, R., Lee, C., Lehoczky, J., and Siewiorek, D. A resource allocation model for QoS management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium* (Dec. 1997), RTSS'97, pp. 298–307.

[110] Rajkumar, R., Lee, C., Lehoczky, J. P., and Siewiorek, D. P. Practical solutions for QoS-based resource allocation problems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium* (Dec. 1998), RTSS'98, pp. 296–306.

[111] Redis: An Open Source, Advanced Key-Value Store. http://redis.io/.

[112] Reed, D. P., and Kanodia, R. K. Synchronization with eventcounts and sequencers. *Commun. ACM 22*, 2 (Feb. 1979), 115–123.

[113] Roscoe, T. Hake. http://www.barrelfish.org/TN-003-Hake.pdf, April 2010.

[114] Roscoe, T. Mackerel. http://www.barrelfish.org/TN-002-Mackerel.pdf, December 2011.

[115] Rusling, D. A. The Linux kernel. http://tldp.org/LDP/tlk/tlk.html, 1999.

[116] Schüpbach, A., Baumann, A., Roscoe, T., and Peter, S. A declarative language approach to device configuration. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems* (2011), ASPLOS '11.

[117] Schüpbach, A., Baumann, A., Roscoe, T., and Peter, S. A declarative language approach to device configuration. *ACM Trans. Comput. Syst. 30*, 1 (Feb. 2012), 5:1–5:35.

[118] Schüpbach, A., Peter, S., Baumann, A., Roscoe, T., Barham, P., Harris, T., and Isaacs, R. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the 1st Workshop on Managed Multi-Core Systems* (June 2008), MMCS'08.

[119] Seward, J. *bzip2 and libbzip2, version 1.0.5, A program and library for data compression*, 2007.

[120] SICStus Prolog. http://www.sics.se/isl/sicstuswww/site/index.html, 2012.

[121] Simonis, H. Developing applications with eclipse. http://eclipseclp.org/doc/applications.pdf, 2012.

[122] Solomon, D. A., Russinovich, M. E., and Ionescu, A. *Windows Internals: Including Windows Server 2008 and Windows Vista*, 5th ed. Microsoft Press, 2009, ch. 4.

[123] Spear, M. F., Roeder, T., Hodson, O., Hunt, G. C., and Levi, S. Solving the starting problem: device drivers as self-describing artifacts. In *Proceedings of the EuroSys Conference* (2006), EuroSys'06, pp. 45–57.

[124] Stanitzki, C. AND: auto nice daemon. http://and.sourceforge.net/.

[125] Sun, E., Schaa, D., Bagley, R., Rubin, N., and Kaeli, D. Enabling task-level scheduling on heterogeneous platforms. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units* (2012), GPGPU-5, pp. 84–93.

[126] Sun Microsystems. Multithreading in the Solaris Operating Environment. White paper, http://www.sun.com/software/whitepapers/solaris9/multithread.pdf, 2002.

[127] Sun Microsystems. *SunPCi TM III 3.1 Users Guide*, July 2003.

[128] Sutter, H. The Free Lunch Is Over – A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's 30*, 3 (Mar. 2005).

[129] Sutter, H., and Larus, J. Software and the concurrency revolution. *Queue 3*, 7 (Sept. 2005), 54–62.

[130] SWI-Prolog. http://www.swi-prolog.org/, 2012.

[131] Thain, D., Tannenbaum, T., and Livny, M. Distributed computing in practice: the Condor experience. *Concurrency: Practice and Experience 17*, 2–4 (2005), 323–356.

[132] The OpenGroup. POSIX Specification. http://pubs.opengroup.org/onlinepubs/9699919799/, 2008.

[133] Thurlow, R. RPC: Remote procedure call protocol specification version 2. RFC 5531, Sun Microsystems, May 2009.

[134] TJworld. PCI dynamic resource allocation management. http://tjworld.net/wiki/Linux/PCIDynamicResourceAllocationManagement, June 2008.

[135] Torrellas, J., Tucker, A., and Gupta, A. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *J. Parallel Distrib. Comput. 24* (1995).

[136] TZENG, N.-F., AND ALLA, P. Guided shared trees for efficient multicast in large networks. In *Communications, 2003. ICC '03. IEEE International Conference on* (may 2003), vol. 1, pp. 87 – 92 vol.1.

[137] VAN MIEGHEM, P., HOOGHIEMSTRA, G., AND VAN DER HOFSTAD, R. On the efficiency of multicast. *IEEE/ACM Trans. Netw. 9*, 6 (Dec. 2001), 719–732.

[138] W3C. Resource description framework, Feb. 2004. http://www.w3.org/RDF.

[139] WEINSBERG, Y., DOLEV, D., ANKER, T., BEN-YEHUDA, M., AND WYCKOFF, P. Tapping into the fountain of CPUs: on operating system support for programmable devices. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008), pp. 179–188.

[140] WILLIAMSON, B. *Developing IP Multicast Networks*. Cisco Press, 1999.

[141] YIN, Q., CAPPOS, J., BAUMANN, A., AND ROSCOE, T. Dependable self-hosting distributed systems using constraints. In *Proceedings of the 4th Workshop on Hot Topics in System Dependability* (2008), HotDep'08.

[142] YIN, Q., SCHÜPBACH, A., CAPPOS, J., BAUMANN, A., AND ROSCOE, T. Rhizoma: a runtime for self-deploying, self-managing overlays. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware* (2009), Middleware'09, pp. 184–204.

[143] ZELLWEGER, G. Unifying Synchronization and Events in a Multicore Operating System. Master's thesis, ETH Zürich, Zürich, Switzerland, March 2012.

[144] ZELLWEGER, G., SCHÜPBACH, A., AND ROSCOE, T. Unifying Synchronization and Events in a Multicore OS. In *Proceedings of the 3rd AsiaPacific Workshop on Systems* (July 2012), ApSys'12, pp. 16:1–16:6.

[145] Zhou, J., Cieslewicz, J., Ross, K. A., and Shah, M. Improving database performance on simultaneous multithreading processors. In *Proceedings of the 31st international conference on Very large data bases* (2005), VLDB '05, VLDB Endowment, pp. 49–60.

[146] Ziakas, D., Baum, A., Maddox, R. A., and Safranek, R. J. Intel QuickPath Interconnect Architectural Features Supporting Scalable System Architectures. In *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects* (Washington, DC, USA, 2010), HOTI '10, IEEE Computer Society, pp. 1–6.

Curriculum vitae

Education

2007 - 2012	**Doctoral studies** Systems Group Department of Computer Science ETH Zurich
2001 - 2007	**Diploma studies in Computer Science** Department of Computer Science ETH Zurich
1994 - 2001	**High school** Gymnasium Typus C Bündner Kantonsschule Chur

Work experience

2007 - 2012	**Research assistant** Systems Group Department of Computer Science ETH Zurich
2004 - 2012	**Community service in computer science** Various projects Spital Davos
2006 - 2007	**Internship** Assentis Technologies AG

Teaching experience

2005 - 2012	**Teaching at ETH Zurich** Computer Architecture Computer Networks Computer Architecture and Systems Programming Advanced Operating Systems