Master Thesis

Message passing and bulk transport on heterogenous multiprocessors

Author(s):
Achermann, Reto

Publication Date:
2014

Permanent Link:

https://doi.org/10.3929/ethz-a-010262232 →

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 118

Systems Group, Department of Computer Science, ETH Zurich

Message Passing and Bulk Transport on Heterogeneous Multiprocessors

by

Reto Achermann

Supervised by

Prof. Timothy Roscoe, Dr. Kornilios Kourtis

October 17, 2014

inf | Informatik
Computer Science

Abstract

The decade of symmetric multiprocessors is about to end. Due to physical limitations, individual cores cannot be made any faster and adding cores won't work anymore (dark silicon). Operating systems face the next big paradigm shift: systems are getting faster by exploiting specialized hardware leading to the era of asymmetric processors and heterogeneous systems. An ever growing count and the diversity of cores yields new challenges regarding scheduling and resource management. Multiple physical address spaces within single machines let them appear as a cluster-like system.

Today's commodity operating systems are not well designed to deal with heterogeneity and asymmetric processors. Many attempts have been made to tackle the challenges of heterogeneous hardware, most of which treat available co-processors like devices or independent execution environments rather than equivalent processors. Thus, true support of a multi-architecture system is rarely seen, despite the fact that such hardware already exists. Examples are, systems like the OMAP44xx SoC or Intel's Xeon Phi co-processor.

In this thesis we will elaborate the arising challenges introduced by heterogeneous system architectures. We ported the Barrelfish research operating system to the Xeon Phi execution environment and explored the impact on performance and system design imposed by the emerging hardware characteristics such as different instruction set architectures, multiple physical address spaces and asymmetric cores. Based on our proposed software framework, we demonstrate a parallel, heterogeneous execution of an OpenMP program using our implementation of a message passing based OpenMP library and show why more threads do not necessary imply a better performance.

Acknowledgments

I want to thank Prof. Timothy Roscoe for giving me the opportunity to work with the new Xeon Phi hardware and for his inspirational thoughts and critical questions during our discussions; Dr. Kornilios Kourtis for his feedback on the benchmarks and Stefan Kaestle for his ideas about possible applications. Further, I'd like to thank Raphael Fuchs for the inspirational discussions about Barrelfish.

Contents

1 Introduction and Motivation **9**

2 Related Work **12**
2.1 Intel Manycore Platform Software Stack 12
2.2 Heterogeneous Systems 13
2.2.1 Heterogeneous Networked Environments 13
2.2.2 The CPU plus Accelerator Model 13
2.2.3 The System-on-a-Chip Model 13
2.2.4 The Heterogeneous Model 14
2.3 Barrelfish 15
2.4 Concluding Remarks 16

3 Barrelfish in a Nutshell **17**
3.1 Architectural Overview 17
3.1.1 CPU Driver 18
3.2 Capabilities 18
3.3 Major Domains 19
3.3.1 Kernel Monitor 19
3.3.2 System Knowledge Base (SKB) 19
3.3.3 Memory Service 19
3.3.4 Device Drivers 20
3.4 Message Passing 20
3.4.1 Exporting a Service 20
3.4.2 Binding to a Service 20
3.5 Hardware Register Access 21
3.6 Hake: Barrelfish's Build System 21
3.6.1 Hakefiles 21

4 The Intel Xeon Phi Co-Processor **22**
4.1 System Overview 23
4.2 Xeon Phi Co-Processor 23
4.2.1 Hardware Specification 24
4.2.2 In Comparison with the Xeon E5 v2 24
4.2.3 Comparison to General Purpose GPUs 25
4.3 Physical Address Space Layout 26
4.3.1 Host Side View 26
4.3.2 Xeon Phi Side View 27
4.3.3 Multiple Physical Address Spaces 27

4.3.4 Accessing Resources of another Processor 29
4.4 The K1OM Architecture 29
4.5 Booting the Xeon Phi 31
4.5.1 Operating Systems 31
4.5.2 Bootstrap 31
4.5.3 Boot Modes 32

5 Memory Access over PCI Express 33
5.1 Memory Hierarchy 33
5.1.1 Memory Access with Co-Processors 34
5.2 Evaluating Memory Access Performance 35
5.3 Memory Access Latency 35
5.3.1 Benchmark Description 35
5.3.2 Benchmark Results 36
5.3.3 Implications 38
5.4 Memory Throughput 39
5.4.1 Benchmark Description 39
5.4.2 Benchmark Baseline: Local Transfers 39
5.4.3 Host to Xeon Phi Transfers 41
5.4.4 Xeon Phi to Host Transfers 43
5.4.5 Between Xeon Phi 43
5.4.6 Implications 43
5.5 Message Passing over Shared Memory 44
5.5.1 Benchmark Description 44
5.5.2 Benchmark Baseline: Local Benchmark 44
5.5.3 Message Passing over PCI Express 46
5.5.4 Implications 48
5.6 A Word on CPU Performance 48

6 Barrelfish on the Xeon Phi 50
6.1 System Overview 50
6.1.1 Xeon Phi Driver 51
6.1.2 Xeon Phi Manager 52
6.1.3 Service and Domain Identification 53
6.2 Differences and Similarities to the MPSS 54
6.3 Modifications to Barrelfish 54
6.3.1 Boot-Loader 54
6.3.2 CPU Driver and Monitor 54
6.3.3 Kaluga and SKB 55
6.4 Creating the Xeon Phi Bootimage 56
6.4.1 Additional Hake Targets 56
6.4.2 Boot-Image Anatomy 56
6.4.3 Building Process 57
6.5 The Boot Process Explained 58
6.5.1 Boot Dependencies 58
6.5.2 Host Side Driver Startup 58
6.5.3 Chain of Boot Events 59
6.5.4 Weever: The Xeon Phi Boot-Loader 60
6.6 Capabilities Revisited 61
6.6.1 The Problem of Multiple Address Spaces 61

6.6.2 Capabilities: Accessing Remote Resources 62
6.6.3 Capability Translation: An Implementation 62
6.7 Flounder over PCI Express 65
6.7.1 Founder Bootstrap Steps 65
6.7.2 Message Stub Interface Extensions 65
6.7.3 Usage 66
6.7.4 Limitations 67
6.8 Xeon Phi Service Interfaces 68
6.8.1 Xeon Phi Driver Service 68
6.8.2 Name Services 71
6.9 DMA Service 75
6.9.1 DMA Library 75
6.9.2 DMA Manager 76
6.9.3 DMA Subsystem Programming Model 76
6.10 Virtual Devices 77
6.10.1 VirtIO 78
6.10.2 Barrelfish Bulk Transport 81
6.10.3 Virtual Devices: A Conclusion 82

7 Applications **83**
7.1 Work Offloading 83
7.1.1 Requirements for Offloading 84
7.2 OpenMP 85
7.2.1 Computation Model 85
7.2.2 BOMP: Barrelfish OpenMP 87
7.2.3 XOMP: OpenMP for Exclusive Address Spaces 87
7.2.4 Library Initialization 90
7.2.5 Library Scaling Characteristics 90
7.3 Use Case Example: Matrix Multiplication 95
7.3.1 General Benchmark Description 97
7.3.2 Baseline: Local Benchmarks 97
7.3.3 Effect of Data Replication 98
7.3.4 Matrix Multiply using Heterogeneous OpenMP 98
7.4 Conclusion 100

8 Future Work **102**
8.1 Towards One System 102
8.2 Xeon Phi as a Collection of Cores 104
8.3 Multiple Address Spaces 105
8.4 Barrelfish: Explore Scaling Behavior 105
8.5 Networking to the Xeon Phi and Host VFS 106
8.6 Extending Flounder 107
8.7 DMA Library 107
8.8 OpenMP 107
8.9 Bulk Transport over PCI Express 108
8.10 Xeon Phi Hardware Features 108
8.11 VirtIO 109

Appendices **110**

A Memory Access Benchmarks **111**
A.1 Xeon Phi Caches . . . 111
A.2 Memory Latency . . . 111
A.2.1 Experiment Description . . . 111
A.2.2 Experiment Parameters . . . 112
A.3 Memory Throughput . . . 113
A.3.1 Experiment Description . . . 113
A.3.2 Experiment Parameters . . . 113
A.4 Memcopy Closeup . . . 113
A.5 DMA Driver Overhead . . . 114
A.6 Message Passing . . . 115
A.6.1 Experiment Description . . . 115
A.6.2 Experiment Parameters . . . 115
A.6.3 UMP Latency Distribution . . . 116

B OpenMP Benchmark **118**
B.1 Library Initialization . . . 118
B.2 Matrix Multiplication . . . 119

C Barrelfish **121**
C.1 Memory Mapping in Barrelfish . . . 121
C.1.1 Benchmark Discussion . . . 121
C.1.2 Side Effects . . . 122
C.1.3 Conclusions . . . 122
C.2 Boot Performance . . . 122
C.3 Querying and Registering Symbols . . . 123
C.3.1 Interface Specification . . . 123

List of Figures

3.1 Barrelfish Architecture 18

4.1 System Overview with Xeon Phi 24
4.2 Address Space Layout of a System with Xeon Phi Co-Processors 26

5.1 The New Memory Hierarchy 34
5.2 Memory Access Latencies on Host (top) and Xeon Phi (bottom) 37
5.3 Memory Throughput on Local Memory 40
5.4 Memory Transfer from Host to Xeon Phi 42
5.5 Memory Transfer from Xeon Phi to Host 42
5.6 Memory Throughput Xeon Phi to Xeon Phi 44
5.7 Message Passing Latency 45
5.8 Message Passing Latency over PCI Express (Host) 46
5.9 Message Passing Latency over PCI Express (Xeon Phi) 47

6.1 Software System Overview 51
6.2 Transferring of Capabilities between Address Spaces 63
6.3 Xeon Phi Name Service 72
6.4 Emulating Virtual Devices on Co-Processors 78
6.5 VirtIO: Virtqueue Operation Scheme 79

7.1 Barrelfish OpenMP Library: Initialization Phase Spawn Time . . 92
7.2 Barrelfish OpenMP Library: Sharing Phase 94
7.3 Barrelfish OpenMP Library: Work Distribution 96
7.4 Local Matrix Multiplication using OpenMP 98
7.5 Matrix Multiplication on Heterogeneous System using OpenMP 99

A.1 Memcopy Throughput 114
A.2 DMA Driver Overhead 115
A.3 UMP Latency Distribution 117

C.1 Memory Mapping Performance with Multiple Domains 122
C.2 Initialization Time per Monitor 123

List of Tables

4.1 Xeon Phi Historic Overview 22
4.2 System Description 23
4.3 Xeon Phi Hardware Specification 25
4.4 Special Header Bytes 32

5.1 Memory Access Latencies 36
5.2 Local Message Passing Latencies (RTT) 45
5.3 Integer Performance in Millions of Operations per Second 48

6.1 Xeon Phi Domain ID Bit Representation 53

7.1 Barrelfish OpenMP Library: Work Distribution (Local) 95
7.2 OpenMP Matrix Multiplication Settings 97
7.3 OpenMP Matrix Multiplication Local Parallelization 97

A.1 Experiment Parameters: Memory Access Latency 113
A.2 Experiment Parameters: Memory Throughput 113
A.3 Experiment Parameters: UMP Latency 115

Chapter 1

Introduction and Motivation

Today's computer systems are getting increasingly complex. Gordon Moore in [54] formulated the prediction that the number of components per dense integrated circuit doubles approximately every two years. The ever increasing number of transistors on a chip together with an ever increasing clock frequency led to very complex and power hungry designs which certainly hit the so-called power wall [66].

Reaching the physical limits of heat dissipation, clock frequency could no longer be doubled every 18 month. As Moore's law still applies, the number of transistors per CPU keeps increasing whereas clock frequency stagnated or even decreased. To overcome the facing plateau of CPU performance chip designer embedded multiple cores onto a single die starting the multi-core era. This ended the *free lunch* [77], in other words, software was no longer getting faster automatically with every new generation: engineers needed to adapt for the new multi processing principles. Today's modern server processor have 10 cores or more.

The offered parallelism is even more increased by putting multiple CPUs in a single system resulting in a NUMAchine [33]. Besides the challenges of using parallelism provided by the cores, software engineers also needed to be aware of where to place the buffers to avoid over-saturation of the memory controllers and the inter-connect. Accessing memory is no longer uniform turning the machine into a distributed system [10].

Specialized hardware accelerators such as video cards were used from the early days on, executing certain task more efficient than general purpose CPUs. Accelerators evolved into sophisticated computation devices supporting a more general purpose compute model. With massive amounts of available cores, co-processors such as Intel's Xeon Phi or nVidia's Tesla provide specialized hardware features for highly parallel workloads in high performance computing.

Still, most of the computation is executed on general purpose processors. With an ever increasing number of transistors per chip and consequently a growing number of cores, the execution performance of scalar code is expected to stagnate. Single threaded programs do no longer run significantly faster with a new CPU generation and the effects are even amplified by the fact that not all code can be parallelized. Technologies such as Intel's TurboBoost [24] try to provide extra compute power for a single core when the other cores are idle. Based on the observation that all cores in today's CPUs are identical, we can infer that it is likely to hit another power wall. Hence, we are facing the end of the multiprocessor decade transitioning to the era of specialized hardware [29]. Offloading certain parts of a computation or even whole processes to co-processors can lead to an overall more efficient computation in terms of response time, throughput or energy consumption.

Today, the demand for energy efficient processors for mobile computing applications or data centers, buzzword *Green Computing* [35], drives the development of specialized hardware. The potential reduction of the overall energy consumption using specialized hardware enables prolonging battery life and reducing the operation costs of data centers.

To sum up, hardware is ever changing, offering new challenges to systems developers to make use of the specialized hardware devices. Further, with co-processors offering massive parallelism, a single machine tends to appear like a high performance cluster. Operating systems have to deal with the additional complexity in terms of number and features of cores, accelerators, co-processors and an overall heterogeneous system architecture. If various architectures are used concurrently, the OS has to be aware of which code can run in which execution environment and where it is expected to run most efficiently. Moreover, traditional abstractions are no longer appropriate for heterogeneous systems: multiple address space layouts, architectures and compute capabilities require new ideas in memory management and scheduling. Additionally, programmers must be aware of a changing memory hierarchy and its implications on execution performance.

This massive amount of parallelism and heterogeneity reveals new questions such as how to manage the resources, how to deal with the different architectures present and how to enable communication of processes running on different architectures?

This thesis will investigate the performance characteristics of the Intel Xeon Phi co-processor and list potential implications towards the design of a system. The proposed software system to manage the heterogeneity forms the basis to show various aspects of such a system and its challenges that need to be solved. Besides, we will present a way to parallelize a program and run across the heterogeneous architectures of a system with Intel Xeon Phi co-processors based on OpenMP.

Thesis Outline

We structure the thesis in eight chapters, starting off with a survey of related work (Chapter 2) followed by a brief description of Barrelfish the operating system used in this thesis (Chapter 3). The next two chapters describe the hardware features of the Xeon Phi (Chapter 4) and a basic performance evaluation of them (Chapter 5). Based on the insights of the performance characteristics, we present a software system for the Xeon Phi (Chapter 6) and potential applications and use cases (Chapter 7). We list discovered issues and possible anchor points for future work in Chapter 8.

Terminology

To clarify the technical terms used throughout this thesis, the following list defines how *we* use the terms to avoid misunderstandings.

- **Accelerator** Hardware device designed to execute a specific task, decoding audio streams for instance.
- **Co-Processor** Hardware device designed to execute a wide variety of tasks, a general purpose execution environment. For instance the Xeon Phi or general purpose GPUs. Not to be confused with co-processors like `CP15` that controls MMU.
- **Host** Execution environment which manages the possible co-processors. In our case `x86_64` architecture.
- **(Compute) Node** Either the Xeon Phi or the host. A system has a single host and may have multiple co-processor nodes.
- **Local Memory** Main memory of the node. On the Xeon Phi the GDDR, on the host the RAM.
- **Remote Memory** From the Xeon Phi point of view referring to the system memory or the GDDR of another Xeon Phi. Viewed from the host, the Xeon Phi aperture space.
- **Thread/Worker** In the OpenMP context, thread and worker (domains) can be used interchangeably.
- **Client Driver** Instance of a driver running on the co-processor connected to its counterpart on the host.

Chapter 2

Related Work

Heterogeneous system in a broader sense exists since the early days, ranging from specialized micro-controllers for digital signal processing to graphic accelerators (video cards). However, despite their heterogeneity, operating systems only deal with the symmetric host processors and uniform architectures leaving the management of co-processors to their device drivers.

In this chapter we present a survey of available heterogeneous hardware and software systems. We partition the discussion into Intel's software environment for the Xeon Phi (Section 2.1), an attempt to classify heterogeneous systems available (Section 2.2) and finally a Barrelfish related section (Section 2.3).

2.1 Intel Manycore Platform Software Stack

With the release of the Xeon Phi co-processors, Intel provided a software environment to support code offloading to the Xeon Phi in various modes such as Intel's MPI [18] architecture, native or offloaded execution [20]. Intel's Manycore Platform Software Stack (MPSS, [17]) supports Microsoft Windows [27] and Linux [63] hosts and includes a Linux-based co-processor operating system. We will go into more detail in Chapter 4 and Section 6.2.

The compute-model of Intel's MPSS treats each Xeon Phi in the machine as its own independent execution environment. From an operating system's perspective, the resulting design resembles a small cluster of independent networked machines rather than a single system. Communication between the nodes is made possible by a socket-like abstraction called Symmetric Communications InterFace (SCIF, [21]).

Even though the machine contains processors of different architectures, the operating systems do not have to deal with the heterogeneity. Similar to a networked cluster consisting of `x86_64` and `ARM` machines for instance, there is no management of multiple architectures by a single operating system. The Xeon Phi co-processor is treated as an ordinary PCI Express device.

2.2 Heterogeneous Systems

All of today's computers can be seen as heterogeneous. The degree of heterogeneity and how it appears highly depends on the system architecture and its intended use. The various designs can be classified into three models which can be combined.

2.2.1 Heterogeneous Networked Environments

Today's machines look like distributed systems [10]. However, a system is not limited to a single machine but can also be a cluster of networked nodes. Whereas a single machine is based on one architecture, the network has to deal with nodes of different types running a variety of operating systems and software. The resulting cluster is inherently heterogeneous. Several approaches to unify and manage networked resources have been investigated. The DAC Networked OS [31] provides resource sharing for a distributed co-operation between applications. 2K [47] deals with resource management in non-uniform, dynamic networks. However, networked environments do not deal with heterogeneity within a single machine.

2.2.2 The CPU plus Accelerator Model

This model is based on one or more main CPUs complemented with additional, specialized accelerators. One of the use cases for this model is to offload specific tasks to a hardware accelerator instead of doing the same computation with the general purpose CPU. This includes audio and video decoding using a DSP, cryptography applications or 3D imaging applications such as video games. Software frameworks like OpenGL [3] for instance provide the necessary abstractions. Accelerators are highly specialized hence not suitable for general purpose computing, especially as their output goes to one of the connectors. They are programmed by their respective device drivers.

Generalizing the accelerator model leads to a system consisting of a main CPU with programmable accelerators such FPGAs or core fusion [44] techniques. The resulting system is able to adapt to changing workloads. As the hardware features offered by the accelerator change depending on the programming, the hardware abstraction layer needs to be adapted as well. Nollet et al. [58] developed an operating system for reconfigurable SoC (OS4RS) based on a Linux real-time kernel. They are using a two level scheduling system where the top level manages task to processor assignment and per core lower level schedulers run the actual assigned task. Switching tasks between different core types can only be done at specific points during its execution.

2.2.3 The System-on-a-Chip Model

In today's mobile appliances, such as smartphones or tablets, the used CPUs are part of the so-called system-on-a-chip (SoC) architecture. One of the limiting factors for mobile applications is battery life and hence maximizing performance while minimizing the energy consumption are one of the driving factors. Similar

to the previous model, SoCs like the `OMAP44xx` [41] include specialized chips to do certain low-level tasks (energy) efficiently. As an illustrative example, one may consider playing an audio file with a dedicated audio DSP enabling the main CPU to go into a power save state.

Another aspect in the SoC scenario is to provide multiple cores with different performance characteristics while being based on the same architecture. Technologies such as ARM's *big-little* [50] are used to reduce the energy consumption of a chip. This setup can be viewed as pseudo heterogeneity where the architecture does not differ, but the compute-power and energy efficiency of the cores does. From an operating system's view switching between the different core types is like an extended change of the current C-state of the processor. The OS itself does not need to be heterogeneous. Replacing some bigger cores in a package with smaller ones can even be beneficial for certain workloads as shown by Kumar et al. [49] for Alpha processors and Li et al. [51] for Intel processors. The resulting chip is no longer symmetric. In both studies, they rely on a common instruction set architecture shared by all cores. This is comparable to some extent with the Xeon Phi as we will see in Chapter 4.

2.2.4 The Heterogeneous Model

The previous model can be seen as not fully heterogeneous: the secondary processors are either of the same architecture (*big-little*) or are too specific for general purpose computation (audio decoders).

Besides, the SoC model is also truly heterogeneous: the `OMAP44xx` [41], for instance, is based on two ARM Cortex A9 application processors which are accompanied by two ARM Cortex M3 multimedia processors. Both, the A9 and the M3 are general purpose CPUs but with different instruction set architectures (ISA) – in fact the ISA of the M3, Thumb 2, is also embedded in the ISA of the A9. Hence, an operating system utilizing both types needs to tackle the differences (see Section 2.3).

In the past, graphic processing units (GPUs) evolved into more general purpose compute platforms (GPGPUs). Streaming processors such as the nVidia Tesla (Section 4.2.3) are used for highly parallel workloads managed by software frameworks such as CUDA [60] or OpenCL [34]. Its general purpose design enables the use of GPGPUs in standard tasks such as sorting [32] or even networking [46].

In contrast to CPUs, the compute-model of GPGPUs is different and more challenging. Finding appropriate OS abstractions for GPGPUs is hard: Rossbach et al. [71] discovered the lack of proper OS abstractions for GPGPU resources such as fairness and isolation. They present *PTasks*, a parallel task API that simplifies the use of GPGPUs by abstractions to hide the orchestration as in CUDA [60]. With *PTasks*, the execution needs to be representable as a directed acyclic graph to match the streaming design of GPGPUs. Therefore, representing loops and conditionals is challenging and inefficient due to the warp scheduling of GPUs.

The question "*What is Heterogeneous System Architecture (HSA)?*" has also

been asked by AMD [4]. They discovered that main CPUs and accelerators/GPUs have driven apart. Their aim is to come up with a re-design of the system architecture to feature a more integrated platform. They classified different processors into *latency compute units* (CPUs) and *throughput compute units* (GPGPUs) [2, 55]. In the current design of a system, data has to be made available for the co-processor which involves copying. They claim that tighter integration of CPU and GPGPU reduces communication latency and lowers the programming barrier. This trend can be observed today: Intel integrated its graphics processors into the CPU. The proposed HSA framework unifies the address space of CPU and GPU and enables data access from anywhere. However, as we show in Chapter 5 just being able to access data from anywhere does not necessarily result in a fast system: smart buffer placement strategies which reflect localities are of great importance.

Similar to *PTasks* and AMD's HSA the latest OpenMP Standard [62] enables executing certain code blocks on a specific target device. In heterogeneous systems the code segment can be executed on a device which supports certain hardware features such as extra wide vector units. However, the support for these pragmas is not fully available: "*omp target constructs will always run on the host*", as stated by GCC [39].

Wang et al. developed EXOCHI [79] which supports work distribution among CPU and integrated graphic accelerators by extending the OpenMP pragmas to support heterogeneous multi-threading. The modified OpenMP library produces architecture specific code for the host CPU as well as the GPU in one binary. Wang et al. used the integrated graphics accelerator of the Intel CPUs for offloading the computation.

Nightingale et al. developed Helios [57] which supports heterogeneous systems by providing an independent interface to applications. The programming model of Helios uses an intermediate compilation step: similar to Java byte code, Helios programs are first translated into an intermediate language and then compiled to the target architectures present on the system upon installation. This eliminates the burden of providing different versions of the same executable from the programmer. Further, Helios distinguishes between coordinator and satellite kernels. As the underlying architectures of satellite kernels may differ, Helios does not support a process to span multiple satellite kernels.

With Popcorn, Barbalace et al. [5] tried to bring support for heterogeneous architectures to the Linux operating system. They added another abstraction layer on top of multiple Linux kernels which provide a unified view of the system (Single System Image, SSI). They use messages between the Linux kernels to maintain the global system state in the SSI. In contrast to virtual machines, their kernels form a peer-to-peer rather than a client/server relationship.

2.3 Barrelfish

With its multikernel architecture (see Chapter 3), Barrelfish naturally supports a heterogeneous system architecture – as long as the processor is capable of

booting an OS node. Barrelfish tackles different types of dissimilarities such as *non-uniformity*, *core diversity* and *system diversity* [74] based on system knowledge [73]. So far, two approaches have been done in the context of heterogeneous systems.

As we will see in Chapter 4, Intel's Single Chip Cloud Computer (SCC) is one of the precursors of the Xeon Phi. In [53] a Barrelfish port was made supporting a heterogeneous architecture consisting on a host (`x86_64`) and the SCC (`x86_32`) running at the same time. The sophisticated design of the SCC required the handling of corner cases which are no-longer needed with the Xeon Phi. Further, the architecture used on the Xeon Phi matches the host architecture close enough that some problems such as different data type sizes do not exist. Barrelfish has dropped the support of the SCC [48].

Recall the two different types of cores on the `OMAP44xx` SoC [41]. An approach to get Barrelfish run on both cores has been undertaken in [30]. However, the implementation is not fully finished and the Barrelfish kernels running the M3 cores are not aware of the A9 cores in the system. This results in essentially two systems running concurrently on the same SoC.

To have Barrelfish make use of GPGPUs as an OS node would require to have a kernel running on the GPGPU. With the different compute model and streaming processor design this is not achievable as also discovered in [71].

Baumann et al. [9] have ported Barrelifsh to the prototype of the Xeon Phi called Knights Ferry[1]. They came up with a software system, COSH, which provided bulk transport facilities in a heterogeneous system. As in the recent Barrelfish bulk-transport project [1, 11], Baumann et al. aimed to provide clear semantics of transfers and enforce access rights to buffers.

2.4 Concluding Remarks

In summary, in general heterogeneous systems are nothing new and, depending on the definition, have existed for a long time as the example of video accelerator showed. However, most of the available systems today are not truly heterogeneous: their operating system deals only with one architecture at a time or having device drivers control co-processors as is the case with Intel's MPSS or nVidia CUDA.

There is a clear trend towards more diverse and less coupled systems. At some point commodity operating systems will have to deal with this heterogeneous architectures as the main processors of computers evolve from symmetric multiprocessors into asymmetric designs which might not even share a common instruction set architecture.

[1] Due to the non-disclosure agreement the code is not available.

Chapter 3

Barrelfish in a Nutshell

In this chapter we will have a brief look at Barrelfish [68]: a research operating system developed at ETH Zurich with assistance from Microsoft Research [28]. The purpose of this Chapter is to set the foundation of the upcoming chapters by describing the most important building blocks and tools of Barrelfish. Readers that are already familiar with Barrelfish may skip this chapter. Starting with a general architectural overview (Section 3.1) and highlighting selected application domains (Section 3.3) we will have a look at resource management using capabilities (Section 3.2) and inter-domain communication using Flounder messages (Section 3.4). The remainder of this Chapter will talk about the domain specific languages for hardware access (Mackerel, Section 3.5) and for the build system (Hake, Section 3.6).

3.1 Architectural Overview

In contrast to the most common operating systems available, such as Linux [63] or Microsoft Windows [27] which are based on a monolithic or hybrid kernel, Barrelfish is based on a new kernel type called the multikernel [8]. An architecture scheme adapted to the Xeon Phi scenario can be seen in Figure 3.1.

Barrelfish tries to avoid sharing of state among cores, in other words, there are no global data structures. Each OS node has a local state replica which is kept up-to-date using message passing among the nodes. This enables lock-free, local access to system state. Barrelfish treats the machine as a distributed system [10]: Well-known agreement algorithms used in distributed systems ensure integrity of the state. As stated in [8], the fundamental model of a multikernel can be defined as:

1. All inter-core communication explicit.
2. Make OS structure hardware-neutral.
3. View state as replicated instead of shared.

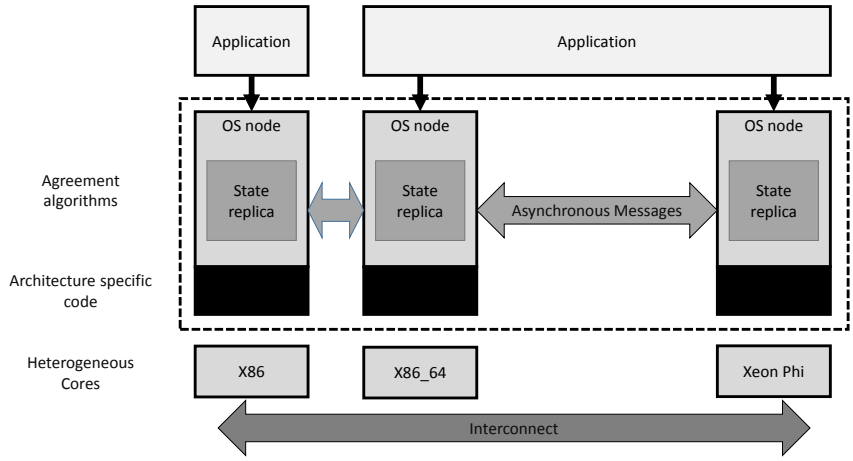


Figure 3.1: Barrelfish Architecture

3.1.1 CPU Driver

In Barrelfish the kernel is referred to as the *CPU driver*. It abstracts architecture specific hardware details and executes in privilege mode. The tasks of the CPU driver involve context switching, memory isolation, fast local message passing and capability management, to name a few.

The CPU driver exports an architecture independent set of services and mediates access to the local hardware features such as the memory management unit or the local APIC. Interrupts are forwarded to the user-space driver domains in form of a message.

3.2 Capabilities

Resource management is a key concern of every operating system. In Barrelfish resources are managed using capabilities. Each capability has a certain type and may refer to a region of memory, a communication endpoint or an operation (capability invocation). Each domain has its own capability tree (*CSPACE*) which stores all the capabilities the domain obtained throughout its runtime. As capabilities are critical, only the kernel can access the CSPACE and create or modify capabilities. The domains refer to a specific capability with a reference to it – the address of the capability in the CSPACE.

If a domain wants to execute a privileged operation, it needs to pass the capability reference. The reference is looked up in the CSPACE and its containing capability validated. For instance, mapping memory requires presenting a frame capability during invocation. Domains can grant access to their resources by passing one of their capabilities to another domain. This can be viewed as a copy operation of an entry in the capability tree to another CSPACE.

In addition to copy there are several other operations that can be invoked on a capability depending on the type. A frame capability for instance can be split into smaller frames or reduced to a read-only mappable frame, to name a few. A complete description of a capability system in a multikernel environment was discussed in [56]. The current implementation state of the capability system can be obtained in the technical note [76].

3.3 Major Domains

In Barrelfish applications and processes are called *domains*. As already explained in the previous sections, the kernel of Barrelfish provides only the necessary set of services which need to be run with privileges. This design choice pushes many of the traditional kernel services to user-space domains. In this section we will have a look at a selection of Barrelfish's core domains.

3.3.1 Kernel Monitor

The design of Barrelfish tries to move as much as possible into user-space domains: only when necessary the CPU driver executes certain supervisor tasks. However, some operations are considered to be privileged with respect to the system state but do not need to be executed in supervisor mode. An example of this would be handling the capability transfer between domains. This tasks are handled by the Monitor which is trusted by the CPU driver. Some kernel services can only be invoked by presenting the kernel capability which only Monitor has. Another task of the Monitor is to exchange messages with the other Monitors in the system to update the OS state.

3.3.2 System Knowledge Base (SKB)

The idea behind the System Knowledge Base (SKB) is to store facts about the system such as which services are running, the number of on-line cores or the discovered hardware devices. Facts can be specified using a declarative language approach [73]: either statically or upon events during boot such as hardware discovery for instance. Providing a service interface, domains can query the SKB to obtain information about the system. A constraint solver is used to answer complex questions such as "What is the optimal PCI configuration given the available devices?"

Another part of the SKB is Octopus [80] which tracks events happening in the system. Domains can publish events or subscribe to them. Octopus can also be used as a key-value store for exported services (name-service functionality) or barriers for thread/process synchronization using message passing.

3.3.3 Memory Service

As explained above, every physical resource is represented as a capability and so are frames of memory. The design of Barrelfish moves the management of physical memory into the user-space domain `memserv`. During boot up, `memserv`

is spawned by the `init` domain and initialized with the capabilities of all the unused frames. If a domain needs memory, it sends a request to the service and obtains a RAM capability which best suits the size and address constraints of the request. It retypes it into a frame or page table capability depending on the use.

3.3.4 Device Drivers

In contrast to other operating systems such as Linux where device drivers are part of the kernel, Barrelfish has its device drivers running in user-space. Each device driver is given access to its registers using a capability representing the MMIO range of the device. The capability is then mapped directly into the driver's virtual address space. This enables the driver to access the device without kernel involvement. Device drivers export a specified service interface to handle requests from other domains over message passing. Interrupts are transformed into messages by the kernel and handled by the user level driver.

3.4 Message Passing

Barrelfish's architecture demands efficient user-level message passing for communication between the domains especially replicating OS state. In Barrelfish this is abstracted using a tool called Flounder. The messaging interfaces are described in a domain specific language (DSL). The DSL compiler generates message passing stubs which are used by the domains to send and receive messages. Flounder supports multiple messaging backends. The following two subsections briefly describe the export and bind processes. For a more detailed description of the DSL and the operations refer to the technical note about inter-dispatcher communication [7].

3.4.1 Exporting a Service

A flounder channel is based on two endpoints: a service (exporting) side and a client (binding) side. Domains acting as servers initialize the service side of the interface by *exporting* it. Each exported Flounder interface can be identified by an IREF which is assigned by the Monitor during the export process. The kernel Monitor will store the IREF-to-endpoint mapping. To make the IREF known to other domains it can be registered with the name-service.

3.4.2 Binding to a Service

To use a service, a domain needs to *bind* to the exported IREF. The IREF can be looked up using the name service or passed by other means directly with program arguments for instance. Monitors keep a IREF to endpoint association which is used to bootstrap the communication channel. With the help of the Monitor a shared frame is initialized between the domains (In case of the user-level message passing backend).

3.5 Hardware Register Access

Device drivers need to be able to access the hardware registers of the device. Recall in Barrelfish resources are represented using capabilities and since memory mapped hardware registers are resources, a capability is needed to access them. Once the registers are made accessible to the driver domain, writing and reading specific bits is tedious and error prone. Barrelfish uses another domain specific language called Mackerel [67] to simplify reading and writing hardware registers or describing the layout of in-memory descriptors. The DSL specification is compiled into a C header file which provides the necessary functions to access the registers.

3.6 Hake: Barrelfish's Build System

With a broad diversity of supported architectures, it is necessary to have a suitable build system which can handle the different architectures, dependencies and domain configurations well. Barrelfish uses a tool called Hake [70] to work out which domains to build and how they are compiled.

To build Barrelfish, Hake is initialized in the build directory and configured with a one or more specified architectures. During the initialization, Hake will walk through the source tree and find all the Hakefiles and evaluates them.

3.6.1 Hakefiles

For the CPU driver and each domain, library or tool there is a Hakefile specifying how to build the target. Based on the Hakefiles the dependencies are generated and in the end a *Makefile* is generated.

The following code snipped shows an example Hakefile with the added dependencies of libraries, Flounder interfaces as well as Mackerel device specifications.

```
build application {
    target           = "device_driver",
    architectures    = [ "x86_64" ],
    cFiles           = [ "driver.c"  ],
    addLibraries     = libDeps [ "skb" ],
    flounderBindings = [ "device_driver_interface" ],
    mackerelDevices  = [ "device" ]
}
```

Listing 3.1: Sample Hakefile

Chapter 4

The Intel Xeon Phi Co-Processor

As with any new hardware, one generally wants to know its features, differences and similarities to other well-known products or architectures, especially `x86_64` in our case. In this chapter we will have a look at the first generation Intel Xeon Phi co-processor [23] (code-name Knights Corner, KNC). After providing historic information, we start this chapter with an overview of a possible system configuration (Section 4.1) and a section about the hardware features of the Xeon Phi (Section 4.2). Following, a section about the new `k1om` architecture with a brief comparison to `x86_64` (Section 4.4). The last part will investigate the boot process (Section 4.5) and its implications for the Barrelfish operating system.

Code Name	Year	Notes
TRP	2006	Tera-Scale Research processor
Larrabee	2007	Prototype (GPGPU)
SCC	2009	Prototype to promote Many Core Research
Knights Ferry	2010	Prototype of the Xeon Phi (MIC Prototype)
Knights Corner	2013	Xeon Phi Generation 1
Knights Landing	2015	Xeon Phi Generation 2

Table 4.1: Xeon Phi Historic Overview

History To set the historic context of the Xeon Phi, we crawl back in time and present the Intel Labs projects that lead to the current Xeon Phi product line. Table 4.1 summarizes the evolution of the Intel Many Integrated Core Architecture (MIC, [26]) micro architecture. In 2006 Intel demonstrated an 80 core Tera-Scale Research Processor (TRP) [38] which had limited features but served as a learning tool for the upcoming 48 core Single Chip Cloud Computer (SCC) [14]. While the TRP had only simple cores, the SCC came with a network of 48 fully functional Pentium based cores. With the Larrabee project [75], a general purpose graphics processor (GPGPU), Intel wanted to bring a new

graphics chip design based on many in-order `x86` cores. However, the Larrabee project was discontinued in 2010.

In 2010 Intel presented the Knights Ferry, the first prototype of the MIC architecture; in other words the prototype of the Xeon Phi co-processors, targeting high performance computing applications. The Knights Ferry can be viewed as a derivative of the previous many core projects Larrabee, TRP and SCC [15]. Evolving from the Knights Ferry, the Knights Corner features more cores, more memory and a peak performance of one teraFLOPS. The Knights Corner was released as the first generation of the new Xeon Phi brand and was made available to customers [23] in 2013. In 2014, Intel announced the second generation of the Xeon Phi, code-name Knights Landing, to be released in 2015 [22].

4.1 System Overview

Before we start with investigating the features of the Xeon Phi co-processor, we present a summary of the system we use throughout our evaluation. Table 4.2 below summarizes the most important numbers concerning the later evaluation of the proposed system. For a detailed description of the hardware refer to the respective hardware manufacturer information pages such as Intel ARK [25]. To illustrate the NUMA setup of the machine a schematic layout of the system can be seen on Figure 4.1 on page 24. Note that the I/O Acceleration Technology (I/OAT) DMA controllers are part of the CPU and described in the processor data sheet [43, Chapter 8.4] rather than residing on the chip-set, despite the fact that they show up as a PCI Express device.

Feature	**Value**
CPU	2x Intel(R) Xeon(R) CPU E5-2670 v2
	10 cores, Hyper-threading, 2.50 GHz
Architecture	Ivy Bridge
Memory	256 GB DDR3-1600
NUMA	2 NUMA Nodes
Xeon Phi	2x Xeon Phi 3210A
Network	Intel I350 Gigabit Ethernet
DMA	I/OAT Crystal Beach Version 3.0

Table 4.2: System Description

4.2 Xeon Phi Co-Processor

This section describes the hardware features of the Xeon Phi and compares them to other well-known architectures such as Intel Ivy Bridge or nVidia Tesla. The reader must be aware of the different models of the Xeon Phi which are currently available each of which vary in certain figures of their specification while others are consistent among the models. We restrict ourselves to the presentation of the Xeon Phi model 3210A which will be used in the later evaluation. For a

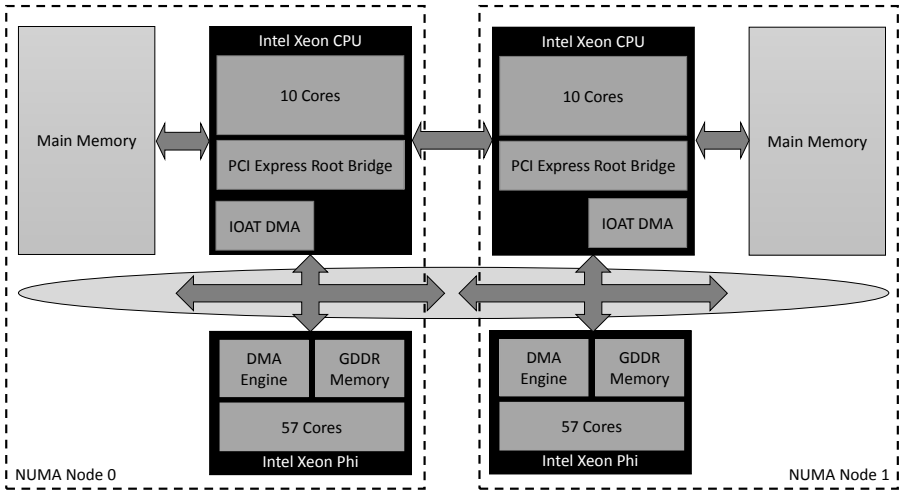


Figure 4.1: System Overview with Xeon Phi

complete of the available models comparison the reader may refer to the Xeon Phi data sheet [19] or Intel Ark [25].

4.2.1 Hardware Specification

The Xeon Phi is not an ordinary processor plugged into the mainboard socket but a co-processor which is connected over the PCI Express Bus [65]. Hence, it appears as a PCI device to the host operating system. Table 4.3 on page 25 shows a comparison of the Xeon Phi with the Xeon E5 v2 series and the current nVidia Tesla architecture. The values were extracted from their respective data sheets [42, 19, 59]. The following two subsections highlight a selection of differences between the Xeon Phi compared with the Xeon E5 v2 and nVidia Tesla respectively.

4.2.2 In Comparison with the Xeon E5 v2

Despite the fact that both architectures are based on variations of `x86`, there are several striking differences between the Xeon E5 v2 and the Xeon Phi: The Xeon E5 v2 is a multi-core CPU with 10 out-of-order cores and 2 hardware threads each whereas the Xeon Phi is a many-core CPU with 57 in-order cores and 4 hardware threads each. On the one hand, the Xeon E5 v2 uses a 3-level caching structure with a large L3 cache, on the other hand the Xeon Phi has a two level caching structure. Additional information about the Xeon Phi caches can be found in Section A.1. The next difference we want to emphasize is the type of supported main memory: the Xeon E5 v2 uses the DDR3 standard with various speeds ranging from DDR3-800 up to DDR3-1866, whereas the Xeon Phi has GDDR5 type memory built onto the co-processor card. Both types of memory are designed for special purposes and have different performance characteristics. As a rule of thumb, GDDR is designed to target high throughput streaming applications while DDR is adapted towards random access patterns

Feature	Xeon E5 2670 v2	Xeon Phi 3210A	nVidia Tesla Kepler K40
Cores	10	57	15
Type	Out-of-Order	In-Order	Streaming
Threads	20	228	2880
Clock Speed	2.5 GHz	1.1 Ghz	0.88 GHz
L1 Cache	32kB + 32kB	32kB + 32kB	16-48kB
L2 Cache	256kb (2.5MB)	512kB (28.5MB)	1.5MB
L3 Cache	25MB	-	-
Coherence Protocol	MOESI	MESI + GLOS	n/a
Extensions	Intel AVX	IMCI	CUDA
Memory (Max)	768GB DDR3	6GB GDDR5	12GB GDDR5
Memory Channels	4	12	n/a
Memory Bandwidth	59.7 GB/s	240 GB/s	288GB/s
Physical Address	46 Bit	40 Bit	n/a
PCI Express Rev	3.0	2.0	3.0
DMA Channels	8x (IOAT)	8x	2x
Peak Performance	0.46 Tflops	1.00 Tflops	1.43 Tflops

Table 4.3: Xeon Phi Hardware Specification

and low latency. The effects on memory access latency and throughput will be investigated in Chapter 5.

Last, one may notice the PCI Express revision on the comparison. As the Xeon E5 acts as the root complex of the PCI Bus, the Xeon Phi will be a PCI client in the system. It is possible to have up to 8 Xeon Phi co-processor in a single host system.

4.2.3 Comparison to General Purpose GPUs

General purpose GPUs (GPGPUs) are comparable to the Xeon Phi as both are co-processors with a huge number of cores and thus providing massive parallelism for high performance computing. Despite the closeness in terms of target applications, their differences are fundamental: The Xeon Phi offers `x86` based cores whereas GPGPUs such as nVidia Tesla are so-called streaming processors. This is reflected in their programming model. In order to program nVidia Tesla co-processors the CUDA ecosystem [60, 61] has to be used which provides a framework to offload computations onto the co-processor. In contrast to the mainly host driven CUDA, the Xeon Phi is capable of running an operating system such as Linux or Barrelfish providing autonomous services. Even though there would be many more things to discuss, a more in depth comparison between the Tesla and the Xeon Phi co-processors is out of scope for this thesis.

4.3 Physical Address Space Layout

In commodity systems, the physical address space layout is based on the assumption that there is an injective mapping between physical address and a byte in main memory. Looking at a system with one or more Xeon Phi co-processors, – note that this is not generalizable for all co-processors – one recognizes that this uniqueness assumption no longer holds. A scheme of the resulting address space layout can be seen on Figure 4.2 below. The following subsections explain the layout and its implications towards the understanding of physical memory.

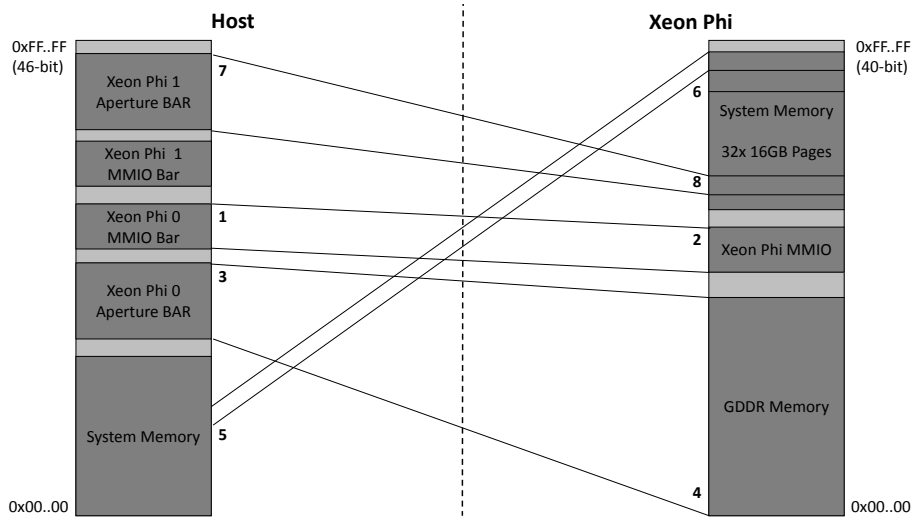


Figure 4.2: Address Space Layout of a System with Xeon Phi Co-Processors

4.3.1 Host Side View

At a first glance, there is not much different in the physical address range as seen from the host side: the Xeon Phi will appear as a normal PCI Express device with two memory regions which are of interest for us (numbers in parentheses refer to Figure 4.2):

- **MMIO Space** The memory mapped IO range (1) is 64 kB in size and consists of the device registers (2) to control the co-processor such as initiate the boot sequence or programming the DMA channels. It is important to note, that the access to the MMIO registers from the host side is limited.
- **Aperture Space** This memory region maps the Xeon Phi's GDDR memory (4) into the host physical address space (3). The size of this region varies between the models from 8GB to 16GB[1].

Two additional regions, the IO range and EXPROM aperture, need to be activated separately and are not of interest for our use case.

[1]The PCI aperture is rounded to the next power of two, the driver programmer has to be aware of the effective size depending on the model as the aperture may suggest a bigger size.

4.3.2 Xeon Phi Side View

On the Xeon Phi there are several memory regions available some of which are also accessible from the host.

- **GDDR Memory** This memory region (4) is the main memory of the Xeon Phi co-processor. An operating system running on the card will treat this memory region as normal RAM. The GDDR Memory region is also mapped as aperture (3) on the host.
- **MMIO** The memory mapped IO registers (2) are located in the so-called SBOX (PCI Express client logic) of the Xeon Phi. While access to the MMIO space is limited from the host, an operating system running on the co-processor can access all 666 registers.
- **System Memory** The top half of the physical address range on the Xeon Phi is called system memory range (6). The size of this region is 512GB. The system memory range is further divided into 32 pages of 16 GB each.

The physical address range of the Xeon Phi is only 40 bits in contrast to the 46 bits of the host (Sandy Bridge micro architecture)

System Memory Page Table

Each of the 32 system memory pages can be configured by writing the address into one of the consecutive system memory page table (SMPT) registers in the MMIO space. Each page can be programmed to map any 16 GB of host physical memory into the address space of the Xeon Phi. The memory block has to be 16GB aligned and must be exactly 16GB in size. Considering the Xeon Phi as a guest, one can draw analogies between configuring the SMPT and setting up of shadow page tables or extended page tables in a virtualized environment. Following we discuss the implications of the system memory range towards the understanding of physical address spaces.

4.3.3 Multiple Physical Address Spaces

In a system without co-processors there is a single physical address space to be managed by the operating system. Recall the assumptions made by the operating system about the uniqueness of references to an addressable byte in the physical address space. In the following we present the fundamental changes and their implications when dealing with a heterogeneous, Xeon Phi enhanced system.

Ambivalence of Physical Addresses

The uniqueness assumption that each byte in memory has exactly one physical address does not hold anymore. Consider the following example:

Let's assume we have a byte at address A in the main memory of the host. Viewed from the host side this address A is unique. Switching side, viewed

from the co-processor this byte in memory is now at address B resulting in two different addresses for the very same byte. To stress out the issue even more, the SMPT can be programmed in such a way that byte A now also appears at addresses C and D or it may not be existing at all.

In addition, let's assume there is an address G in every address space present in the system. Obviously, this address is valid in all address spaces but refers to another byte in memory or even a something different like a hardware register. Therefore, we have the same name pointing to different things.

With up to 8 Xeon Phi co-processors supported in a single system, there is the possibility that an address A in the host physical address space is mirrored in any of the co-processors address ranges. One may go even further and map the aperture spaces of the Xeon Phi in the SMPT. That way it is possible to have to distinct co-processor local addresses point to the same local address in GDDR memory.

Non-consistent Physical Addresses

Like the extended page tables in virtualized environments, the SMPT may be configured at any point of time. Hence, the physical addresses in the system memory range of the Xeon Phi behave more like virtual addresses rather than real physical addresses. The result of this is a broken assumption that a physical address space always refers to the same byte in memory – we assume that the PCI devices BARs are not altered once programmed.

Implications

Based on the above stated observations we claim that a single machine no longer has a single physical address space. We rather have a multiple address space scenario which are interleaved: system memory and co-processors GDDR can be accessible from both sides. A multiple address space scenario raises new questions and has several implications:

- **What is a physical address space?** This question is even more important when we consider methods to access remote memory directly such as RDMA, PCI aperture or system memory ranges.
- **How are bytes in memory referenced?** The design of Barrelfish allows that the system to span a heterogeneous setup such as ours with a host and Xeon Phi co-processors. Hence, it is possible to pass capabilities around which implies that we must make sure that address A (a byte in memory on the host) is translated into an address B on the Xeon Phi referring to the same byte in memory.
- **How are physical address spaces identified?** Having multiple physical address spaces accessible in a system means that there might be a physical address A which is valid in all address spaces, but refers to different regions in memory. Consequently, we need a way to uniquely refer to a specific address space. This is even more important with technologies such as RDMA where it is possible to access memory of a remote system.

- **Trust Issues:** The fact that the co-processor OS can access system memory may lead to unintended or malicious access of main memory[2]. Even with appropriate programming of the SMPT, the coarse granularity of the pages is not suitable for access control. In that case, the use of an IOMMU is highly encouraged.
- **Malicious co-processor OS:** Even though the host OS loads the co-processor OS onto the card, once it is running it is out of host's control. The host must trust the co-processor OS not to mess with host memory and vice versa. This can be reduced to a similar problem in virtual environments: the hypervisor needs to be trusted.

These implications are a research topic on their own and would be out of scope for this thesis. Some of the points stated above had to be solved during the development of the system and are described in Chapter 6.

4.3.4 Accessing Resources of another Processor

So far we have explained how the host can access the GDDR of the co-processor. However, it may be the case that multiple Xeon Phi co-processors are present in the system and they want to communicate with each other without host involvement. Therefore, it must be possible to access the GDDR of another Xeon Phi co-processor directly. This can be done by programming one entry of the SMTP with the host physical address of the aperture space of the other Xeon Phi.

This clearly adds the 16GB alignment constraints for the PCI aperture space when programming the PCI device BARs. The problem boils down to the Xeon Phi host side driver knowing where the other Xeon Phi's apertures are mapped and which slot to program. A possible solution to this will be presented in Chapter 6.

4.4 The K1OM Architecture

Intel introduced a new architecture called `k1om` which is an adaption of the well-known `x86_64` architecture. There are several differences between standard `x86` and the new `k1om` architecture which we will elaborate in this Section. So far, every new generation of processors was backwards compatible. This is not the case with the Xeon Phi. Most of the changes are considered to be a limitation of features on which out-of-the-box operating systems may rely. Hence, system software needs to be adapted to the new environment and the new instruction set architecture. The following paragraphs describe the major differences and its implications to the operating system intended to run on the Xeon Phi. For a complete list of limitations refer to the Xeon Phi Systems Software Developer Manual [20] or the Xeon Phi Instruction Set Architecture [16].

[2]It is generally assumed that the host trusts the co-processor not to mess with the host system. This assumption is justified by the fact that everything that is loaded on the co-processor somehow has to go through the host.

Compatibility Mode In general, processors supporting long mode have a compatibility sub mode which can handle 32-bit applications without recompilation. Even though the Xeon Phi is 64-bit, the compatibility sub mode is not available.

SIMD and Vector Instructions Intel equipped the Xeon Phi with extra 512-bit wide vector registers while truncating the support of the smaller MMX and SSE instructions (`xmm` and `ymm` registers). Only 512-bit registers (`zmm`) are supported, instructions using the smaller vector units trigger a unsupported instruction fault.

Local APIC With the increased number of cores and hardware threads, the APIC ID fields of standard `x86_64` are too small to reference all threads. Therefore, the APIC on the Xeon Phi does have extended APIC ID representations. Operating system must take the changed bit representation into consideration when booting new cores or issuing interrupts. Note that interrupts can not only target a local core but also a core on the host or another Xeon Phi card in the system.

Timer Hardware The commonly found timer hardware such as the programmable interrupt timer or the advanced configuration and power interface (ACPI) are not present on the Xeon Phi. This implies that the operating system must use the local APIC for timer interrupts and scheduling.

Global Page Tables The Xeon Phi does not support the global bit in the page directory entries or the page table entries. Attempts to enable it by writing the enable bit into CR4, will trigger a global protection fault.

Memory Prefetching To optimize certain memory operations in the standard C library, memory prefetch instructions are used. Normal prefetch operations are not available, but the Xeon Phi Instruction Set Architecture manual describes the new instructions that can be used.

Memory Ordering As the memory model of the Xeon Phi is stricter than the `x86_64` model, reads and writes to the memory appear in program order. This stricter model eliminated the need for memory ordering instructions such as MFENCE or SFENCE.

I/O Devices Similar to system on a chip designs (SoC) the Xeon Phi does not have a PCH south-bridge. Therefore, lots of devices that normally reside on the PCI bus are not present on the Xeon Phi. Without I/O devices present, the `in` and `out` instructions are not supported.

Serial Port Even though the Xeon Phi has a serial port, it is not directly usable. The port is located at the far end of the I2C bus and it is tedious to access. In addition one needs to connect additional hardware to the card in order to get the output. Thus, serial output needs to be handled otherwise.

4.5 Booting the Xeon Phi

The boot procedure of the Xeon Phi consists of several stages and boot modes. This section will briefly discuss the different boot modes (Section 4.5.3), supported operating systems (Section 4.5.1) and the booting process itself (Section 4.5.2). For a complete and more detailed boot process description one may refer to the Xeon Phi System Software Developers Guide [20].

4.5.1 Operating Systems

The limitations and differences to the `x86_64` architecture (as explained in Section 4.4) implies that operating systems will most likely not work out of the box. In the Intel provided software stack, a modified Linux is already included. In any case, after the required modifications are incorporated into the code, the Xeon Phi is capable of booting any operating system referred to as third party OS.

4.5.2 Bootstrap

In contrast to a normal `x86` system where s a BIOS handles the early boot stages, there is no BIOS on the Xeon Phi. The early boot tasks are handled by the so-called *bootstrap* which is executed on power-on and reset events. It consists of two different stages: `fboot0` and `fboot1`.

Stage 0: fboot0

After a reset or on power-on the `fboot0` code is executed. This block of code resides in the ROM of the Xeon Phi and cannot be changed once the co-processor is deployed. Hence, it acts as the root of trust for the further boot stages. Its task is to authenticate the integrity of `fboot1`: if the signature verification of `fboot1` succeeds the root of trust is transferred to `fboot1` code, otherwise the co-processor shuts down.

Stage 1: fboot1

In contrast to `fboot0`, the code of `fboot1` resides in flashable ROM and can be upgraded in the field. This stage executes early configuration routines and applies possible silicon workarounds. It waits for the *download ready* signal of the host and authenticates the image to be loaded. The root of trust is not passed any further if the image is a third party operating system, in other words, when the authentication process fails. Once `fboot1` has finished with initializing the co-processor hardware, it transitions the cores into 32-bit mode with paging disabled.

Stage 2: Linux Loader

The bootstrap calls the 32-bit entry point of the Linux kernel as documented in [64]. The image to be loaded must be in a binary format and certain bytes of its header must have a very specific value otherwise the boot will fail. These bytes are shown in Table 4.4 on page 32. They have been evaluated empirically by

Byte Offset	Value
0x202-0x205	Signature of the image, reads “HdrS”
0x1f1	size of the header in 512 byte sectors
0x1f4-0x1f7	size of the boot loader in bytes

Table 4.4: Special Header Bytes

zeroing the other bytes. In order to boot another third party operating system such as Barrelfish, a loader is used which resembles the early stages of the Linux boot process.

4.5.3 Boot Modes

The Xeon Phi can be booted in several modes. The chosen boot method affects the hardware state of the Xeon Phi after the boot procedure is completed.

Normal Mode In normal mode the Xeon Phi is capable of booting a 64-bit binary image such as the Linux bzImage [52] or directly a 64-bit ELF. When supplying a binary image, the boot process follows the Linux boot specification [64] and requires two images: one with the kernel and one of *initrd*. In this mode access to the hardware registers is restricted.

Maintenance Mode During the boot process the boot-loader will check the signature of the boot image. If the image is signed by Intel the card enters the maintenance mode. The card trusts the signed image and enables access to all the hardware registers.

Chapter 5

Memory Access over PCI Express

The most interesting aspects of every new system architecture are its base performance characteristics. This is especially true for heterogeneous systems such as ours (cf. Table 4.2). In the previous chapter we had a look at the hardware specifications of the host system and the Xeon Phi (Section 4.2). In this chapter we will first set the context of the new memory regions available on host and the Xeon Phi with respect to the memory hierarchy (Section 5.1). Based on this we will conduct initial benchmarks to assess its nominal performance characteristics: memory access latency (Section 5.3) and throughput (Section 5.4). Based on these observations we will investigate the optimal message buffer placement in Section 5.5. A detailed description of the used benchmark can be found in Appendix A.

5.1 Memory Hierarchy

Adding a co-processor with memory to a machine changes the memory hierarchy of the system. The purpose of this section is to relate the new layer in the hierarchy to its implications for a system or application programmer and the conducted benchmarks following later in this chapter. Recall the memory layout of the Xeon Phi (Section 4.3). Based on this memory layout we have created a graphical representation of the resulting memory hierarchy of the system which can be seen in Figure 5.1 on page 34.

Co-Processor View The different layers of the memory hierarchy basically resemble a traditional caching structure: the main memory (GDDR) of the co-processor can be seen as a cache for the system memory. One must be aware that there are no IO devices present on the Xeon Phi and hence the hard drive or network storage is limited.

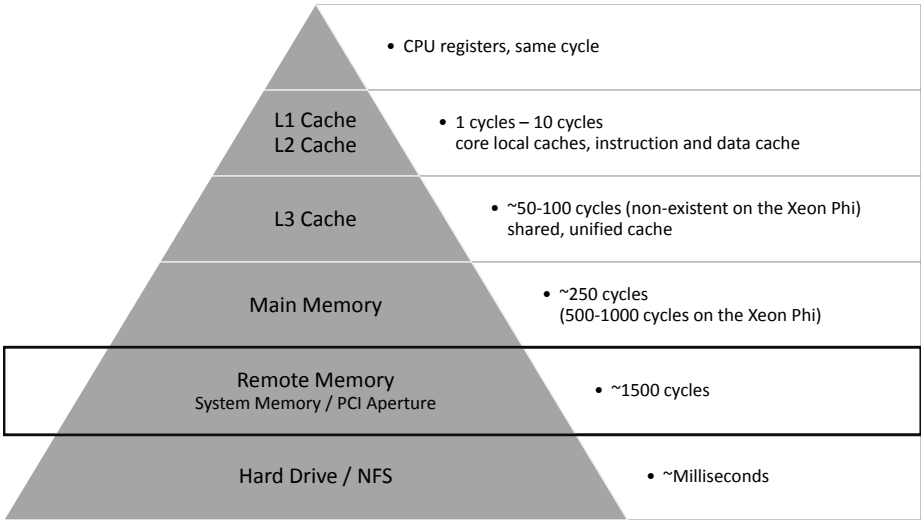


Figure 5.1: The New Memory Hierarchy

Host View Adding the highlighted layer has a different meaning on the host compared to the Xeon Phi. It is doubtable if the aperture space of the co-processor can be considered as a cache for the bottom layers: even though the performance characteristics may match caching purposes the intended use of this memory is different.

Accessing Remote Memory In contrast to other types of memory such as disk or network storage, remote memory appears as local RAM and provides the same ease of access. On has to be aware that access to the remote memory range will trigger a PCI Express transaction. For a domain in Barrelfish it is hard to tell whether the frame capability is referring to local or remote memory[1] and hence the programmer needs to be aware of potential performance penalties when accessing remote memory.

5.1.1 Memory Access with Co-Processors

The reasons for using co-processors in a system are versatile as we will see in Section 7.1. However, the compute-model of the available co-processors differ in various aspects. In any, case co-processors need to have access to the data they have to process. This can be achieved by two different models:

- **Push Based:** The host actively copies data onto the co-processor's memory.
- **Pull Based:** The copy process is initiated and executed by the co-processor on host request.

[1] It could be done by looking up the physical address of the capability and relate that to the system memory layout which would be to be stored in the SKB.

It depends on the co-processor which model is supported: while the Xeon Phi is capable of applying any of the two models other co-processors such as graphic cards may support only the push based model. In general, pull semantics are preferable as this frees up the host of the burden of copying and lets the co-processor decide which parts of the data to copy. Possible differences in copy performance will be evaluated in Section 5.4.

5.2 Evaluating Memory Access Performance

The performance of memory bound programs is limited by the accesses to data residing in memory. The key figures of reading or writing to any type of memory are its access latency and the offered throughput. Assigning appropriate weights to the two proprieties highly depends on the type of workload. This is also reflected in the type of the memory modules: DDR favors latency while GDDR provides better throughput.

Latency In a system which uses message passing over shared memory as communication between domains, memory latency is of high importance. The lower the latency, the faster a message sent is seen by the receiver. For every poll on a message slot, a memory read is issued.

Throughput On the other hand, programs may need to copy data from one location to another or to process large amounts of data in parallel. The performance of such an operation is affected by the memory throughput. Although, latency plays a role here as well, it can be hidden by pipelining multiple loads and stores.

Normalized Cycles

In all benchmark we used the local time stamp counter (TSC) to measure the time. In order to get a comparable value we divide the cycle count by the number of TSC-ticks per micro second. The obtained value represents absolute time and is referred to as *normalized cycles*.

5.3 Memory Access Latency

As explained above, memory latency affects the performance of message passing over shared memory which is widely used in Barrelfish. Therefore, knowing the raw access latencies to the different types of memory is of particular interest.

5.3.1 Benchmark Description

In the benchmark we used two different types of access patters in order to measure the latencies of the different types of memory:

1. **Randomized:** Traversing a randomly shuffled list of cache-line-sized elements with a working set size bigger than the available last level cache.

The purpose of this pattern is to maximize the cache miss likelihood while traversing the list to measure latency to main memory.

2. **Sequential:** Traversing an ordered linked list of pointer-sized elements with a working set size that fits easily into the L1 cache in order to measure cache hit latencies.

A full description of the benchmark and its settings can be found in Appendix Section A.2.

5.3.2 Benchmark Results

The results of the memory latency benchmark can be seen in Table 5.1 below and are visualized on Figure 5.2 on page 37. The times shown are the latencies in cycles to access a single element of the list. One must be aware of the non-comparability of the cycles as they are measured on two different CPUs with disparate clock rates. We differentiate the discussion of the results based on the type of the memory: local and remote memory.

Memory Type	**Cycles**
L1 Cache	4 (0.001)
Main Memory	295 (0.118)
Other NUMA	295 (0.118)
PCI Aperture	1545 (0.619)
PCI Aperture (C)	1401 (0.562)

(a) Host

Memory Type	**Cycles**
L1 Cache	3 (0.002)
GDDR Memory	547 (0.522)
System Memory	835 (0.798)
System Memory (C)	443 (0.423)
Other Xeon Phi	1355 (1.295)
Other Xeon Phi (C)	1109 (1.060)

(b) Xeon Phi

The values in brackets represent normalized cycles (cycles / ticks-per-microsecond). The standard deviation for all values was $\ll 1$ cycle. The addition (C) means cache friendly sequential access.

Table 5.1: Memory Access Latencies

Local Memory

In terms of local cycles the measured latencies for the L1 cache are in the same range for the host (Xeon E5 v2) as well as for the Xeon Phi with 4 and 3 cycles per element[2] respectively. Hence, accessing data residing in L1 cache has comparable latencies on host and the Xeon Phi co-processor even though in normalized cycles the latency on the Xeon Phi is about twice as high.

As expected when accessing main memory we experience an increased latency by two orders of magnitude. Whereas the L1 caches on the cores used the same SRAM technology resulting in similar access latencies, the used main memory technology differs between the host and the Xeon Phi: one clearly recognizes the contrasting characteristics of the used DDR3 and GDDR5 standards. Accessing main memory on the host is almost twice as fast as accessing main memory on

[2]Intel claims to have a memory access latency of the Xeon Phi L1 cache to be 1 cycle [20, 13] under the condition that the loaded value is used in integer operations.

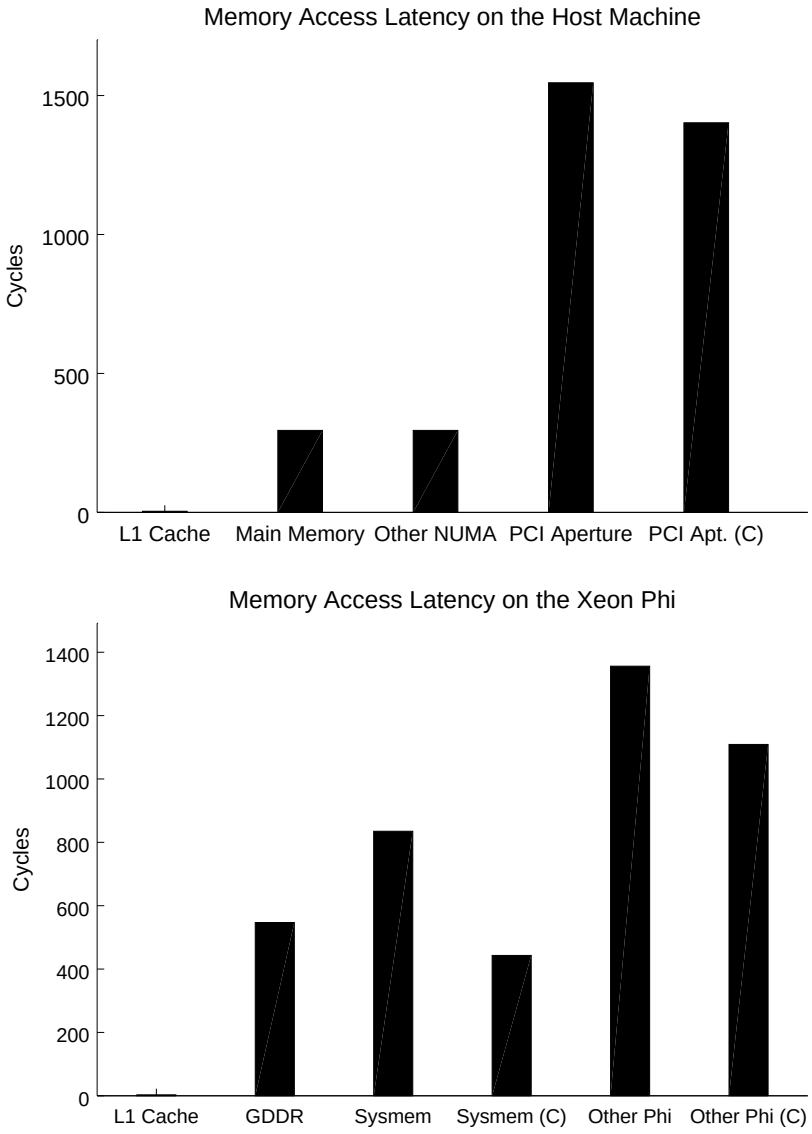


Figure 5.2: Memory Access Latencies on Host (top) and Xeon Phi (bottom)

the Xeon Phi[3] (in terms of raw cycles). The difference gets even more significant when comparing the normalized cycles: the cache miss penalty on the Xeon Phi is almost a factor of 4.5 higher than on the host.

Remote Memory

In our case accessing remote memory always involves crossing the PCI Express bus at some point of time. Thus, the access latency is dominated by the PCI Express latency which adds approximately 1000 cycles to local accesses on the host or 300 cycles on the Xeon Phi for the randomized access pattern. Accessing the PCI aperture from the host is an order of magnitude slower than main memory which is expected. The interesting case is the Xeon Phi where the overhead is less than a factor of two for accessing system memory. The resulting access latency is comparable on both nodes with 0.6 vs 0.8 normalized cycles.

When changing the access pattern to a cache friendly list traversal we see a slight decrease in latency on the host while almost cutting down the latency on the Xeon Phi by a factor of two. It is even more surprising that accessing (remote) system memory in a sequential pattern is faster than randomized access to local GDDR memory.

As explained, it is possible to access GDDR memory of another Xeon Phi. In our machine a read or write request to another Xeon Phi's memory needs to go from one PCI Express bus to another. By adding up the numbers for system memory access plus local GDDR access we get about the same value as for accessing to remote GDDR memory. In any case, compared to local GDDR access, the penalty for accessing remote memory is roughly a factor of two.

There is an interesting relationship between the inter-Xeon Phi memory latencies. The normalized time for accessing 2x PCI aperture space from the host is almost exactly the same as accessing GDDR memory of the other Xeon Phi: $2 \times 0.619 = 1.238 \approx 1.295$ normalized cycles.

5.3.3 Implications

In general organizing its own hot set of data in such a way that it fits in the L1 cache is a good practice. On systems with GDDR-based main memory it is even more important as there is a huge penalty for every cache miss. In addition, the use of prefetch instructions is encouraged if the access pattern can be predetermined. In contrast to the Xeon Phi, the host needs to pay attention of data locality as accessing PCI aperture space involves a high penalty in terms of latency.

Relation to Hardware Features

With a high access latency to GDDR memory and relatively small caches, the CPU of the Xeon Phi has to wait for data for several hundred cycles. This waste of cycles is even more increased by the in-order execution. However, having four

[3]Intel specifies [13] the GDDR latency to be 500-1000 cycles which supports our measured value of 547 cycles.

hardware threads per core, the Xeon Phi CPUs are able to execute instructions coming from another instruction stream while waiting for data to be ready.

5.4 Memory Throughput

In this section we explore the data transfer capabilities of the Xeon Phi using different directions of data flows and compare the various options of data transfer techniques. Memory throughput is important because we want to get the hot working set of data closer to the accessing node to achieve a low latency as shown in the previous benchmark.

5.4.1 Benchmark Description

For each of the different transfer types the task was essentially to copy a buffer from one location to another. The copy is done in a single threaded fashion using either a `memcpy()` or a single DMA transfer. The rationale behind this simplification is that buffers are usually copied using a single `memcpy()` operation. Therefore, the presented values can be considered to form a lower bound of the copy operation as parallel copies and transfers can be used. For each direction we investigate the behavior for different buffer sizes. A full description of the Benchmark can be found in Appendix A.

DMA Measurements DMA measurements are taken directly at the driver. The use of a DMA service will introduce an additional overhead which is briefly discussed in Appendix A.

NUMA The shown results are only within the same NUMA region. Performance differences when transferring between NUMA nodes can be found in Appendix A.

Values The shown values are always the median with the standard error based on an experiment repetition of 500 rounds.

5.4.2 Benchmark Baseline: Local Transfers

We take local memory to memory transfer as the baseline for the later benchmarks over PCI Express. The rationale is that local transfers are faster than going over the PCI bus. On the host we use the IOAT DMA engine and on the Xeon Phi the co-processor local DMA engine to do memory to memory copy.

Benchmark Results

The measured throughput can be seen on Figure 5.3 on page 40. The x-axis shows the relevant sizes with respect to the cache hierarchy. On both nodes DMA outruns `memcpy()` significantly after the setup latency is dominated by the transfer time. It is beneficial to start a DMA transfer for buffers larger than 16 kB. The benchmark results reveal interesting behaviors which will be addressed in the following paragraphs.

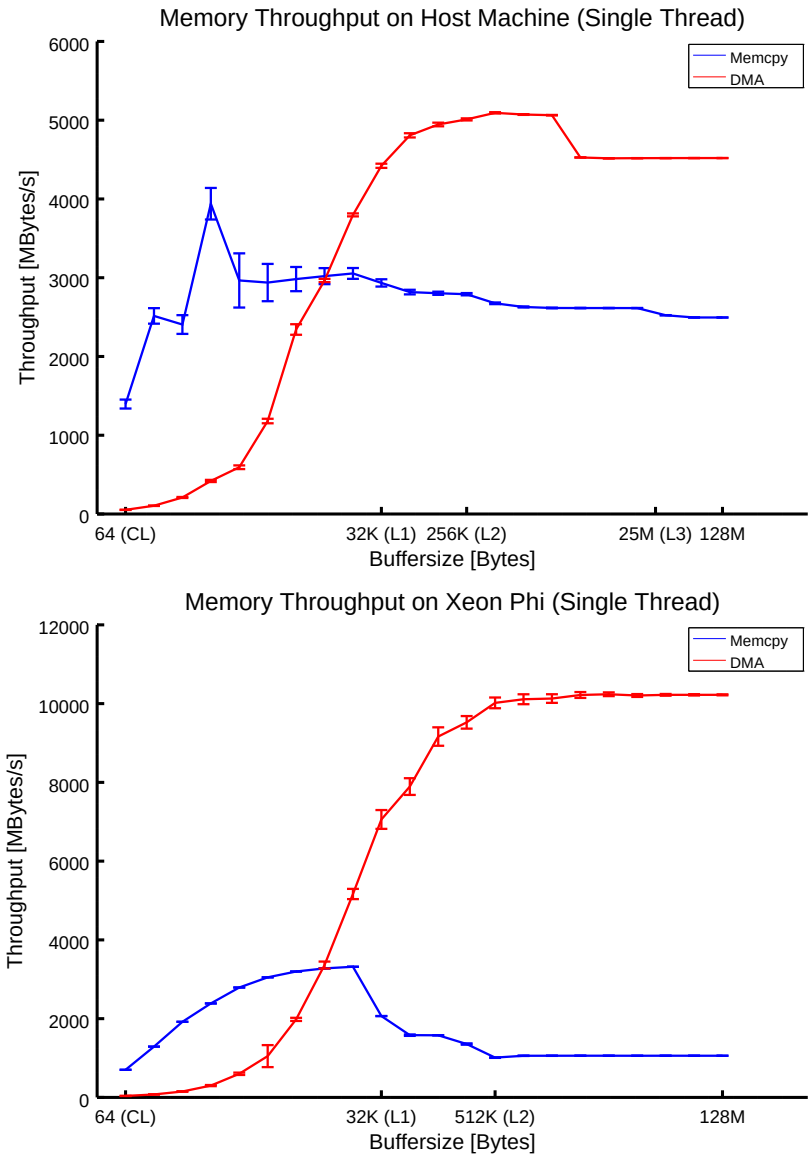


Figure 5.3: Memory Throughput on Local Memory

Memcpy Copying a buffer using `libc`'s `memcpy()` on the host showed quite some odd behavior for small buffer sizes up to 1kB with a relatively high standard error[4]. This may be caused by caching anomalies as the buffer lines fall into the same set. With increasing buffer sizes the available cache is used more evenly resulting in a more stable performance. There are few drops to see, each at the boundary of the three cache levels (32kB, 256kB and 25MB).

On the Xeon Phi there are no such anomalies as on the host. However, when

[4]The results were consistent throughout repetition of the entire experiment runs.

the size of the L1 cache is reached a vast drop in the throughput can be seen. This can partially be attributed to the lack of prefetch instructions in the used C standard library for the Xeon Phi. There is another drop when the size of the L2 cache is reached. A similar behavior to the host only with a higher impact. These reductions in throughput are also caused by the high latency of the GDDR memory.

A Word on Prefetching Both experiments were using the `memcpy()` function of `newlibc` [36] which is hand-optimized assembly. The used prefetch instructions are not valid on `k1om` and hence are deactivated on the Xeon Phi.

DMA Transfers On both nodes DMA transfer suffer from a longer latency for issuing transfers than `memcpy()` which is compensated by a higher throughput. The peak throughput is reached when there is at least one full descriptor used – 512kB on the Xeon Phi and 1MB on the host. Comparing the peak transfer speeds the Xeon Phi does about a factor of 2 better than the host. This can be accounted to the GDDR memory the Xeon Phi uses. If there are more than two descriptors used on the host, there is a drop in the DMA throughput which may be partially attributed to an increased setup time.

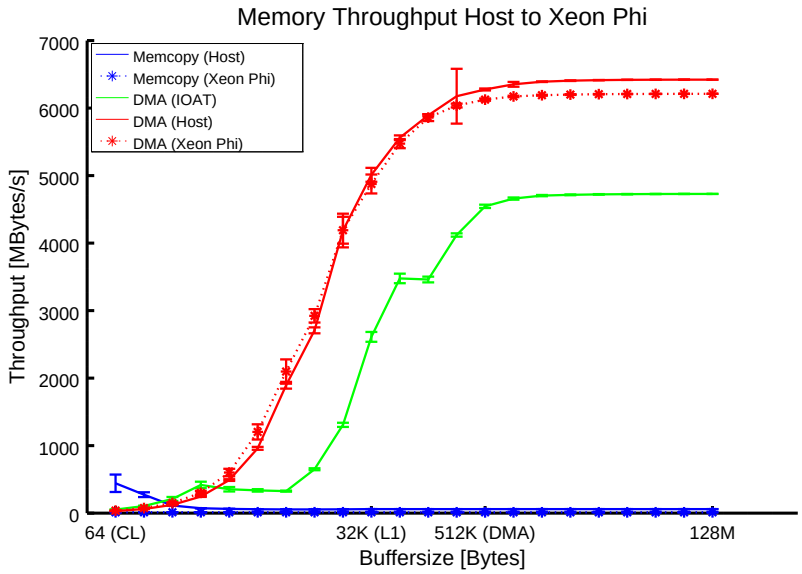
5.4.3 Host to Xeon Phi Transfers

Transferring data from the host to the Xeon Phi is crucial for offloading tasks. We repeated the exact same experiment as before but replace the target address with a region of memory residing on the PCI aperture space of the Xeon Phi. The experiment is conducted on the host as well as on the Xeon Phi to catch potential differences. The results can be seen in Figure 5.4 on page 42. In order to show the differences of `memcpy()` with a higher resolution a scaled up graph can be found in Appendix A. When going over PCI Express there is a huge difference between `memcpy()` and DMA transfer: issuing a DMA request is faster after a few hundred bytes.

I/OAT DMA Engine versus Xeon Phi DMA Engine

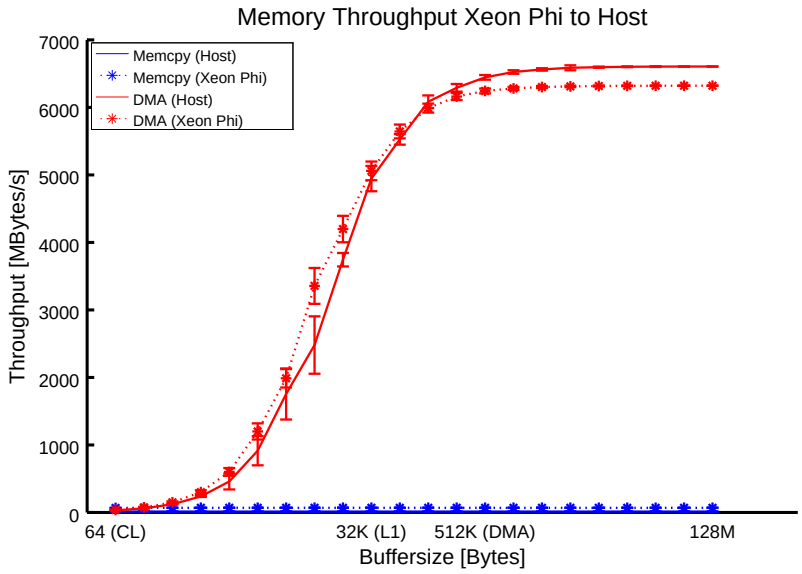
If we compare the performance of the I/OAT DMA engine (green) with the baseline we see that we reach about the same peak throughput. However, compared to the Xeon Phi DMA engine (red) there is a significant performance difference. This can be explained by the maximum transfer size of a PCI Express transaction. In contrast to the 128 bytes of the I/OAT DMA engine, the Xeon Phi DMA engine supports up to 256 bytes per PCI Express message. In addition, the IOAT DMA issues a memory write to the co-processor GDDR while the Xeon Phi DMA basically does a read – we will see performance differences of reads and writes later.

Comparing DMA transfers controlled by the host to the ones managed by the Xeon Phi, we see that there are no statistically significant differences (both red lines follow each other closely). The DMA transfers issued by the host seem to have a marginally higher throughput at bigger transfer sizes. This is also backed by the Systems Developer Manual [20] which states that the DMA engine delivers a slightly better performance when operated from host side.



The note in bracket specifies the origin of transfer.

Figure 5.4: Memory Transfer from Host to Xeon Phi



The note in bracket specifies the origin of transfer

Figure 5.5: Memory Transfer from Xeon Phi to Host

5.4.4 Xeon Phi to Host Transfers

The results from the offloaded computation may be transferred back to the host. Accordingly, we repeat the previous experiment with opposite data flow direction. The results can be seen on Figure 5.5 on page 42. The reader may have noticed the lack of the I/OAT DMA engine. This is because DMA transfers with a MMIO region as source are not supported by the I/OAT DMA engine. We obtain a similar picture as before showing the extreme benefits of doing a DMA transfer instead of `memcpy()`. Comparing to the opposite data flow direction there is almost no difference: the peak throughput is in the same 6.5GB/s range. Furthermore, DMA transfers issued by the host are a bit faster than the ones controlled by the Xeon Phi.

5.4.5 Between Xeon Phi

Some work flow may need to distribute data among the Xeon Phi. The questions that arises is if doing a direct co-processor to co-processor transfer is better compared to going through host memory. Transfers between two Xeon Phis can be classified into a *read* (copy data from the remote GDDR into the local GDDR) and a *write* (copy data from local GDDR to the remote GDDR). Certainly, we expect similar performance patterns like the ones we obtained when transferring data between host and Xeon Phi. However, the observed throughputs are much slower compared to host—Xeon Phi transfers by up to an order of magnitude. Moreover, the transfers are skewed, in other words, reading is faster than writing by a factor of 3x.

The question stated above has to be answered with *it depends*. On the one hand, going directly from Xeon Phi to Xeon Phi is expected to be slower than going through main memory of the host. On the other hand, it does not use resources from the host.

5.4.6 Implications

The results obtained from the memory throughput benchmarks have direct implications for potential bulk transfer solutions. Any reasonable bulk transfer implementation which supports transfers between host and Xeon Phi must incorporate DMA services. This not only frees up CPU resources and reduces pipeline stalls but also executes the transfer considerably faster despite the additional message passing overhead (Section A.5).

On local transfers doing a `memcpy` makes sense up to a buffer size which matches the size of the L1 cache. In any case, copying data locally should be avoided as possible and may be better done using lazy techniques such as copy-on-write.

Furthermore, to transfer data from and to the Xeon Phi GDDR, one should use the DMA engine of the Xeon Phi rather than the I/OAT Crystal Beach DMA engine. Despite the difference in throughput, DMA transfers can be initiated by both sides. Judging from the inter-Xeon Phi transfer performance, the use of such transfers is discouraged.

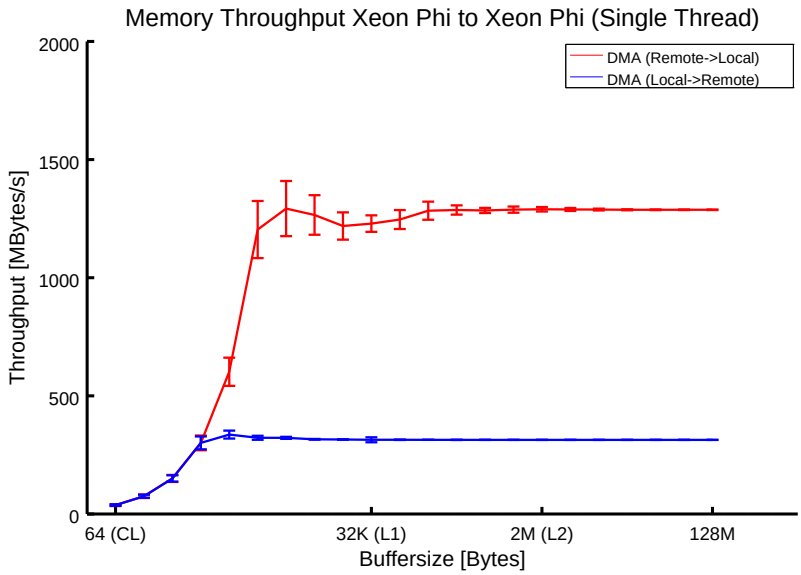


Figure 5.6: Memory Throughput Xeon Phi to Xeon Phi

5.5 Message Passing over Shared Memory

With a multikernel design many traditional system services such as memory management or device drivers are pushed to user-space. Therefore, an efficient method of communicating between domains is of the high importance. Recall, Barrelfish supports multiple backends for establishing channels between domains which are abstracted by Flounder [7] . For our application in mind, user-level message passing (UMP) is the backend of interest. The goal of this benchmark is to investigate the raw UMP latency when sending a message over PCI Express compared to node-local communication. In particular the effects of receive and transmit buffer placements.

5.5.1 Benchmark Description

In the following benchmarks we measure the round-trip time (RTT) of a zero byte payload UMP message. We use the bare UMP backend in order to get a lower bound on the performance. The setup is based on the already existing `ump_latency` benchmark: of two domains which bootstrap a UMP channel over a shared frame. A master domain starts sending messages to an echo server which basically just replies with a message.

5.5.2 Benchmark Baseline: Local Benchmark

Similar to the memory throughput benchmark we use a local setup as a baseline for message passing over PCI Express. For this purpose we run the `ump_latency` benchmark on the host and the Xeon Phi. The measured round trip times can be seen on Table 5.2 below and they are visualized in Figure 5.7 (page 45).

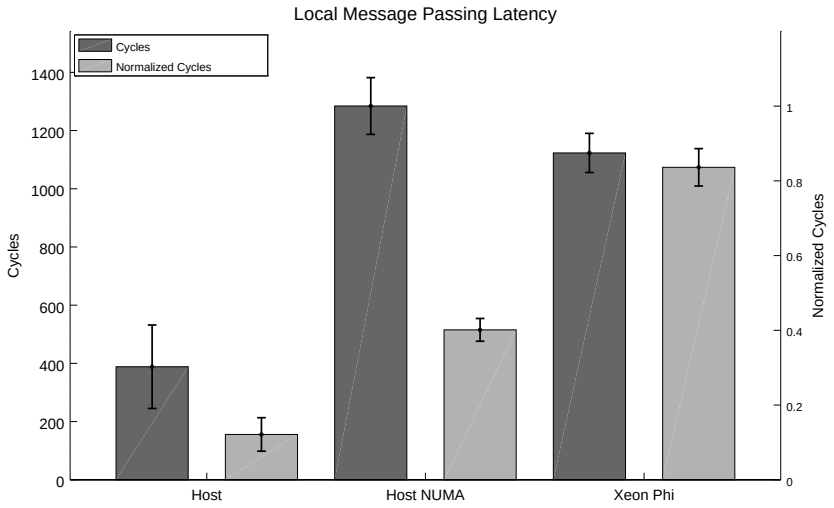


Figure 5.7: Message Passing Latency

The results show that a round trip between two domains on the Xeon Phi is about the same as between the two NUMA nodes on the host (in terms or local cycles). Considering normalized cycles the Xeon Phi has a RTT which is roughly an order of magnitude slower than on the host. The following two paragraphs relate the measured round trip times to the raw memory latencies of Section 5.3.

Location	**Min**	**Median**	**99-Percentile**
Host (within NUMA)	323 (129)	388 (155)	608 (244)
Host (inter NUMA)	895 (359)	1284 (515)	1906 (764)
Xeon Phi	842 (805)	1133 (1083)	1406 (1344)

Values in local cycles. In brackets, normalized cycles.

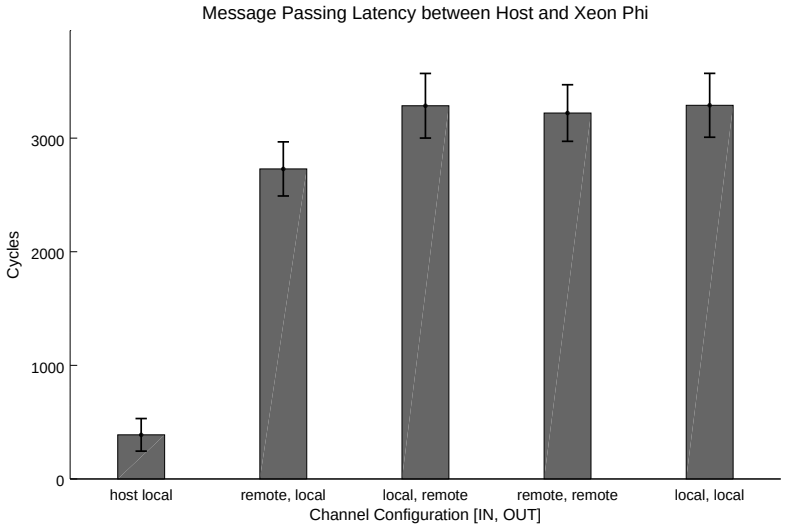
Table 5.2: Local Message Passing Latencies (RTT)

Host: On the host the worst case scenario happens when both endpoints observe a cache miss resulting in $2 \cdot 295$ cycles ≈ 600 cycles which is about the 99-percentile value (608 cycles) observed in the experiment. In the best case, the cache line is present on both cores and the cache-coherency protocol handles the transfer: the cache line is loaded directly from the other core's cache instead of main memory. This is almost twice as fast as going to main memory.

Xeon Phi: Recall MESI+GLOS, the special cache coherency protocol of the Xeon Phi. With no owner state, when the message is sent, the cache line on the other cores must be invalidated first and re-fetched from main memory later on – in contrast the MOESI protocol supporting an owner state enables broadcast of modifications to the other cores without the need of explicit cache-line invalidation.

Therefore, a higher latency is expected which may even be more amplified by the ring interconnect of the Xeon Phi cores. Intel claims in [20] that the MESI+GLOS protocol can be used to emulate the missing owned state to achieve the benefits of a full MOESI protocol. Compared to the host where the median is significantly below the double memory latency on the Xeon Phi we get about twice the memory latency: $2 \cdot 547$ cycles ≈ 1100 cycles. With the GLOS addition, in the best case we achieve a latency which is only 1.6 x GDDR memory latency (compared to the MOESI protocol of the host, where we got a factor of 1.1 for the round trip). To sum up, there is an improvement over the standard MESI protocol but in most of the cases the latency is more than two cache misses.

RTT per Core Distribution The `ump_latency` benchmark runs once between each core. The distribution of RTT per core on the host and the Xeon Phi can be found in Appendix A.

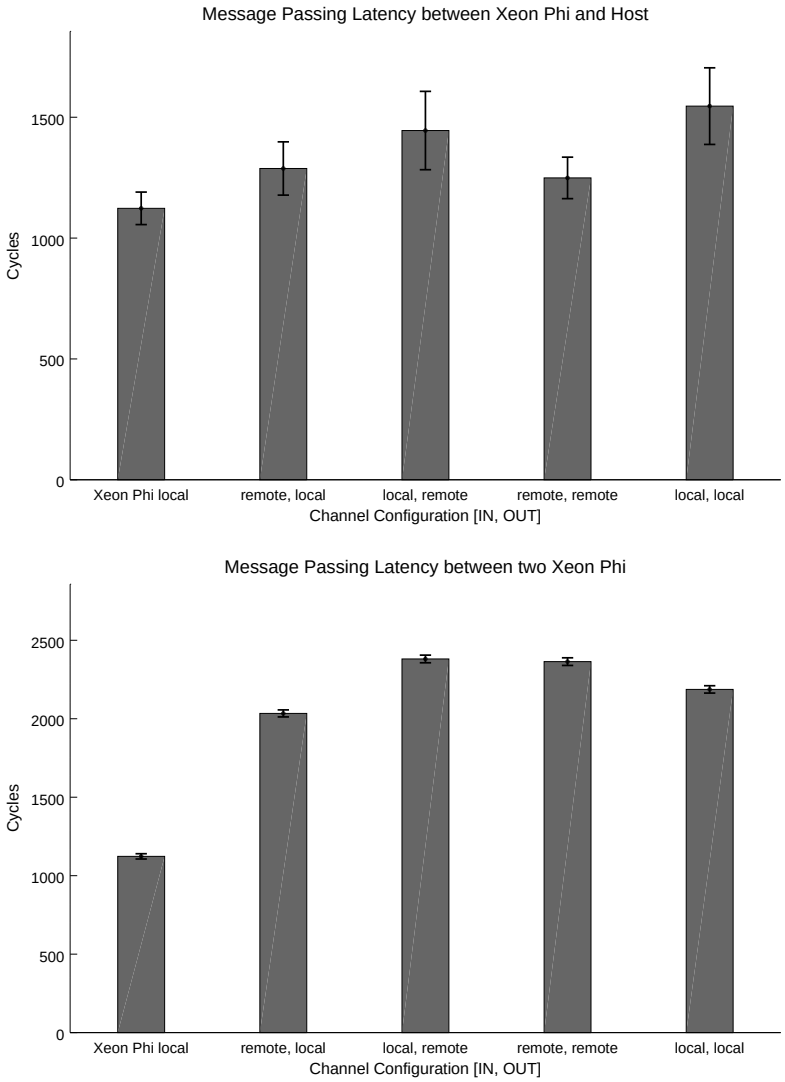


Values in local cycles, errorbars show standard deviation. The bar description defines the buffer configuration for the receive buffer and transmit buffer respectively.

Figure 5.8: Message Passing Latency over PCI Express (Host)

5.5.3 Message Passing over PCI Express

The next step is to elaborate on the performance characteristics when we exchange messages between domains running on different nodes and hence need to cross PCI Express. Obviously, based on the data gathered for raw memory latency the results cannot be faster than our baseline. The goal is more understand the effects of different buffer placement strategies and likewise estimating the resulting overhead when going over PCI Express. Each UMP channel has a receive and a transmit buffer which can reside either in local or remote memory. The results compared to the baseline can be seen in Figure 5.8 and Figure 5.9 on page 47.



Values in local cycles, errorbars show standard deviation. The bar description defines the buffer configuration for the receive buffer and transmit buffer respectively.

Figure 5.9: Message Passing Latency over PCI Express (Xeon Phi)

Host to Xeon Phi

Compared to the host's local round trip times, sending messages over PCI Express bus is about an order of magnitude slower than within the same NUMA node (or a factor of 3x compared to inter NUMA messages). Analyzing the four possible buffer configurations we see that polling a remote receive-buffer and writing messages to a local transmit-buffer offers the best latencies. There is not enough statistical confidence to make a statement about the ranking of the other three configurations.

Xeon Phi to Host

We repeat the experiment but initiate the messages from the Xeon Phi side. The results shown the top graph of Figure 5.9 reveal completely different, yet surprising patterns as on the host: even though Xeon Phi local latencies are significantly faster than going to the host and back, the overhead for crossing the PCI Express bus is rather low. This can be accounted to the high latency of the GDDR memory and the lack of a proper MOESI cache-coherence protocol. We cannot select one buffer configuration as the best with statistical confidence. However, if we chose to poll a remote receive buffer we are on the faster side while the location of the transmit-buffer does not matter. This is because the remote buffer may reside in the host CPU cache which can be snooped by PCI Express.

Xeon Phi to Xeon Phi

The last experiment evaluates the latency between two Xeon Phis. Compared to the Xeon Phi local baseline we observe a round trip time which is twice as high. Compared to the previous benchmarks, the distinct buffer placement strategies provide slightly different outcome. Again, polling a remote receive buffer and sending data to the local transmit buffer is the fastest option.

5.5.4 Implications

Based on the observations of the experiments, we conclude with statistical confidence that polling a remote receive buffer and writing messages to a local transmit buffer produces the best results in terms of round trip times – no matter where we initiate the request. Perhaps the most counter intuitive result observed was the messaging latencies from the Xeon Phi to the host which are in the same order of magnitude as the Xeon Phi local measurements. Even though there is a certain overhead it is small enough that invocations of services running on the host may become an option even a necessity as we will see in the following chapters.

5.6 A Word on CPU Performance

	Host	**Xeon Phi**
Integer Operation	3309 (3.684)	1044 (1.047)
Vector Instruction (32-bit elements)	12375 (23.12)	4196 (0.457)
Vector Instruction (64-bit elements)	2781 (2.009)	2088 (0.228)

Table 5.3: Integer Performance in Millions of Operations per Second

Obviously memory access latency is not the only parameter that affects the overall performance of a system: depending on the workload, CPU computing power is the most important factor. Comparing the theoretical peak performance of the host CPU with the Xeon Phi there is a clear winner: with a nominal Trillion double precision floating point operations per second the Xeon Phi can do more

than twice as much as the host. However, recall the power wall in microprocessor history [66] and the paradigm shift towards software concurrency ("the free lunch is over", [77]): the nominal peak performance figures have to be taken with care as they are only valid for highly parallel workload which use the extra wide SIMD vector units.

Table 5.3 above shows the number of integer instructions per second for a single core. We used `addl/addq` and `vpaddd/vpaddq` instructions and divided the number of processed 32 or 64-bit elements by the measured execution time. The results show, that on the Xeon Phi the use of `AVX-512` instructions are highly recommended whenever possible.

Chapter 6

Barrelfish on the Xeon Phi

In the previous chapters we described the features and specifications of our new hardware (Chapter 4) and investigated its performance characteristics with micro benchmarks (Chapter 5). In this chapter we describe our proposed software system which runs in the heterogeneous environment consisting of `x86_64` and `k1om` micro-architectures. We base our system on Barrelfish's multikernel architecture which natively supports the desired functionality such as user-level message passing as communication between domains.

This chapter is structured in the following way: We start with presenting the global system architecture (Section 6.1) and its differences to Intel's MPSS (Section 6.2). Followed by two sections describing the boot process (Section 6.5) and the anatomy of the Xeon Phi boot image with its assembly process (Section 6.4). We will revisit capabilities (Section 6.6) in the context of multiple address spaces and talk about bootstrapping message passing channels over PCI Express (Section 6.7). In the remainder of the chapter we present an overview of the different new services available in the system (Section 6.8).

6.1 System Overview

Our proposed system adds two new domains to Barrelfish which provide the indispensable services to communicate with domains on the Xeon Phi and so forth. A graphical representation of the system can be seen in Figure 6.1 on page 51. The following subsections describe the roles of the domains and its provided services. At a top level, each of the domains shown in the scheme can be classified into domains that

1. collaborate in the boot process of the Xeon Phi, and co-processor
2. form the software runtime system for applications.

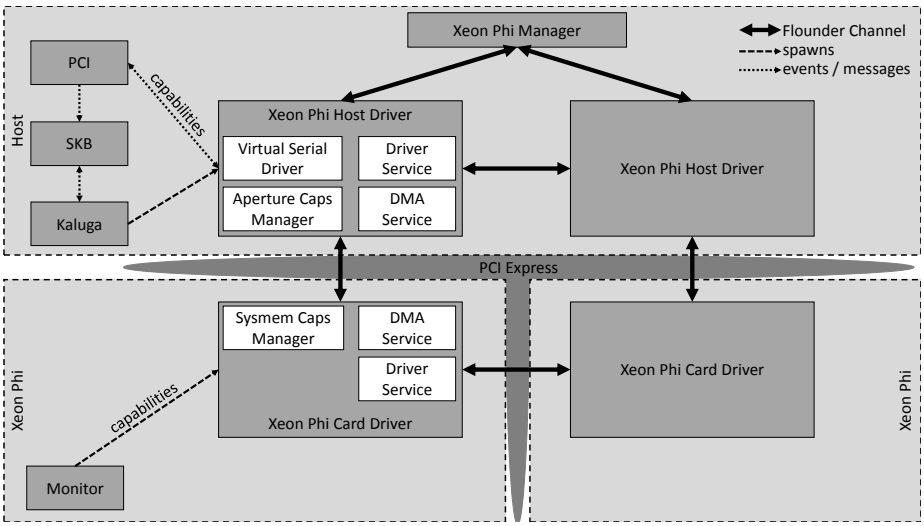


Figure 6.1: Software System Overview

6.1.1 Xeon Phi Driver

The Xeon Phi driver domains have multiple roles that differ depending on the node where the driver domain is running. The host-side driver is more complex and has to fulfill more tasks while the co-processor driver provides only a subset of the host-side functionality. We give a high-level overview of the tasks in the following sections and defer its details to Section 6.5, 6.9 and 6.8. On the host a new Xeon Phi driver instance is spawned with every Xeon Phi co-processor discovered whereas on the Xeon Phi only a single instance is running.

Host and Co-Processor Driver's Common Functionality

Both, host-side and co-processor-side Xeon Phi driver share a set of common functionality and provided services including:

1. **Driver Service** which provides an interface for client domains to spawn new domains or to transfer capabilities.
2. **DMA Service** delivering high throughput memory copy functionality between host and co-processor or GDDR–GDDR transfers.
3. **Inter-Driver Communication** All Xeon Phi driver domains are interconnected with each other in a mesh like topology. This non-public interface enables forwarding of requests to specific driver nodes.

It may be a point of argument whether the DMA service should be run as a separate domain or stay integrated into the Xeon Phi driver. We have chosen to keep the DMA service as a component of the Xeon Phi driver because the memory mapped registers of the DMA engine cannot be fully separated from the other registers. This would be a potential security risk because the DMA driver

would be able to access unrelated registers. Further, the Xeon Phi DMA engine operates with co-processor local addresses. This requires an address translation which is problematic as the SMPT entries may change. Last, the layout of DMA registers do not allow separating channels which are divided among host and co-processor. Therefore, we integrate the DMA service into the trusted Xeon Phi driver to overcome this issues.

Host Side Driver

In addition to the common features and services stated above, the host-side driver executes boot and capability management tasks. The Xeon Phi co-processor shows up as an ordinary PCI Express device during machine boot. After the PCI domain is done with basic device initialization, it informs Kaluga about the device. Kaluga spawns the driver instance and transfers control to the Xeon Phi driver for device-specific configuration. The host-side driver starts the reset and configuration sequence and boots the co-processor operating system. (Boot related steps are explained in Section 6.5).

As we already explained in Chapter 4, the GDDR memory of the Xeon Phi shows up as the PCI aperture space on the host. This memory region is represented as a capability which has to be managed. The host-side driver initializes a driver-local memory manager for the aperture space capability which is used to look up capabilities in transfer requests.

Among the additional runtime services provided by the host-side driver, there is a virtual serial device. This is necessary because the Xeon Phi lacks of a PCH south bridge with its related devices (Section 4.2). The virtual serial device supports a basic protocol to enable console output from the Xeon Phi. The host-side driver forwards the received data to its console port.

Co-Processor Side Driver

On the co-processor the driver is spawned as a special domain by monitor supplying the system memory capability. Similar to the host side driver, the local memory manager is initialized with the capability for the SMTP region. The Xeon Phi driver running on the co-processor acts as a gate keeper to the host's main memory and therefore has to be trusted.

Currently, there are no additional services provided by the co-processor-side driver. Possible future services include power management and reliability functionality.

6.1.2 Xeon Phi Manager

The Xeon Phi Manager assumes the role of administering the Xeon Phi driver domains. Analogously to the MPSS, we assign each Xeon Phi co-processor a system-wide unique identifier. The Xeon Phi IDs range from 0 to 7 and are assigned by the Xeon Phi Manager during driver initialization resulting in a support of eight Xeon Phi co-processors in a single system.

Intel specifies that in maximum eight Xeon Phi co-processors can be used si-

multaneously in a single machine. However, it is uncertain if this limit is due to hardware constraints or if it is purely software-based limitation of the MPSS[1]. In any case, we also adapt the same limitations and support up to eight co-processors in a single system.

The Xeon Phi Manager is aware of the driver instances running on the system. Besides assigning IDs, the manager is involved in bootstrapping the inter-driver communication channels on the host by distributing the registered IREFs. There is only one Xeon Phi Manager running in the entire system.

6.1.3 Service and Domain Identification

In a single address space Barrelfish, we can refer to services and domains using IREFs and domain IDs respectively. Monitors are involved by assigning the values. The IREF values are not random but encode the ID of the core the domains are running on:

```
iref = MAX_IREF_PERCORE * my_core_id + i + 1
```

Currently core IDs and domain IDs are managed node-locally. This implies we cannot tell whether the IREF or domain ID refers to a local or remote instance. Establishing an inter-node communication channel is handled without IREFs anyway (Section 6.7). However, bootstrapping the channel requires sending a capability to the target domain which is identified by its extended domain id. We extended the 32-bit Barrelfish domain ID to a 64-bit value and prepended additional information about the domain. The bit representation can be seen on Table 6.1 below.

Bits	63	62 48	47 40	39 32	31 0
Value	Host Bit	Reserved	Xeon Phi ID	Core ID	Domain ID

Table 6.1: Xeon Phi Domain ID Bit Representation

The most-significant bit of the ID represents the location, host or co-processor, the domain is running on. Depending on the bit value the Xeon Phi ID field has a different meaning. Domains running on the Xeon Phi encode their Xeon Phi ID and core ID whereas domains running on the host encode their NUMA node and core ID. The least-significant four bytes are occupied by the node-local domain ID. The values are assigned based on the location where domain is spawned. Potential domain migration is not reflected.

[1] Clearly the data structures of the Xeon Phi driver module are designed to deal with eight Xeon Phi in maximum. The memory layout in [20, Section 2.1.12] reserves the bottom 64GB for local resources followed by 7×64GB blocks labeled as "Local resources of other KNC". This introduces a hardware constrained limit of 8 co-processors. However, this memory region is declared reserved in the released Xeon Phi products.

6.2 Differences and Similarities to the MPSS

Intel provides its own software environment called Intel Manycore Platform Software Stack *MPSS* [17] for use with Xeon Phi on Linux and Windows based hosts (Section 2.1). The MPSS includes host side drivers, a co-processor operating system based on Linux and a runtime providing necessary services. The symmetric communications interface *SCIF* [21] enables the creation of socket-like channels between processes running on host and the co-processor for data transfer. Similar to Barrelfish's user-level message passing, the SCIF framework uses a shared memory approach for data exchange.

In contrast to our extension to Flounder (Section 6.7) which enables the use of Barrelfish's native channel abstractions, MPSS based applications have to use the the specific SCIF interface (Intel's MPI [18] framework is based on SCIF). The benefits for a unified communication interface are clearly its portability and flexibility for a host-only execution support. Furthermore, our system is designed to have as less kernel involvement as possible by having the host and co-processor side driver modules run in user-space. We make DMA transfers explicit and enable the user to decide when a DMA requests is sent or data is copied using `memcpy()` instead.

Our implementation does not include any power management or reliability, availability and serviceability functionality as described in [20, Chapter 3].

6.3 Modifications to Barrelfish

In general, the Xeon Phi is capable of booting any third-party operating system (Section 4.5.1) as long as the limitations stated in [20] are handled properly. In order to make Barrelfish boot on the Xeon Phi, a new Hake target architecture was introduced (Section 6.4.1). Furthermore, the kernel and certain user-space domains were adapted to reflect the specialties of the new architecture. In the following we briefly mention the patches applied to existing Barrelfish domains.

6.3.1 Boot-Loader

On `x86_64`, Barrelfish uses a tool called `Elver` as a boot-loader which transitions the CPU into 64-bit mode. On the Xeon Phi we had to develop a new boot-loader which satisfies the requirements of the Linux boot-loader as described in Section 4.5.2. Besides enabling 64-bit mode and virtual memory initialization, the boot-loader has to ensure the virtual serial device is operational by creating a mapping for the MMIO register range. More details about the boot-loader will be discussed in Section 6.5.4.

6.3.2 CPU Driver and Monitor

Representing every resources as a capability implies that there must be a way device drivers can obtain the capability of their device. In `x86_64` Barrelfish, the PCI domain manages the capabilities for each device on the PCI Express

bus. Device drivers receive their capabilities by request from the PCI domain. Obviously this cannot be done on the Xeon Phi co-processor as there is no PCI.

On the Xeon Phi, the architecture specific CPU driver creates two special capabilities referring to the MMIO registers and the system memory range of the Xeon Phi. These two capabilities are passed to the Monitor on boot-up. We treat the Xeon Phi driver instance on the co-processor as a special domain. Similar to the SKB, Monitor spawns the Xeon Phi driver directly and copies the two capabilities into well-known slots in the driver's CSPACE. This ensures that only the Xeon Phi driver can obtain the sensitive system memory capability.[2].

Besides the creation of the stated capabilities, the CPU driver also handles the special cases of the `k1om` architecture such as the extended APIC ID field or the virtual serial device for console output. Furthermore, the CPU driver parses the Xeon Phi ID from the passed parameters. The ID is cached in every dispatcher.

6.3.3 Kaluga and SKB

We decided have a separate device driver instance for each Xeon Phi co-processor in a machine. The corresponding entries were added to the device database to let Kaluga handle the spawning of the Xeon Phi driver. To make Kaluga work with the design of our system we had to adapt Kaluga to allow multiple driver instances being spawned. For this purpose we extended the device database in the SKB with additional fields:

- `multi_instance:` when set to `true` tells Kaluga that multiple instances of the driver are allowed
- `core_offset:` if non-zero the stride of the core where the next instance(s) will be started.

Kaluga keeps track of the running instances and spawns a new driver domain on the core defined by the base core id (`core_hint`) plus a multiple the `core_offset` depending on the number of already running instances. Upon spawning, Kaluga appends additional parameters to the command line. The existing arguments were extended to encode the PCI address of the device:

```
vendor_id:device_id:bus:device:function
```

All the numbers are in hexadecimal format. The first two tokens are the PCI vendor ID and the device ID and the last three token represent the address of the device on the PCI bus.

The extension was necessary because if we have multiple devices with the same vendor and device ID, we need to guarantee that there is no ambiguity when

[2]Another approach would be to have the capabilities requested similar to the IO-capability on the PandaBoard. We decided against this because requesting it should only be possible by the Xeon Phi driver which is not enforced that way.

registering with the PCI domain. Obviously, this leads to problems when only vendor and device ID are supplied. A more transparent solution would be similar to Barrelfish's PandaBoard implementation where Kaluga manages the device capabilities and passes the required ones to the driver as arguments. This not only reduces the need for requesting the capabilities, but also ensures obtaining the correct ones.

6.4 Creating the Xeon Phi Bootimage

On a `x86_64` machine, the modules specified in the `menu.lst` file are loaded using the TFTP protocol. During the loading process, the corresponding multiboot [40] information structure is generated containing the addresses, names and sizes of the modules. A pointer to the data structure is passed to the `Elver` and further to the CPU driver.

This is not possible on the Xeon Phi co-processor: the host-side driver has to provide access to the modules by downloading them into the co-processors GDDR (Section 6.5.3). Obviously, this does not generate the needed multiboot information structure. We would have to handle this on our own. We decided to follow the Linux way of booting [64] and use a boot-loader and boot-image approach which is created off-line. This section explains the major steps in generating the boot-image and its corresponding multiboot information structure.

6.4.1 Additional Hake Targets

As explained in Section 4.4, the Xeon Phi co-processor has a different instruction set architecture as `x86_64` processors. This is reflected in Hake by an separate target architecture (`k1om`). In order to build the modules needed for booting a Xeon Phi co-processor Hake has to be configured with two architectures:

```
> $BF_SOURCE/hake/hake.sh -a k1om -a x86_64 -s $BF_SOURCE
> make install
```

During the installation, the Xeon Phi boot image will be uploaded to a NFS share. The address of the NFS can be defined using the respective variable in `symbolic_targets.mk`. The build system will handle the dependencies in generating the boot-image and the multiboot information structure.

Compiler Dependency The current setup of `Hake`'s the `k1om` target uses Intel's modified GCC 4.7 (`k1om-mpss-linux-gcc`) which can be obtained as part of the MPSS SDK[17]. Other tools used in the Barrelfish tool-chain, such as Binutils, can be used without modifications in their most recent version.

6.4.2 Boot-Image Anatomy

Booting the Xeon Phi co-processor requires two files: the boot-loader `Weever` and the boot-image. The generation of these files is done in several stages using

external tools and scripts. The reader may have noticed the similarities towards Linux's `vmlinuz` and `initrd` which served as an inspiration for the setup.

Xeon Phi Multiboot Image

The contents of the Xeon Phi boot image are specified based on the entries in the `k1om.menu.lst` file. After the modules have been built, a script parses the `k1om.menu.lst` and concatenates the listed modules into single file. Adding zero paddings between subsequently modules ensure alignment on page boundaries. During the assembly process (Section 6.4.3) the offsets and sizes of each module within the boot image are stored for the multiboot information.

Weever: The Xeon Phi Boot-Loader

Weever is the first code being run on the Xeon Phi after the bootstrap (`fboot0` and `fboot1`, Section 4.5.2) has finished its execution. Weever contains the multiboot information structure based on the modules in the boot image. Therefore, `Weever` can only be built after the boot image has been created. Moreover, `Weever` contains the header with the magic values (Section 4.5.2), pointer to the command line and the assigned Xeon Phi ID which is set after download to the co-processor.

6.4.3 Building Process

In the previous section we have seen the anatomy of the boot-image and of the boot-loader. The generation process of the image involves multiple stages and external tools which can be found in `/tools/weever` directory of the Barrelfish source tree [69]. The reader can assume that the respective modules for the `k1om` architecture have already been built. Following, we briefly explain the major steps in generating `Weever` and the boot-image:

1. Parse `k1om.menu.lst` and generate a list of the modules' meta-data while concatenating them into `xeon_phi_multiboot`
2. Generate the multiboot information file, `mbi.c`, based on the extracted and adjusted module locations within the boot image (including a hard coded memory map of the Xeon Phi co-processor).
3. Compile `Weever` with the generated `mbi.c`
4. Convert the `Weever` ELF into binary format using `objcopy`.
5. Patch the missing entries in `Weever`'s header using tool `weever_creator`.

After the steps have been completed successfully, the files needed to boot the Xeon Phi co-processor are ready. Optionally, the boot-image could be compressed to speed up the loading time.

6.5 The Boot Process Explained

This section illustrates the boot process of a host machine with Xeon Phi co-processors. Compared to a machine without Xeon Phi, there are several additional requirements (Section 6.5.1) that have to be satisfied in order to successfully boot the Xeon Phi co-processors. After the discovery of the Xeon Phi, the host-side initialization sequence starts (Section 6.5.2) followed by a specific chain of boot related events (Section 6.5.3).

6.5.1 Boot Dependencies

The created Xeon Phi boot image (Section 6.4) cannot be loaded by adding it to the host's `menu.lst` file. By doing so, the total size of the modules exceeds a critical size which triggers a boot loop (Section 8.4) Therefore, we need to use an NFS share to load the Xeon Phi boot-image before we can download it to the co-processor. This adds the following dependencies to the environment the Xeon Phi is running in:

- **NFS Share** The host machine needs to be able to reach a NFS server which hosts the Xeon Phi boot image and the boot loader binary. The server URI can be passed to the driver by command line arguments (Section 6.5.2)
- **Network Stack** NFS requires a working network stack. The host machine needs a supported network interface card. For further information about networking refer to `readme_networking` in [69].
- **Device Database Entries** Different models of the Xeon Phi may have varying PCI device IDs. The corresponding entries in the device database must be added.

6.5.2 Host Side Driver Startup

We do not go into the Barrelfish boot-up process here. The reader can assume that the boot dependencies are satisfied and Kaluga has spawned the Xeon Phi driver which is about to parse the command line arguments.

Driver Command Line Arguments

The driver takes two arguments while both may be omitted. The first one specifies the address of the NFS share containing the Xeon Phi boot image. If this parameter is not supplied, the driver will try look up the boot image among the modules specified in `menu.lst` and loaded over TFTP.

The second (last) argument is expected to be the PCI Express address information as specified in Section 6.3.3. If the argument is not passed, it will ignore the PCI address and use the vendor and device ID to register with the PCI domain. It is highly recommended to supply this argument manually when Kaluga is not used to spawn the Xeon Phi driver.

A possible `menu.lst` entry would look like the following. Note we also have to start the Xeon Phi manager and the `pciaddress` is supplied by Kaluga.

```
# Xeon Phi Manager and Driver
module /x86_64/sbin/xeon_phi_mgr
module /x86_64/sbin/xeon_phi auto -nfs=server/share  <pciaddress>
```

Pre-Initialization Sequence

The Xeon Phi driver contacts the PCI domain to request the capabilities for the MMIO register and aperture spaces based on the supplied PCI address. Next, the driver binds to the Xeon Phi Manager service and registers its previously exported inter-driver interface. The registration response contains the assigned Xeon Phi ID and an array of IREFs of already running Xeon Phi driver instances in the system. The Xeon Phi driver issues a mount request for the supplied NFS location to `ramfsd`. After having successfully completed the pre-initialization steps, the actual boot process follows.

6.5.3 Chain of Boot Events

The following list contains the occurring events and initialization steps during the boot procedure of a Xeon Phi co-processor in Barrelfish. In the following enumeration "***H***" represents action executed on the host, while "***X***" stands for events happening on the Xeon Phi co-processor.

1. ***H:*** Initialization of the own Xeon Phi node and the host side inter-driver interface
2. ***H:*** Register own node IREF with the Xeon Phi Manager and obtain IREFs to the other Xeon Phi nodes
3. ***H:*** Reset and initialize the Xeon Phi hardware, setup system memory page tables with a 1-to-1 mapping
4. ***H:*** Register with the other host-side drivers running on in the system
5. ***H:*** Initialize aperture capability manager
6. ***H:*** map initial 1GB of aperture space for downloading the boot-image
7. ***H:*** Enter boot procedure:
    - 7.1. ***H:*** Read scratch-register to determine load address
    - 7.2. ***H:*** Download `Weever` into co-processor GDDR memory
    - 7.3. ***H:*** Setup command line for the CPU driver
    - 7.4. ***H:*** Download the Xeon Phi boot-image into co-processor GDDR
    - 7.5. ***H:*** Prepare `interphi` messaging channel between host and client drivers
    - 7.6. ***H:*** Fill in missing information into the bootinfo structure of the bootloader header

7.7. ***H:*** Send the bootstrap notify interrupt to the BSP core of the co-processor

7.8. ***H:*** Wait until Xeon Phi firmware signals with ready bit

7.9. ***X:*** Execute bootloader code (Section 6.5.4)

7.10. ***X:*** Set ready bit to signal host-side driver bootloader has started execution

8. ***H:*** Wait for Co-processor to have completed booting, in other words, for the connected event on the `interphi` channel

9. ***X:*** General system boot sequence as specified in `k1om.menu.lst`: cores, domains, services etc.

10. ***X:*** Initializing card side driver

10.1. ***X:*** Xeon Phi driver service initialization and export

10.2. ***X:*** Initialization of system memory capability manager

10.3. ***X:*** Wait for `all_spawnd_up` event.

10.4. ***X:*** Connect to the host driver channel to signal readiness

10.5. ***X:*** Go into message handling loop

11. ***H:*** Initialize remaining services (DMA and Xeon Phi driver)

12. ***H:*** Prepare `interphi` channels to other Xeon Phi nodes

12.1. ***H:*** Send `bootstrap` message to client driver

12.2. ***X:*** Connect to other Xeon Phi nodes and inform host when done

13. ***H:*** Register `xeon_phi.X.ready` event.

14. ***H:*** Start the services and go into message handle loop

The `xeon_phi.X.ready` event can be used by domains which want to use services on the Xeon Phi to check if the card is already on-line. The bootstrap message contains the address of a frame on the host which serves as a communication channel to other Xeon Phi co-processors in the system.

6.5.4 Weever: The Xeon Phi Boot-Loader

On `x86_64` machines Barrelfish uses a tool called `Elver` to do low-level boot time initialization steps before control is handed over to the actual Barrelfish CPU driver. On the Xeon Phi, we follow a similar approach using `Weever`. Before control is handed over to the CPU driver, `Weever` executes the following steps:

1. **Long Mode:** The bootstrap leaves the BSP core in 32-bit protected mode with paging disabled. Weever initiates the transition to long mode with paging enabled. Entering long mode as early as possible is crucial because the MMIO range is located above the 4GB boundary. Only after successfully transition to long mode basic communication using serial output is possible.

2. **Initial Page Tables**: `Weever` sets up the boot-time page tables with a 1-to-1 mapping covering the first 1GB of memory and the MMIO range for virtual serial console support.

3. **Multiboot Adjustment:** The multiboot data structure generated during build contains (offset, size) pairs of the modules from the beginning of the image. Weever updates the information with the actual address in memory.

4. **Load Kernel:** `Weever` loads the Kernel module from the multiboot image, relocates it and jumps into the entry point.

After enabling long mode, `Weever` reports possible errors by writing error codes to the virtual console.

6.6 Capabilities Revisited

In Section 4.3.3 we have stated the arising problem of a heterogeneous machine with multiple physical address spaces. The whole problem needs deeper understanding to be solved in general. However, in our system we need a way to refer to resources located in a remote address space. In this section, we present one possible solution to the problem of capability transfer and translation between distinct physical address spaces. We start off with a motivational example, why we need such a mechanism in our system followed by a description of a simplified model and how our implementation works.

6.6.1 The Problem of Multiple Address Spaces

To motivate our solution, one may assume that there is a domain running on the Xeon Phi which wants to use one of the services on the host, a file server for instance. There are two aspects in this scenario: first, there must be a communication channel for issuing requests and secondly, a bulk transfer mechanism for data transfers.

In both channel types, the problem at hand essentially boils down to the question: how to refer to a frame in memory? The domain on the co-processor allocates a buffer to hold the file and sends its (local) address to the service running on the host. The supplied target buffer needs to be translated into a valid representation on the host, otherwise the host is likely to overwrite its own memory. The same applies to bootstrapping the communication channel.

In Barrelfish, sending just a physical address is hardly not enough to do an actual transfer. To copy data into the target buffer, its backing frame has to be mapped into the virtual address space of the service domain which can only be done by presenting the corresponding capability. Therefore, we need a capability translation mechanism.

6.6.2 Capabilities: Accessing Remote Resources

Based on the motivational example from the previous section, we narrow the problem down to our system. The desired map function has the following properties:

I co-processor GDDR ↔ host PCI aperture space

II co-processor SMPT ↔ host main memory

III co-processor GDDR ↔ other co-processor GDDR

We limit our scope to a static setting, in other words, the entries of the SMPT do not change once programmed with a one-to-one mapping. This enables us to construct a single mapping function for the translations. Another simplification is the representation of remote capabilities: we only consider local capabilities which are transformed into their equivalent when they are transferred between address spaces. For instance, the co-processors GDDR capability is transformed into the aperture space capability when being sent to a host domain. This eliminates the burden for represent remote capabilities in the local address spaces and provides a mechanism to translate them on transfer request. Obviously, these simplifications must be eliminated in a complete model of multiple address spaces with dynamic mappings. However, this is out of scope for this thesis.

6.6.3 Capability Translation: An Implementation

We present the implementation of the model described in the previous section which allows establishing a shared frame between domains running on any node. We partition the implementation section into two main parts: hardware configuration and software implementation.

SMPT Configuration

In contrast to the PCI aperture space, the system memory range of the Xeon Phi allows more flexibility in programming. Despite potential security concerns, we initialize the SMPT with a one-to-one mapping between system memory pages and host main memory 16GB chunks. This allows us to have the following translation functions:

```
phys_addr_host = phys_addr_xphi + aperture_base
phys_addr_xphi = phys_addr_host + 0x8000000000
```

This enables the Xeon Phi drivers on host and co-processor to translate a received remote physical address into a valid local representation by adding the respective offset to the received physical address.

We cannot apply the same strategy to enable access the GDDR of another Xeon Phi. Recall, with the SMPT we can map any host-physical address into

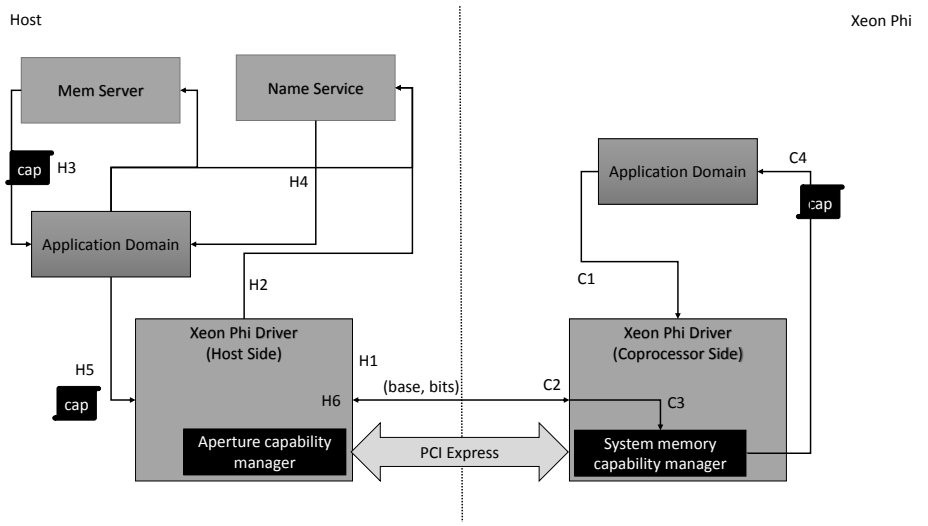


Figure 6.2: Transferring of Capabilities between Address Spaces

the local physical address space of the Xeon Phi. We can use this to install the desired mapping: the SMPT needs to be programmed with the host-physical base address of the other Xeon Phi's PCI aperture space. The reader must be aware of the typo in the Xeon Phi Systems Developer's Manual [20] providing misleading information.

The assignment of unique IDs to the Xeon Phi co-processors and the defined maximum of eight supported Xeon Phi per system allows us to specify a corresponding translation function for Xeon Phi—Xeon Phi address translations. We reserve the top 7 slots – we do not need a translation for our own GDDR – to hold the mappings for the GDDR regions of the other Xeon Phi co-processors. The address can be translated by the following function:

```
slot = SMPT_MAX_SLOTS - id
phys_addr_other = phys_addr + slot * SMPT_PAGE_SIZE
```

Note, the implementation itself is not that trivial. The SMPT pages must be aligned at a 16GB boundary. This implies we need to account for a potential offset if the aperture space is not 16GB aligned.

Capability Transfer Process

A scheme illustrating the send-process of a capability from the host to a domain running on the co-processor can be seen in Figure 6.2. As one recognizes there are several steps and domains involved during the operation. In the following, labels in parentheses refer to the labels in the figure.

In contrast to Flounder, our implementation does not require an existing communication channel between the two application domains. However, the actual sending process starts at the receiver side. The application domain on the Xeon

Phi has to register itself with the Xeon Phi driver running on the co-processor (C1). The driver stores the domain id—channel association and forwards the registration information to the host-side driver (H1) which publishes the extended domain ID (see Table 6.1) with the name-service (H2).

The sending application on the host requests a new frame from the `memserver` (H3). In order to send the frame to the application running on the co-processor, the sender needs to know the extended domain ID of the receiver which can be obtained either by querying the name-server (H4) or as a return value of the spawn request.

The actual transfer operation starts with invoking the Xeon Phi driver service supplying the capability and the target id (H5). The host-side Xeon Phi driver serializes the capability information (base, size) and forwards the request to its client driver (H6, C2). Upon receiving the message, the Xeon Phi driver translates the address into its local representation and obtains the capability from the system memory manager (C3).

Last, the Xeon Phi driver obtains the Flounder channel to the domain based on the registered domain ID and sends the capability to the target application domain (C4). It is left to the user, if the received capability is accepted or rejected.

Limitations

The described capability translation model and its implementation have several limitations. First, the system relies on a static one-to-one mapping of the system memory region. Secondly, reserving the top 7 slots of the SMPT reduces the amount of usable memory to 400GB. This may turn into a problem, if the host system has more than 400GB of main memory available. Similar strategies to paging might be applicable in that case.

Furthermore, the Xeon Phi driver domains must be trusted: with a capability to the system memory range a domain on the Xeon Phi can interfere with domains or even the CPU driver running on the host. Even though we treat the Xeon Phi drivers as a trusted domain, still requires remote capability revocation functionality.

Last, a bad capability transfer sequence can lead to allocation failures in the memory managers of the Xeon Phi drivers. To give a simple example: domain A allocates a frame and splits it up into smaller chunks. It then sends its smaller capabilities to domains B and C running on the Xeon Phi. Afterwards it wants to send the original capability to the domain D. However, the memory nodes have already been split up and the original capability cannot be found. A synchronized allocation i.e. each time a new frame is allocated on the host the changes are propagated to the system memory manager on the Xeon Phi may evict some problems at the cost of additional messages.

6.7 Flounder over PCI Express

As sketched in Section 3.4, Flounder plays a central role in Barrelfish. On a shared address space, channel bootstrap using `bind` and `export` events involves Monitor participation. The request is handled by the Monitor which uses its IREF—endpoint association to inform the service domain of a new connection. This is not applicable in a heterogeneous environment.

The following sections we give details about how we adapted Flounder to run between different physical address spaces. We made use of the memory layout design of the Xeon Phi, in particular the ability to access remote memory through aperture space or SMPT respectively.

Using Flounder to send messages between physical address spaces in a different setup has already been done. Hauenstein et al. [37] implemented a new backend to support Flounder-based message passing over Ethernet. In contrast to this implementation, we do not have to pack Flounder messages in Ethernet frames and send them over the network. Other aspects such as node-to-IP mappings can be transformed into a node-to-address mapping, in other words address translation (Section 6.6.3).

6.7.1 Founder Bootstrap Steps

Flounder supports multiple backends of which the shared memory variant is of interest for us. We decided not to implement a new Flounder backend but rather extend the existing UMP backend. Bootstrapping the underlying UMP channel can be separated into two parts:

1. Setting up a shared frame between two domains.
2. Initializing a channel over that shared frame.

This high-level simplification enables us to recognize the two essential problems at hand. First, with our implementation of transferring and translating capabilities between physical address spaces (Section 6.6.3) we have the basic mechanisms ready to set up a shared frame between two domains running in different nodes. Point (1) is solved by our software framework.

So far, the generated Flounder message stubs do not provide an option to pass information about a frame to be used for the channel. Rephrasing the problem of point (2), we have to adapt the generated Flounder message stubs to bypass Monitor and IREF based `export` and `bind` events. We added two new functions and a new data structure to the interface.

6.7.2 Message Stub Interface Extensions

The added function declarations and the data structure definition can be seen in Listing 6.1 on page 67. We use the prefix `if_` to refer to a generic interface.

The data structure `if_frameinfo` contains the information about the shared frame to be used for the channel. We use virtual addresses for the receive-buffer and transmit-buffer-pointers to give the user more control. Using a capability instead is impractical because a single one might be used for both, receive and transmit buffers.

The two additional functions `if_accept` and `if_connect` are adapted from their `export` and `bind` counterparts. They take an `if_frameinfo`-pointer as argument instead of an IREF. Both support continuation functions which are called upon channel-established events.

6.7.3 Usage

We aimed for a consistent programming model for the two new functions. Despite different function names and signatures, their use is the same as the original `export` and `bind` functions.

On the service side, the user needs to call `if_accept` which initializes a listening endpoint on this side of the channel. The client calls `if_connect` to start the handshake protocol. Receiving the one of the added internal control messages, the corresponding event triggers and the registered continuation functions are called. At this point, the channel is established. Afterwards, there is almost no difference to a Flounder channel bootstraped by `export` and `bind`.

The implementation can also be used within the same address space or even between two threads of the same domain. Reasons for choosing the manual setup of our extensions include more control of the used frame and the explicit use of a shared frame instead of the local channel backend.

```
/*
 * The message buffer structure (for accept/connect)
 */
struct if_frameinfo {
    /* Physical address of send buffer */
    lpaddr_t sendbase;

    /* Pointer to incoming message buffer */
    void *inbuf;

    /* Size of the incoming buffer in bytes */
    size_t inbufsize;

    /* Pointer to outgoing message buffer */
    void *outbuf;

    /* Size of the outgoing buffer in bytes */
    size_t outbufsize;

};

/*
 * accept method to allow incomig connections
 */
errval_t if_accept(struct if_frameinfo *_frameinfo,
                   void *st,
                   if_bind_continuation_fn *_continuation,
                   struct waitset *ws, idc_export_flags_t flags)


/*
 * connect method to connect to the service accepting new
 * connections
 */
errval_t if_connect(struct if_frameinfo *_frameinfo,
                    if_bind_continuation_fn *_continuation,
                    void *st, struct waitset *ws,
                    idc_bind_flags_t flags)
```

Listing 6.1: Added Flounder Interfaces

6.7.4 Limitations

In contrast to the IREF based channel setup, there are some limitations in the use and setup of the channel. First, the user must make sure that the `if_accept` function is always executed before the `if_connect` function. If the client side initializes the channel prior the service side, the receive-buffer containing the control message is zeroed out resulting in a missing connect event.

Secondly, filling out the `if_frameinfo` structure has to be done with care. Mixing up the order of send and receive buffers results in an unusable channel.

Last, we do not support capability transfers with this channel even if both domains reside in the same address space. This is because Monitor was not involved in the channel setup and hence certain identifiers in the channel metadata structure are not set. In any, case sending a capability across address

spaces is not straight forward as elaborated in Section 6.6. Clearly, this is a functionality future implementations should support.

6.8 Xeon Phi Service Interfaces

The Xeon Phi driver provides three different services to the other domains running in the system:

1. Xeon Phi Driver Service
2. Xeon Phi Name-Service
3. DMA Service

We will outline the interface specification of the driver service and name-service in this section, and postpone the DMA service to Section 6.9 because its interface applies to all DMA services in general.

We provide a Xeon Phi client library which abstracts the communication layer to the Xeon Phi driver. The library is architecture specific as certain invocations behave differently on the host than on the co-processor. For instance, registering a new entry in the name-service goes to the Xeon Phi driver on the co-processor whereas the request is sent directly to Octopus on the host.

6.8.1 Xeon Phi Driver Service

The functions and related declarations to use the Xeon Phi driver services are declared in the following headers:

```
#include <xeon_phi/xeon_phi.h>
#include <xeon_phi/xeon_phi_client.h>
```

The library serves two purposes: first, providing an interface for clients to actively issues requests to the driver. Secondly, giving the driver the possibility to notify the client about new events such as a received capability.

Before the library can be used, the client has to register itself with the Xeon Phi driver. This can be done by calling `xeon_phi_client_init` which executes the handshake protocol with the Xeon Phi driver. The registration is handled implicitly upon issuing the first request to the driver. In case the domain wants to wait for an event first, the registration has to be done explicitly. An extract of the provided functions and its signatures can be seen in Listing 6.2 on page 70. Following, we will briefly explain the spawning of domains and opening a channel (sharing a frame).

Spawning Domains

In `x86_64` Barrelfish, spawning a new domain allows the selection of a core on which the domain should be spawned. In a heterogeneous system, in addition

to the core we can also select the compute-node we want to spawn the domain on. Therefore, we need to provide functionality to spawn domains in a foreign address space.

The interface of our `xeon_phi_client_spawn` function is adapted from the interface of Barrelfish's `libspawndomain`. In addition to the core, the user also needs to specify the Xeon Phi ID (`xid`) of the target node. The local Xeon Phi driver will receive the request and forwards it to the node specified by the `xid` parameter.

Upon receiving the request from its peer Xeon Phi driver, the domain is spawned using the node-local spawning service (`spawnd`). The returned domain id is extended by the core and node information and returned to the calling domain.

Our interface also supports additional command line arguments and a capability to be handed over to the new domain. The capability is copied into the ARGCN slot of the domain's CSPACE. This reduces the steps needed to bootstrap a communication channel between the two domains: the supplied capability can directly be used to bootstrap the channel. The connection event can be used to signal readiness. Note, in contrast to `libspawndomain`'s spawn functions, we currently only support handing over a single capability not an entire CNODE.

```
/**
 * \brief initializes the Xeon Phi client
 *
 * \param xid   Xeon Phi ID of the card to initialize
 */
errval_t xeon_phi_client_init(xphi_id_t xid);

/**
 * \brief spawns a new domain on the Xeon Phi or on the host
 *
 * \param xid       Xeon Phi ID to start the domain
 * \param core      Core to start
 * \param path      Program to spawn
 * \param argv      Program arguments
 * \param cap       Capability to pass
 * \param flags     spawn flags
 * \param domid     returns the domain id of the spawned domain
 */
errval_t xeon_phi_client_spawn(xphi_id_t xid, coreid_t core,
                               char *path, char *argv[],
                               struct capref cap, uint8_t flags,
                               xphi_dom_id_t *domid);

/**
 * \brief sends an channel open request to the domain
 *
 * \param xid       Xeon Phi ID
 * \param domid     Domain ID
 * \param usrdata   Supplied data for the other side
 * \param iface     Interface name of the domain
 * \param msgframe  Message frame
 * \param chantype  Type of the channel
 */
errval_t xeon_phi_client_chan_open(xphi_id_t xid,
                                   xphi_dom_id_t domid,
                                   uint64_t usrdata,
                                   struct capref msgframe,
                                   xphi_chan_type_t chantype);

/**
 * \brief sets the callbacks for incoming messages
 *
 * \param cb    Xeon Phi callbacks
 */
void xeon_phi_client_set_callbacks(struct xeon_phi_callbacks *cb);

/**
 * \brief Callback function typedef for channel open events
 */
typedef errval_t (*xphi_chan_open_t)(xphi_dom_id_t domain,
                                     uint64_t usrdata,
                                     struct capref msgframe,
                                     uint8_t type);
```

Listing 6.2: Xeon Phi Driver Interface

Sending Capabilities

The ability to establish a shared frame between two domains running on different address spaces is crucial for bootstrapping a message passing between

them (Section 6.7). A capability transfer request can be issues by invoking the function `xeon_phi_client_chan_open()`.

As a prerequisite, the caller must know the extended domain ID of the target domain. The ID can be obtained through a lookup using the name-service (6.8.2) or as a return value of a spawn request. Additionally, the receiver must register a callback handler for the notification using `xeon_phi_client_set_callbacks()`. If these requirements are not satisfied, the transfer fails.

The local Xeon Phi driver obtains the channel to the target domain by a lookup based on the supplied extended domain ID. It then sends the capability over the Xeon Phi driver channel to the target domain. The library handles the receive and triggers the registered callback. The application can decide whether to accept or reject the received capability. The library passed the ID of the sending domain to the receiver by callback arguments.

Parameter `usrdata`: This argument of 64-bit size can be used to supply further information to the receiving domain. The semantics of the values passed for this parameter are completely user defined. Possible use cases include the virtual address where to map the capability or a pointer to a data structure containing additional information.

Parameter `chantype`: To give a hint to the receiving domain what the shared frame should be used for or how it should be mapped the interface supports some basic channel types. The type parameter can be used for predefined types such as shared frame or Flounder channel or have user defined semantics.

6.8.2 Name Services

Barrelfish's name-service is basically a key value store. It serves two purposes: First domains can register the IREFs of their exported services with a name. Other domains can query the name-service using the name to obtain the IREF needed for binding to the exported service. Secondly, with the ability to wait for a specific name – an event in this case – the service also serves as a basic form of synchronization and barrier during system boot. The calling domain blocks until the corresponding name is registered.

This name-service runs node-local and therefore cannot be used directly to synchronize between domains running on different nodes. However, there is a need for having exact this functionality of a global name-service: for instance, the Xeon Phi ready events must be receivable from any node. Moreover, domains must be able to query the name-service to obtain whether a certain domain is running and its domain ID for transferring a capability in order to establish a communication channel. We implemented the support for a system-wide accessible name-service.

Global Name Service

There are many possibilities how to design and implement such a globally accessible name-service. We will present our implemented solution first and do a survey of alternative architectures later (Section 6.8.2).

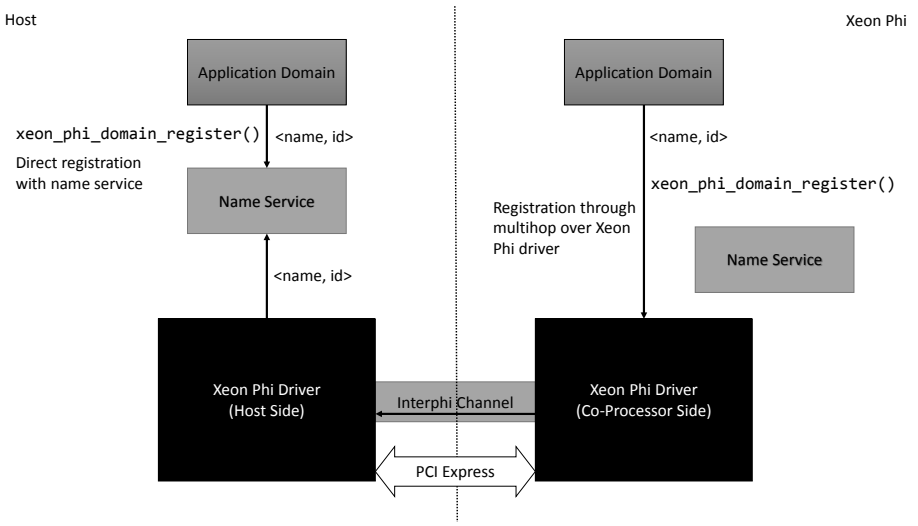


Figure 6.3: Xeon Phi Name Service

Our design features a hierarchical name-service, distinguishing between global and local entries. A scheme of the service architecture can be seen in Figure 6.3. We justify our approach by the observation that certain events are purely node-local and have no meaning in another node or they occur in every node such as `all_spawnd_up`. These events do not have to be globally accessible. Other events can make system-wide known explicitly.

Our multi-level name-service consists of un-modified local name-services dealing with purely local events. This allows the system within a node to run without any further modifications. We would like to emphasize the double-role of the name-service running on the host. We use it to store our global accessible entries in addition to its local records. From a host-side view, the global entries do not differ from the local ones.

The name-service records can be registered and queried system-wide using our new API (Listing 6.3). The request execution itself varies depending on the originating node as shown on Figure 6.3:

- **Host:** The API calls on the host are directly transmitted to the (local) name-service.
- **Co-Processor:** The request from a domain running on the co-processor is received by the Xeon Phi driver at first hand. Secondly, it gets forwarded over the inter-phi channel to the host-side driver. Last, the host-side Xeon Phi driver forwards the request to the local name-service on behalf of the application domain running on the co-processor.

```
2  /**
   * \brief builds the iface name representation
4  *
   * \param name  Name of the domain
6  * \param xid   Xeon Phi ID or XEON_PHI_DOMAIN_HOST the domain is
   *              runnig on
8  * \param core  The core the domain is running on
   *
10 * \returns string with the proper format
   */
12 char *xeon_phi_domain_build_iface(const char *name,
                                    xphi_id_t xid,
14                                  coreid_t core);

16 /**
   * \brief Blocking name service lookup
18 *
   * \param iface     Name of the domain
20 * \param retdomid  returns the Xeon Phi Domain ID
   */
22 errval_t xeon_phi_domain_blocking_lookup(const char *iface,
                                          xphi_dom_id_t *retdomid);
24
   /**
26 * \brief Register with name service
   *
28 * \param iface     Name of the domain
   * \param retdomid  returns the Xeon Phi Domain ID
30 */
   errval_t xeon_phi_domain_register(const char *iface,
32                                  xphi_dom_id_t domid);
```

Listing 6.3: Xeon Phi Nameservice Interface

Interface Specification

Before starting with the interface description, we want to give a motivational example to justify our design. The reader may assume we have a distributed service running on multiple cores on the host and the Xeon Phi. Clients want to ask the following queries:

- What is the domain ID of the service instance running on Xeon Phi 0, core 3?
- Is there a service instance running on the host?
- Is there any instance of the service running in the system?

If we want to answer these questions, we need a way to exactly refer to a specific service or refer to any service instance running in the system. Our implementation of the global name-service supports these types of questions.

The interface specification and its semantics are adapted from the name-service client of `libbarrelfish`. We support the same basic functionality such as

registering of records, blocking and non-blocking lookups. The interface can bee seen in Listing 6.3 (we omitted the non-blocking lookup function here).

In contrast to `libbarrelfish`'s name-service client, we store 64-bit sized values. This is necessary to hold the extended domain id (Table 6.1). In addition, we provide a more flexible way of querying the key-value store. The library provided a function to construct the interface name which contains the node and core information (`xeon_phi_domain_build_iface`). When a domain registers with the Xeon Phi driver, a record is created containing the ID and the encoded name of the domain using the build iface function.

The `xeon_phi_domain_build_iface` function can also be used to construct the name of the record for querying. The node and core information can be omitted by using the wild card value `XEON_PHI_DOMAIN_DONT_CARE`. This converts the generated name into a regular expression matching possible multiple entries.

Alternative Approach

In our proposed architecture we distinguish between a local name-service and a system-wide accessible name-service running on top of the local ones. We present two alternative designs of a global name-service:

1. **Global Nameservice** This architecture features only a single name-service running on the host. In this setup, there is no change on the host, whereas on the co-processors every access to the name-service has to go over PCI Express. The benefits for this setup is clearly a single location where the information is stored which is also a negative point at the same time (single point of failure). Other drawbacks for this approach include channel setup form co-processor domains, multiple records with the same name and need to assign prefixes to names.

2. **Distributed Nameservice** This alternative design is based on a fully distributed architecture. Each node has its local name-service containing a replica of the global state. Changes are broadcasted to all other nodes. This eliminates the connection setup problem of the previous alternative but introduces additional messages to keep the state up-to-date. The maximum number of name-servers is 9 per system, one for the host and one for each of the Xeon Phis. Updates will eventually been received implies that requests can observe stale values. In addition, we still face the name-clash problem.

After all, based on the drawbacks of the two alternative approaches, having a multi-level name service with explicit global entries is expected to be the optimal solution in this case. It allows fast local lookups while providing a transparent interface for global accesses.

6.9 DMA Service

With an 8 channel DMA controller, the Xeon Phi has sufficient capacity to offload memory transfers to the DMA engine. Our benchmarks (Section 5.2) have shown that the use of these DMA channels is highly recommended. After we give a brief overview of the DMA capabilities of the Xeon Phi, we introduce the generic DMA framework with its library (`libdma`), DMA Manager service and the programming model.

Xeon Phi DMA Engine Each DMA channel can be owned either by the host or the Xeon Phi. The location of the descriptor rings is determined by the ownership. We distribute the channels evenly between host and co-processor. The DMA engine supports the following transfers:

- GDDR—GDDR transfers (co-processor local)
- Host system memory—GDDR transfers
- GDDR—host system memory transfers
- Co-processor—co-processor transfers

6.9.1 DMA Library

The DMA library was introduced with the intention to provide a unified interface for different DMA devices such as the Xeon Phi DMA engine or Intel's I/OAT Crystal Beach 3 DMA controller. We use the following abstractions:

- **Device:** A DMA device abstracts a physical DMA controller (driver side) or a virtual DMA device (client side). A DMA device may have multiple DMA channels.
- **Channel:** A DMA channel abstracts a hardware descriptor ring (driver side) or a Flounder channel to the driver service (client side). DMA channels manage DMA requests issued to a specific channel.
- **Request:** A DMA request abstracts a DMA operation such as copying memory. On the driver side, a request is a collection of one or more DMA descriptors on the descriptor ring. On the client side, a DMA request is a pending notification from the DMA driver.

There abstractions are relevant for both, driver developer and clients using the DMA service.

Driver Side Initialization

There are two different parts of the library that have to be initialized by the device driver: first, configuration of the device hardware and secondly, the generic DMA service. The library provides hardware specific backends for the different

DMA devices. The driver needs to initialize the appropriate backend to configure its device. During the initialization, DMA channels are discovered and their descriptor rings allocated.

As soon as the hardware is ready, the driver starts with setting up the DMA service. The library provides a generic service interface which supports request issued by the DMA client device. Drivers register their service callbacks to get informed of new requests. The callbacks can be used to verify the (address, length)-paris (Section 6.9.3) .

Client Side Initialization

The differences in the initialization sequence on the client side compared with the driver side are minimal. The client starts with configuring its DMA client device, a virtual device abstracting the underlying flounder channel. Based on the supplied arguments, the setup function connects to the corresponding DMA service.

6.9.2 DMA Manager

DMA engines are normal devices residing on the PCI bus or on the chipset. With an increased core count, it is likely that then number of available DMA engines also increases, especially with DMA controllers directly integrated into the CPU as it is the case with Intel's I/OAT technology. We observed, that not all DMA controllers available in a system provide the same transfer capabilities. Therefore, we need a way to find the right DMA service for our transfer.

DMA drivers announce their transfer capabilities to the DMA Manager which keeps track of running DMA drivers in the system. Client domains can query the DMA Manager to obtain the IREF of the DMA service most suitable for their intended transfers. The DMA Manager queries its local driver database based on the source and destination address.

The DMA Manager offers clients to find the best driver for their needs. It is also possible to bind to the service based on its exported interface name. (refer to `dma/dma.h`)

6.9.3 DMA Subsystem Programming Model

One of the goals of Barrelfish's DMA library is to provide a transparent interface to its users. The programming model of the DMA library for issuing DMA requests is described in the following list. The signatures of the relevant functions are shown in Listing 6.4.

1. **Initialization of the DMA client device** (`dma_client_device_init`). Based on the argument value the setup tries to connect to a specific service using its name or IREF. Alternatively, asking the DMA Manager for a suitable service based on an supplied address range.
2. **Memory registration** (`dma_register_memory`). The DMA engine deals with physical addresses only. To avoid sending capabilities on every re-

quest while enforcing memory isolation at the same time, memory regions have to be registered beforehand by presenting a frame capability to the driver. It's up to the driver to do the book keeping.

3. **Issuing a DMA request** (`dma_request_memcpy`). The actual request to execute and the parameters are determined by the values of the setup parameter.

DMA drivers are highly recommended implementing a memory verification method or use the standard implementation of the DMA library. Invalid DMA requests are rejected by the service.

```
/**
 * \brief initializes a DMA client device with the giving
 *        capability
 *
 * \param info      stores information how to find the device
 *                  driver service
 * \param dev       returns a pointer to the device structure
 */
errval_t dma_client_device_init(struct dma_client_info *info,
                                struct dma_client_device **dev);

/**
 * \brief registers a memory region for future use
 *
 * \param frame     the memory frame to register
 */
errval_t dma_register_memory(struct dma_device *dev,
                             struct capref frame);

/**
 * \brief issues a new DMA memcpy request based on the setup
 *        information
 *
 * \param dev      DMA device
 * \param setup    DMA request setup information
 * \param id       returns the DMA request ID
 */
errval_t dma_request_memcpy(struct dma_device *dev,
                            struct dma_req_setup *setup,
                            dma_req_id_t *id);
```

Listing 6.4: Extract of the DMA client library

6.10 Virtual Devices

This section briefly discusses possibilities of using virtualized devices on the Xeon Phi with the Barrelfish OS and explains why VirtIO may not be your first choice. The lack of (PCI-)devices, especially network interface cards and disks, are one of the drawbacks for the OS and applications running on the Xeon Phi. Device support can be provided by implementing techniques known from

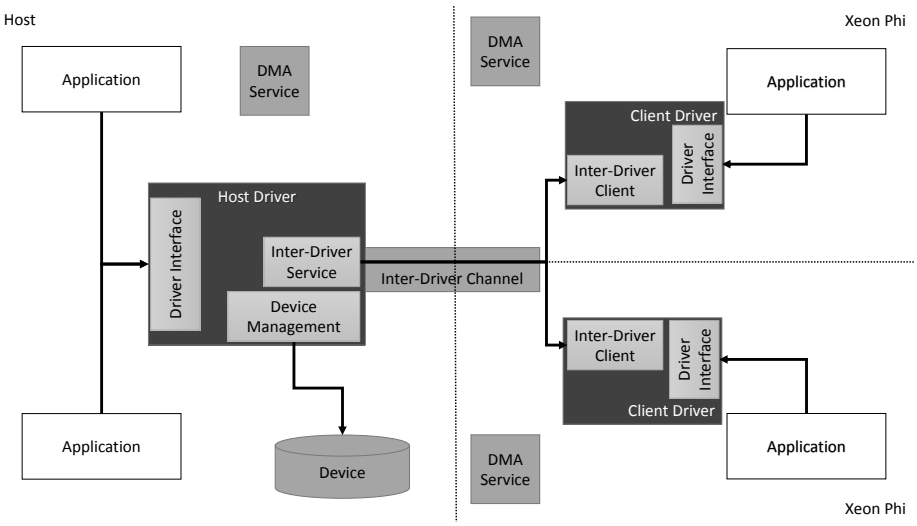


Figure 6.4: Emulating Virtual Devices on Co-Processors

virtualized environments. A hypervisor, Xen [6] for instance, prepares virtual devices which can be used by the guest operating system.

This model can be transformed to our system. The host OS takes over the role of the hypervisor and the co-processors OSes are the guests. Based on this model and our software framework, we present an exploration of possible ways how we can emulate virtual devices in Barrelfish.

Basic Device/Driver Structure In our proposed system architecture, we have made use of one possible model for emulating virtual devices. The Xeon Phi driver is split into two parts acting as a gateway for issuing requests on another node. Transforming this approach to virtual devices we can run a host-side driver which accesses the device hardware or the actual driver service similar to a proxy. A client driver on the co-processor connects to the host-side driver using an inter-driver communication channel. Requests from the co-processor are forwarded to the host-side driver in a multi-hop fashion. A scheme of a possible architecture can be seen in Figure 6.4

Besides the control channel, certain applications, file servers or network interface cards for instance, are likely to move large amounts of data around. This requires an efficient bulk-transport mechanism. Obviously, the hardware should directly DMA into the buffer residing on the co-processor if possible. In the following sections we present two possible ways how bulk transport may be done.

6.10.1 VirtIO

Virtualization technology enables multiplexing of a single machine to multiple operating systems at the same time. While certain resources such as the CPU can be multiplexed similar to scheduling another domain, this may not be the case with other hardware devices. In a virtual environment, devices such as network interface cards or block devices either provide virtual functions or a

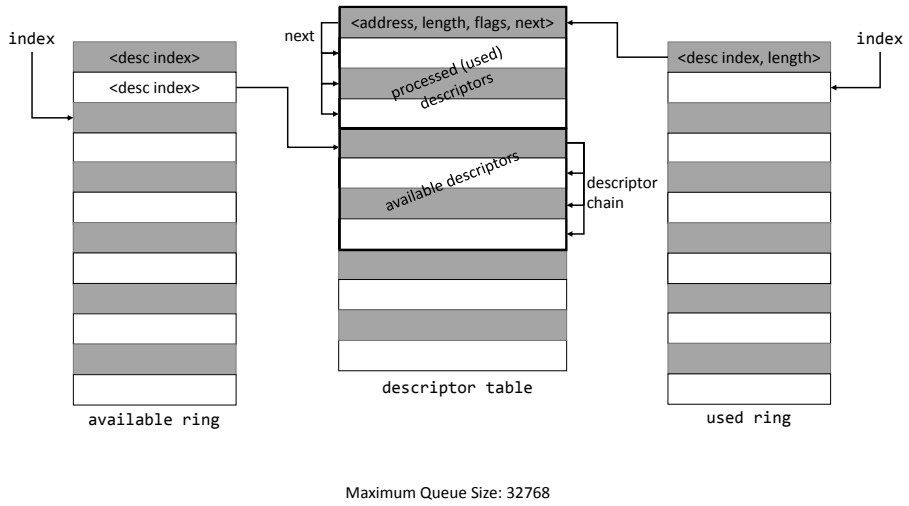


Figure 6.5: VirtIO: Virtqueue Operation Scheme

virtual device has to be emulated by the host. Rusty Russel discovered the problem with the various different visualization systems [72]. One of which is the variety of standards which are not always well maintained. He proposed VirtIO [78, 45] as the de-facto standard for virtual IO devices. VirtIO is integrated into the Linux kernel with the goal to provide a set of well maintained drivers. In his paper, Russel describes how VirtIO devices communicate with their device drivers in the virtualized environment. In the following sections we will give an overview of the assumptions made by VirtIO as well as giving reasons why it may not be the first choice in a capability based multikernel operating system such as Barrelfish.

Programming Model

In a virtual environment, the hypervisor emulates a VirtIO devices and exports it to its guest operating system using one of the possible transport specifications. The VirtIO device is discovered and a compatible VirtIO driver is loaded. Both, VirtIO device and driver, need to make sure they obey the VirtIO specification [78].

The basic operation scheme is shown in Figure 6.5. VirtIO uses the abstraction of a `virtqueue` as a communication layer. A VirtIO device has one or more `virtqueues` which form the data path between device and driver. The standard specifies multiple options for the control plane including PCI, MMIO or Channel IO [78, Chapter 3]. Each `virtqueue` consists of a descriptor table for buffer descriptors and two rings called available and used ring.

The VirtIO driver writes the buffer addresses into the descriptor table and sets the index of the available ring to point to the corresponding slot in the descriptor table. The device consumes the available buffer and updates the used ring once finished with the request.

Assumptions Made

VirtIO was designed to be used in a virtual environment. Following we present a list of assumptions made by VirtIO which obviously hold in a virtual environment.

- **Buffer Locations:** Like normal devices, VirtIO device drivers write physical addresses into the descriptors. However, in a virtualized environment these addresses are guest physical addresses rather than machine physical addresses. In any case, the buffer location will be in main memory of the machine and hence accessible to all hardware devices.
- **Device Registers:** Writing to the device registers should work as with normal devices such as defined in the PCI standard [65] (if the PCI transport layer is used).
- **Accessing Buffers and Queues:** The VirtIO device must be able to access the descriptor table, its rings and the supplied buffers. With the VirtIO device being emulated in the hypervisor this is naturally satisfied as the hypervisor is expected to be able to access the entire machine.

VirtIO in a Heterogeneous Multikernel OS: An Evaluation

The ideas behind VirtIO are reasonable well founded and is applicable to a virtualized environment using the hypervisor/guest setup. The implementation of a VirtIO library for Barrelfish revealed several issues in the usability of VirtIO. We claim that using VirtIO in a heterogeneous multikernel OS such as Barrelfish is not the first choice. Following a list of reasons against VirtIO:

I **Buffer Locations** VirtIO buffer locations reside on the guest side. In a heterogeneous system such as the one described in Section 4.1 the buffers would be allocated on the Xeon Phi. Accessing a remote buffer results in high latency and low throughput as shown in Chapter 5. Therefore, if data has to be processed by the VirtIO device residing on the host, its buffers need to be copied into host memory. This requires the host to allocate memory on its own behalf rather than supplied by the device driver.

II **Accessing Buffers / Descriptor Rings** The domain emulating the VirtIO device must be able to access the supplied descriptor rings and its buffers. While this applies to the hypervisor/guest scenario, it does not necessarily in capability based systems: The VirtIO device domain would need the capability for any of the descriptor rings and buffers. Furthermore, devices may not be able to DMA into the aperture space of the Xeon Phi.

III **Mapped Buffers** The VirtIO device domain is required to access the buffer locations because they contain information about the requests to execute. Hence, every buffer must be mapped in the virtual address space of the domain.

IV **Guest Physical Addresses** The hypervisor is maintaining the second level page tables in the virtual environment and therefore it is possible to do an address verification and translation of the supplied buffers. In a capability based system, the domain needs to present a capability for the corresponding buffer to authenticate the address. Furthermore, the VirtIO device domain must be able to translate the address into a local representation.

V **Multikernel OS Structure** In Barrelfish device drivers are running in their own domains. This requires that a VirtIO device must be able to forward the buffer location to the actual driver domain. In other words, this means we need to distribute the capabilities among multiple domains.

VI **Emulating Device Registers** Even though the emulation of VirtIO device registers over shared memory (MMIO) is possible, its operational characteristics are rather tricky: the registers have to be polled by the VirtIO device. In a message passing based system, transforming the registers to either device specific messages or generic offset-length requests are preferable.

To sum up, implementing VirtIO in a heterogeneous multikernel environment introduces additional overhead to a hypervisor/guest setup. Next, we will present an alternative to VirtIO based on the Barrelfish bulk transport framework.

6.10.2 Barrelfish Bulk Transport

Recent semester projects [1, 11] explored the possibilities on how to do bulk transport in a multikernel OS efficiently and secure. The ideas behind the architecture of the proposed bulk transport framework could be adapted to the heterogeneous architecture of our system. We justify our claim by the fact that their proposed framework already supports bulk transport between two distinct address spaces using Ethernet. To enable bulk transport between host and coprocessors similar techniques can be applied. Strategies of the Ethernet backend for receive buffer handling combined with control channels of the shared memory backend and DMA transfers is expected to produce an optimal framework. Due to time constraints this approach could not be implemented.

Comparison to VirtIO

The proposed bulk transfer framework of [1, 11] differs significantly from VirtIO. Following, we list the major differences between the two architectures:

Buffers In contrast to VirtIO, the bulk transport framework makes use of buffer pools that are allocated and assigned to a channel. Only buffers belonging to an assigned pool can be sent over a channel. On the other hand, VirtIO does not know such restrictions. Using the bulk transport framework, the problem of authenticating addresses and distributing capabilities can be done once during assignment time of the buffer pool.

Descriptor Rings By using message passing as the control channel, the need for polling a descriptor ring and device registers, in other words, multiple mem-

ory locations, is no longer existent. Each advance of the head pointer can be translated into a message with the needed information. Furthermore, we can use a single control channel can be used for all data paths, which eliminates polling multiple UMP channels.

Buffer Re-Use A bulk transfer pool can be assigned to multiple channels resulting in a multi-hop path. That way a buffer can be easily shared and passed around to a network stack or file system/block service stack without the need for wrapping the buffers into another framework.

Buffer Sizes The bulk transport implementation allows only buffer sizes which can be represented by a capability (in minimum page size and a power of two). This gives the benefits of associating a single buffer to a capability which is not the case in VirtIO where the buffers can have arbitrary sizes.

6.10.3 Virtual Devices: A Conclusion

Applying common virtualization techniques in a heterogeneous multikernel system results in several pitfalls because some assumptions no longer hold. Transforming the VirtIO descriptor rings into a message passing based equivalent is definitely the way to go. We give further directions in the future work section (Section 8.11). In any case, applying the Xeon Phi driver model combined with elements of VirtIO may bring the best of both architectures together.

Chapter 7

Applications

In the previous chapters we have seen the hardware features of the Xeon Phi with its performance characteristics and the architecture of our proposed software framework. In this Chapter we will investigate possible use cases and applications of the system. In order to set the context of potential applications we first discuss offloading in general (Section 7.1). Next we present our implementation of a non-shared address space version of the OpenMP library for parallel processing (7.2). In Section 7.3 we evaluate the performance of an offloaded OpenMP matrix multiplication and compare it with host only execution.

7.1 Work Offloading

A generic use case for any co-processors in particular the Xeon Phi is to offload from the main processor. The motivations why offloading is used are versatile. There is a strong desire to complete a certain task more efficiently in terms of time and/or energy. We give four examples to motivate possible use cases of offloading:

I **Free Host Resources:** Assume a server handling tasks of different types, some of which are long running. The server will receive normal and high priority tasks from its clients and should process them accordingly to their assigned priority. In order to free up host resources, low priority tasks and long running tasks are offloaded to the co-processor. This gives the host the possibility to use the local resources for high priority or small tasks resulting in an better response time in expectation while making progress on the long running task simultaneously.

II **Specialized Hardware:** The current CPUs are general purpose compute units. Even though they are capable of handling any task its execution may require additional instructions which slows down the overall computation or even affects other programs running on the same machine by spilling the cache for instance. Hence offloading certain parts of the computation to

specialized hardware such as a crypto co-processor or video decoder card for instance not only reduces the used cycles of the host CPU but also diminishes possible effects on other running tasks.

III **Energy Efficiency:** With specialized hardware as described in the previous paragraph the overall execution may be carried out faster and more efficient resulting in a lower energy consumption compared to the host CPU-only execution. This is not only important for mobile devices but also for huge data centers as a reduction of some milliwatts per request results in a noticeable effect on the total operating costs.

IV **Exploiting Parallelism:** By adding new (co-)processors software may be able to exploit the offered parallelism by distributing the work among more nodes resulting in a faster execution in terms of lower response time and higher throughput.

Offloading with the Xeon Phi: The use case of the Xeon Phi can be best described with scenario I and IV: the host exploits the compute power of the co-processor to offload certain tasks of the execution or even complete tasks natively to the Xeon Phi. Further with its huge number of cores and the extra wide vector registers the Xeon Phi brings massive parallelism to the system. However, in contrast to GPGPUs the level of specialization of the Xeon Phi is rather low compared to the general purpose `x86_64` CPUs.

7.1.1 Requirements for Offloading

Recall the memory layout of the Xeon Phi which introduces a separate physical address space for each co-processor. In addition to that we have a distinct architecture compared to `x86_64`. Hence, in order to make offloading work several things have to be ensured:

1. Code to execute in the target architecture(s)
2. Synchronization between the nodes (including locks to prevent race condition)
3. Access to data of the working set

In contrast to some highly specialized accelerators such as audio decoders, general purpose co-processors need to know what has to be computed. This implies that the co-processor must be able to obtain or work out the sequence of instructions to execute. Thus, for each target architecture there must be a compatible version of the binary available.[1] This requires that the entire application or specific parts of it are compiled with the target instruction set architecture of the co-processor (unless there is a shared subset supported by both architectures).

Secondly, a basic form of synchronization among the execution units is required to avoid data races and inconsistent states when a shared data structure is

[1] This can be separate or integrated into one big binary as in [79]

accessed. Further barriers are needed to coordinate the progress of the participating compute nodes.

Third, the co-processor must have access to the data of the working set. This can be achieved by having the host loading data into the co-processor's memory and/or the co-processor must be able to access the host memory directly (pull or push model). The Xeon Phi is capable of doing both. Recall the measurement results of Chapter 5, therefore choosing an appropriate the data location *is* of great importance and suitable techniques for replication must be applied.

Offloading in Barrelfish

Translating the described requirements above to the Barrelfish environment we must be able to:

1. Execute code on the co-processor by spawning a (worker) domain on it
2. Synchronize using atomic instructions or message passing
3. Share state by distributing frame capabilities

With the system as described in Chapter 6 we have all the needed building blocks to offload work onto the Xeon Phi. What's left is to specify the compute models and the corresponding interface.

7.2 OpenMP

In general, a program's execution can be made parallel or offloaded by explicitly create threads and/or domains to process the data and by dividing the working set among the workers. When doing this standard task manually, it has to be repeated for every new program. Hence, the likelihood of copy-and-paste bugs increases. A generic approach to the problem at hand is specified by OpenMP[2]. The mission of OpenMP is to standardize techniques to provide a high level of parallelism by adding compiler directives (pragmas) and hence remove the burden of managing the threads from the programmer and push the logic into a support library. OpenMP is supported by various companies which form the Architecture Review Board. The standard is defined in the OpenMP API [62]. We won't go into the specification in detail but rather briefly mention the compute-model and its underlying assumptions.

7.2.1 Computation Model

The fundamental rule for any OpenMP program is that it must not depend ot the OpenMP provided parallelism for correctness. Therefore, it is completely valid for a compiler to ignore the OpenMP directives and produce scalar code instead. Listing 7.1 shows a basic example of an OpenMP program.

[2]OpenMP Architecture Review Board, `http://openmp.org/`

```
#pragma omp parallel for
for(int i = 0; i < length; ++i) {
    sum += data[i];
}
```

Listing 7.1: OpenMP: A Basic Use of OpenMP Pragmas

```
/**
* \brief generated function for a parallel section
*
* \param arg pointer to the program argument
*/
void main._omp_fn.0(void *arg)
{
    /* extracted code from for loop */
    /* sum += data[i]; */
}
```

Listing 7.2: OpenMP: Generated Function

The execution of an OpenMP program follows the so-called fork-join programming model [12]: the program's execution starts with a single threaded in other words with the initial thread. At the point where the control flow reaches the start of a parallel section (for instance `#pragma omp parallel` in Listing 7.1) the execution gets parallelized by forking new threads and waiting for them to finish (join). Additional keywords as defined in [62] may be used to adjust the behavior of the OpenMP parallel section.

Assumptions of OpenMP

The fundamental assumption made by the memory model of OpenMP is the existence of a shared virtual address space: each thread has access to *the memory* – the OpenMP term referring to local, global and heap variables – as well as a private view of *the memory*, in other words: a thread local storage. Any of the OpenMP managed threads can any access variables declared and initialized prior the parallel section and hence there is no work involved to share data.

Enabling OpenMP

In order to activate the OpenMP parallelism, two things are required: First an OpenMP compatible compiler such as GCC[3] has to be used to compile the program. In general, a compiler will ignore OpenMP directives unless the appropriate flags are supplied. Secondly the program needs to be linked against an OpenMP runtime support library. GCC uses its internal `libgomp` for this purpose. The support library has to be operating system dependent as certain OS services such as creating threads are invoked.

[3] GNU Compiler Collection

7.2.2 BOMP: Barrelfish OpenMP

In Barrelfish the runtime support for OpenMP is supplied by the `BOMP` library. The library does not support all features specified by the OpenMP 4.0 standard [62]. Compared to the Linux implementation, using OpenMP on Barrelfish requires additional initialization steps such as spanning a domain over a subset of available cores. Further, the threads have to be created on a specific core otherwise the threads will be run on the same core. We have chosen to let the programmer decide when to initialize the library by invoking the corresponding function. The BOMP library satisfies the assumption of a shared address space.

7.2.3 XOMP: OpenMP for Exclusive Address Spaces

Recall the need of a shared address space for OpenMP programs. However, this may be hard to establish even impossible to achieve due to system design. We face such a situation with our heterogeneous system having multiple physical address spaces. Therefore, we need a way to provide OpenMP functionality which does not rely on a shared address space. We propose `XOMP` an eXclusive address space OpenMP support library.

Rationale for the Need of non-shared Address Space

Recall the potential issues with multiple physical address spaces in a heterogeneous system. Establishing a shared virtual address space between two physical address spaces may be not feasible because of mismatching address sizes. Even further, the different base addresses of the page tables makes their reuse impossible and therefore requires copying and adjusting the page tables. In addition to that the mappings must be kept up-to-date.[4] Another point to mention is the lack of flexibility with shared address spaces: a certain virtual address will point to the same byte in memory. If it resides in remote memory, this results in high penalties for accessing memory as shown in Section 5.2. To sum up, a non-shared implementation gives us the possibility to use OpenMP between different architectures and address spaces as well as the flexibility of data locality.

OpenMP with Exclusive Address Spaces

In the following we detail our implementation of a non-shared address space OpenMP support library. The Barrelfish's existing `libbomp` was extended by an additional backend called XOMP – to clarify: XOMP is part of `libbomp` and not a standalone library – which provides the runtime support for exclusive address spaces. The following list outlines the changed characteristics of the new XOMP backend compared to the original `libbomp` implementation:

1. **Library Initialization:** For each OpenMP thread we have to spawn an entire worker domain. In general, spawning a new process is much more heavyweight than creating a new thread. In Barrelfish however, we need to

[4] This involves propagating every new mapping which may occur in response to a `malloc()` and every time a mapping is removed. Besides the mappings, the capability structure needs also to be synchronized as well

allocate a new dispatcher in both cases which evens out the initialization time (refer to Section 7.2.5).

2. **Roles:** The domain started initially is called *master* while the other domains are labeled as *workers*. Designated worker domains act as *gateways*.
3. **Accessing Global State:** With exclusive address spaces, global state has to be shared explicitly. Data accesses from within parallel sections needs to be done with great care to avoid dangling pointers.
4. **Distributing Work:** Instead of creating threads, the task is distributed using message passing to the worker domains.
5. **Barriers:** Implementation using message passing instead of accessing a shared barrier variable.

Even though most of the things mentioned in the list are abstracted by the library, the most important one has to be handled with extra precaution: ensuring access to data. The programmer needs to be aware of the data which may be accessed within an OpenMP parallel section and has to ensure that the corresponding frames must be shared and the pointers to it initialized properly. We ensure this by mapping the shared frame at the same virtual address.

OpenMP in a heterogeneous System

We implemented the non-shared address space variant of OpenMP to work on a shared physical address space and on a heterogeneous system with multiple physical address spaces. Our implementation is based on the software framework described in Chapter 6 and enables offloading to the co-processor using standard OpenMP compiler directives.

We have to emphasize that heterogeneous execution, in other words executing a task concurrently on different micro architectures, does not necessarily work out of the box: first, as we spawn new worker domains we must compile the code for each of the target micro architectures. Compiling a program with different tool-chains most likely generates varying binaries, especially with distinct architectures, resulting in different addresses for the functions. Secondly, we have to do a function address translation as the compiler generates new functions (Listings 7.2 and 7.1) for each parallel section. Last, we need to make sure the data argument pointer is valid in the worker domains.

Function Address Translation When entering a parallel section, the generated code calls into the support library supplying the function address, an argument pointer and the number of threads to be used. We came up with a method for transforming the function address into a target architecture valid representation.

Our mechanism is based on the following observations and assumptions: We observed that the generated functions can be identified by having the tag `_omp_fn` in their symbol name. In addition, we assume the number and order of the generated functions within the symbol table is consistent among the architectures.

An extract of the interface can be found in Appendix C.3. We briefly outline the translation process at this point:

1. **Spawning:** The OpenMP program needs to be spawned with specific spawn flags namely `SPAWN_FLAGS_OMP`. When the OpenMP flag is supplied, `spawnd` scans the symbol table and builds an index of the OpenMP functions by extracting the information out of the symbol table. The index for the binary is stored in the SKB.
2. **Library Initialization:** During the initialization phase of the XOMP backend, master and worker side, a cache of the generated index is obtained from the SKB to speed up future look-ups.
3. **Work Distribution:** The master domain will look up the function index if the target worker is of a difference architecture. It then masks and sends the index instead of the function address to the worker domain. Receiving a masked index, the worker extracts the function number and queries the local cache for the actual function address.

Implementation Variants

With our implementation of the exclusive address space OpenMP based on message passing there is more freedom we on how to structure the flow of control messages during task execution. The following paragraphs name a few possibilities of which we implemented III.

I **Star** We apply the well known star topology with the master in the center connected to all its workers directly. The benefits for this structure includes the ability direct communication with each worker and hence tight control. However, with more workers the number of open connections that needs to be polled also increases. Further any message for memory sharing and task distribution needs to be sent by the master which results in a linear execution.

II **Clustered** This topology groups worker domains into clusters reflecting the underlying system architecture: for each co-processor and NUMA node a cluster will be formed. A dedicated domain of each cluster will act as gateway to talk to the master and potential to other clusters. The number of connections at the master are lower than in the star setup resulting in a more parallel way to distributed and share memory. The gateway domains will receive the requests form the master and forward them to the cluster-local workers. One of the drawbacks of this approach is that the master has no direct connection to the clients.

III **Memory Gateway** The message topology which is currently implemented is based on a combination of the two named above: The master has a communication channel for each client to distribute work. However, each Xeon Phi and the host will form a node in the memory distribution graph (NUMA is currently ignored). The frame capability is sent only to the gateway domains and the other worker in the node will request the memory directly from their local gateway.

IV **Hybrid** Another possibility would be to combine both the shared and exclusive backend: using the clustered approach between nodes and the shared memory implementation within the clusters. The gateways will act as a local master.

The loss of control in variant II and IV is contradictory towards the OpenMP idea of the initial thread. However, applying a hierarchical approach may increase performance because work and memory distribution can be better parallelized.

7.2.4 Library Initialization

The BOMP/XOMP library has to be initialized explicitly by a call to the initialization function of the corresponding backend (Listing 7.3, page 90). The last initialized backend will be active and used by the library. During the initialization the needed threads or domains are spawned based on the supplied arguments.

Use cases such as parallel loading on the host followed by distributing work to the co-processors require the ability to switch between the different implementations. Our interface provides a `bomp_switch_backend()` function to context-switch between the two different backends.

```
/**
 * \brief initializes the shared address space backend of BOMP
 *
 * \param nthreads    the number of threads to activate
 */
int bomp_bomp_init(unsigned int nthreads);

/**
 * \brief initializes the exclusive address space backend of BOMP
 *
 * \param args    pointer to backend arguments
 */
int bomp_xomp_init(void *args);

/**
 * \brief switches the backend to be used
 *
 * \param backend   Backend to activate
 */
int bomp_switch_backend(bomp_backend_t backend);
```

Listing 7.3: Barrelfish OpenMP Initialization

7.2.5 Library Scaling Characteristics

The two backend variants are fundamental disparate in how they are initialized and in their underlying compute model. Before we evaluate the performance of

the OpenMP library we are interested in the overheads introduced for initializing the library, sharing the frames and distribute the work.

Initialization Phase

During the initialization the (master) domain is either spanned to other cores or new worker domains are spawned. Clearly, spawning is expected to be more cost intensive than just spanning. Interestingly the measured initialization times show exactly the opposite as we will elaborate later (Figure 7.1, page 92). Furthermore, we also account distributing memory to the initialization phase. The sharing of frames was expensive but for different reasons which we will explain in Figure 7.2 on page 94.

Spawning of Worker Domains Currently, there is no asynchronous implementation to spawn new domains in Barrelfish therefore the spawning process is rather serial than parallel. Accordingly, the time to initialize the non-shared backend of the library linearly in the number of worker domains to spawn. The library initialization can definitely benefit from asynchronous spawn implementation because each request is sent to another core.

The time difference between spawning local and remote workers is roughly an order of magnitude, to be more precise 8x, a value which we will encounter again later. The estimated time it takes to initialize the library *on our system* can be calculated using the following formula with l, r being the number of local and remote workers respectively:

$$\text{init\_time}(l, r) = 40\text{k} \cdot l + 330\text{k} \cdot r \qquad \text{[Normalized Cycles]}$$

The time measured was from the sending of the spawn request until the master successfully receives a new connection attempt on the supplied Flounder channel.

BOMP Domain Spanning Even though the spanning of a domain does not require new domains to be spawned the process is carried out quite similar: in both cases a new dispatcher has to be allocated and initialized. This is done by sending a message to that core. Once the dispatcher is created the remote initialization function is being executed. With certain blocking operations as the spanning request and the remote initialization the overall process is serialized as well. The resulting average time to initialize the shared backend of the library takes longer than with the non-shared XOMP backend.

Distributing Memory (Sharing Phase) The duration of the sharing process depends on several factors as shown in the graph: the number of worker domains which need to have access to the frame, the size of the frame to be shared and whether it is replicated or not. In contrast to spawning/spanning, memory distribution is mostly asynchronous except for gateway nodes. Figure 7.2 shows two graphs: purely local memory sharing (top) and the distribution of frames between host and Xeon Phi (bottom). Note, the graphs are in log scale.

We see two completely different behaviors in terms of frame sizes: Whereas a single page can be distributed with a constant overhead as seen on the top

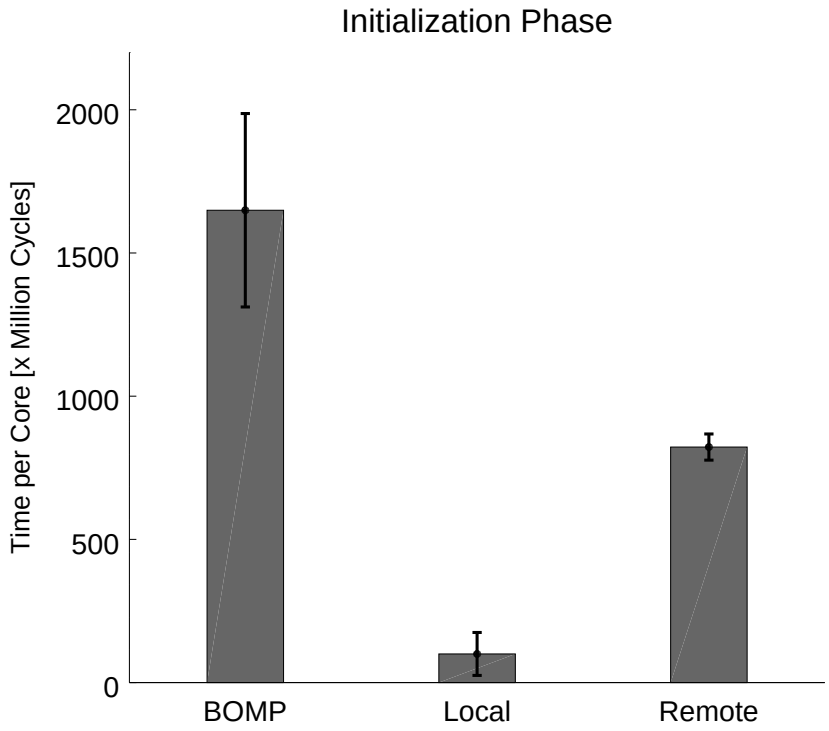


Figure 7.1: Barrelfish OpenMP Library: Initialization Phase Spawn Time

graph the sharing of bigger frames takes longer with each additional worker. The reader may also note the differences between host, Xeon Phi and heterogeneous sharing.

- **4 kB Buffer:** Mapping a single 4kB page occupies just one slot in the page table. In x86_64 there are 512 slots per page table and hence allocating a frame to hold a new page table is scarcely needed. Scaling up to 50 worker domains results in an almost constant distribution time because of the asynchronous implementation and the parallel mapping operations. Note the larger standard error when the Xeon Phi is active: with multiple runs per experiment there will be the situation that a new page table has to be allocated resulting in a map time explosion. Overall, when going over PCI Express (red) we pay a constant overhead as the first memory transfer needs to go through the Xeon Phi driver.
- **32 MB Buffer:** Comparing to a single 4k page mapping a 32MB buffer requires 8192 page table entries and 16 page directory slots. Thus, mapping as 4k pages results in 16 requests for memory to hold the new page tables. Each request is handled by the memserver which turns into a serialization point of the operation. The total request count grows in the number of workers and so does the total execution time. The effect is even stronger when messaging latency is bad as it is on the Xeon Phi (green). Furthermore, we obtain that distributing over PCI Express scales quite nicely: with two Xeon Phis and the host a total of 70 workers ends up at 30 domains per Xeon Phi. Comparing red at 70 with green at 30 we have about the same value. With larger frames, the mapping dominates the replication time.

Overall we can say that there is a relatively high overhead for sharing memory. With an increasing number of workers and frame size the proportional overhead of message passing goes down and the time is dominated by the mapping operation. To overcome the bottleneck, the use of large pages is highly encouraged as soon as their support is integrated into Barrelfish. In addition, having a single memory server clearly acts as a serialization point and thus the use of a distributed version is suggested. We provide additional data on the mapping times in Section C.1 of Appendix C.

Work Phase

The steps needed to initialize and terminate a work phase are very different: in the shared backend new threads are created to execute the tasks whereas in the non-shared backend a message is sent to the worker domain. We are interested in the raw time to distribute work and how it scales with respect to an increasing number of worker domains/threads.

As a workload for this benchmark, we have chosen a simple `parallel for` loop on which every thread does a constant number of iterations; in other words the total amount of loop iterations scales with the number of threads. Figure 7.3 shows the normalized results of this benchmark and the corresponding absolute values can be seen on Table 7.1. The values shown on the graph is the overhead

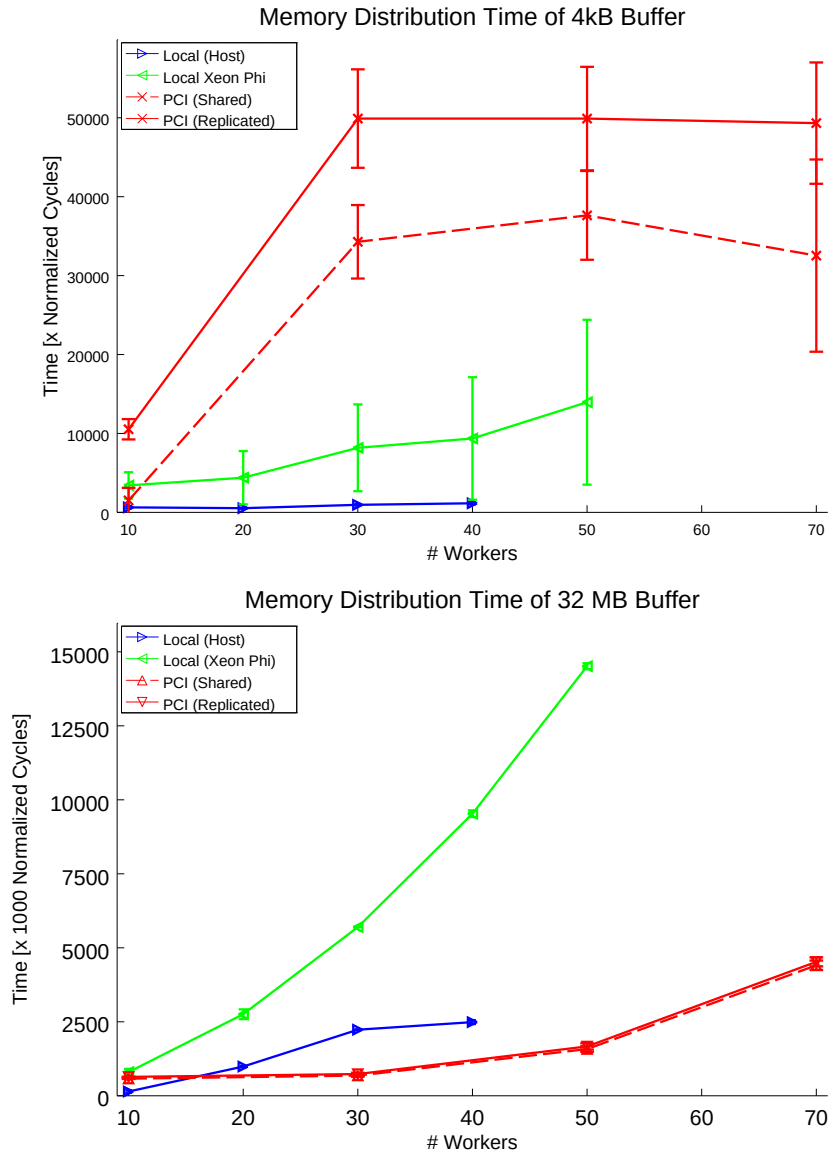


Figure 7.2: Barrelfish OpenMP Library: Sharing Phase

# Threads	Host		Xeon Phi	
	BOMP	XOMP	BOMP	XOMP
2	13422	15698	45560	36798
10	4484	5022	4888	3775
20	3297	3669	4186	3395
30	3777	3189	4020	3331
40	3960	2959	3942	3286
50	-	-	3900	3249

Median time in non-normalized, local cycles to distribute work per thread

Table 7.1: Barrelfish OpenMP Library: Work Distribution (Local)

per thread i.e. total running time divided by the number of workers. We use the 99-percentile values on the graph. The medians can be seen in Table 7.1.

Local Work Distribution Running the benchmark purely local shows a decreasing overhead per thread as the number of workers grows. On the Xeon Phi, there is no statistically significant difference between the shared and non-shared implementation. On the host, message passing seems to be the better option with more worker domains. In general the trend on both nodes is the same: reduced overhead per thread with an increasing number of workers. The results are consistent with previous benchmarks as the Xeon Phi has a worse message passing latency than the host resulting in a higher overhead. Overall there is a median overhead that needs to be paid per thread of about 4000 normalized cycles.

Work Distribution over PCI Express Recall the message passing benchmark: when going over PCI Express we have encountered a significant increase in terms of message passing latency. With an growing number of worker domains on the Xeon Phi not only the Flounder channel count at the master is increased but also the share of channels going over PCI Express among those goes up. The polling and involved messages issue many PCI Express transactions. This results in a higher overhead per thread as shown on Figure 7.3 (bottom). Compared to the local measurements, the 99-percentile value increases with more present workers (green) caused by more slow memory sharing over PCI Express.

7.3 Use Case Example: Matrix Multiplication

To demonstrate the compute-performance and the correctness of the OpenMP library for Barrelfish we use a matrix multiplication program. The algorithm takes two square matrices, multiplies them and calculates the sum of the diagonal. To check for correctness, summed up value is written to a shared memory location and validated against the expected value. The implementation is based on a cache-optimized $O(N^3)$ nested loops algorithm shown in Listing B.2 in Appendix B.

The rationale behind taking the matrix multiply as an example is the following:

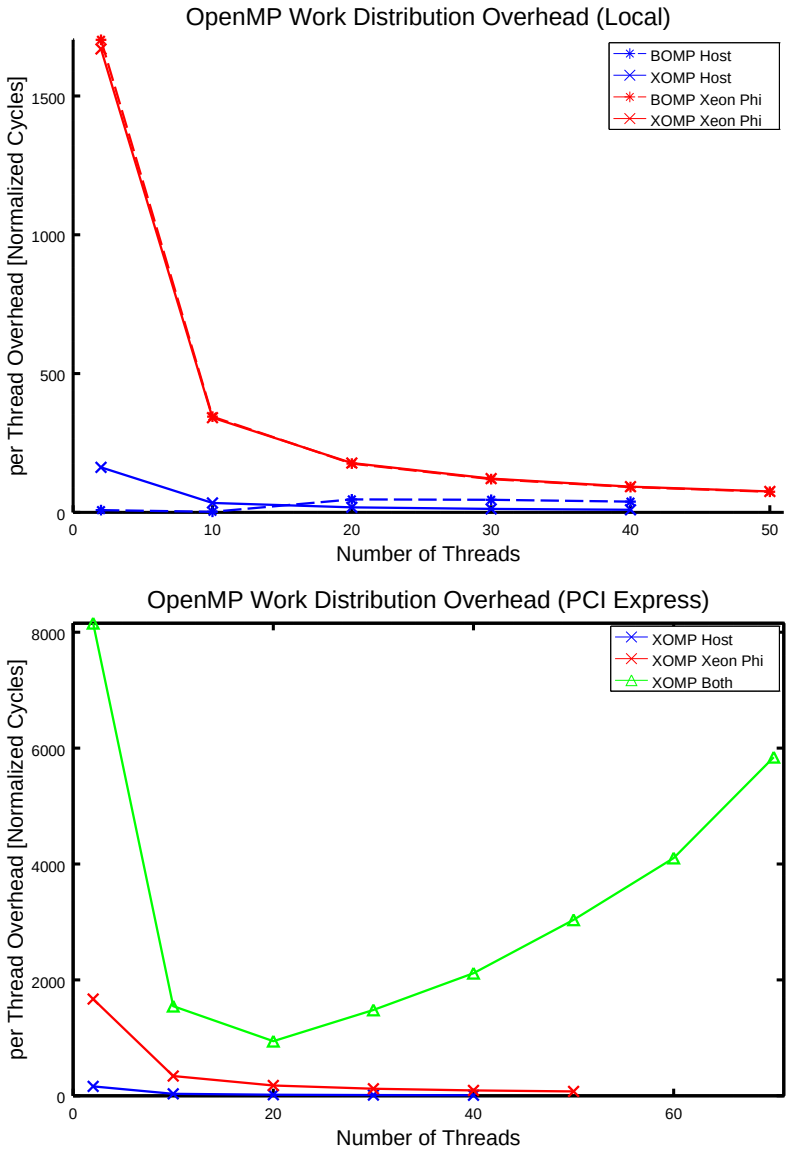


Figure 7.3: Barrelfish OpenMP Library: Work Distribution

on the one hand, its work is evenly distributed and on the other hand there are may applications, signal processing for instance, which use matrix multiply. We are interested in how the program execution scales when parts of the matrix multiplication are offloaded onto the co-processor.

7.3.1 General Benchmark Description

For any of the benchmarks we used the parameter values as shown on Table 7.2 below. The master domain runs on the host. The term remote workers refers to worker domains spawned on the Xeon Phi and local workers to the domains running on the host. For any of the results only the execution time of the OpenMP loop was taken into account. We did not make use of special hardware features such as the 512-bit vector registers of the Xeon Phi and use just scalar integer operations.

Key	**Value**
Matrix Dimensions	1500x1500
Element Type	long (64-bit)
Matrix Size	17.16 MB
Local Workers	10
Remote Workers	55 (per node)
Data	Replicated
Repetitions	25
Hardware Features	None

Table 7.2: OpenMP Matrix Multiplication Settings

7.3.2 Baseline: Local Benchmarks

To get an idea how the different Barrelfish OpenMP library backends scale on the host and the Xeon Phi we take local benchmarks as a baseline for the heterogeneous benchmarks. The outcome of the running time compared to the single threaded run can be seen in Figure 7.4 or on Table 7.3. With the 10 threads used we reached the expected performance improvement by a factor of 10. Further, we see that the Xeon Phi is about factor 9 slower than the host for doing the same task.

Location	**Single**	**OpenMP**	**Speedup**
Host (BOMP)	8400316	712342	11.79x
Host (XOMP)	8400316	885877	9.48x
Xeon Phi (BOMP)	70010484	7343763	9.53x
Xeon Phi (XOMP)	70010484	6980953	10.02x

(Normalized Cycles)

Table 7.3: OpenMP Matrix Multiplication Local Parallelization

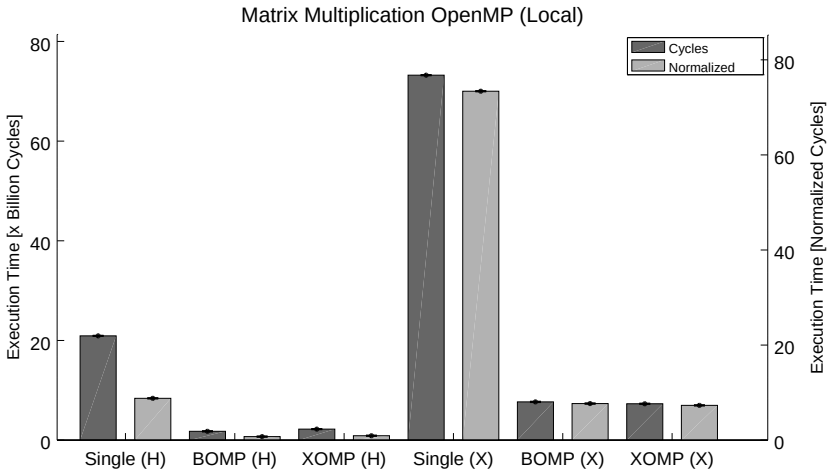


Figure 7.4: Local Matrix Multiplication using OpenMP

7.3.3 Effect of Data Replication

The memory throughput and latency benchmarks revealed the huge penalties when accessing remote memory locations. To measure the drawback of non-local data buffers the exact same benchmark has been repeated with a matrix buffer residing in the system memory. Thus, the Xeon Phi worker domains had to access the data through PCI Express. The replicated version is about two orders of magnitude faster (1.4 vs 149 Billion cycles) than the non-replicated version. As shown in the Library initialization above the replication of data using the DMA engine adds a reasonable small overhead to the memory distribution process because it has only to be done once per node. Compared to the penalty for accessing the shared frames on the system memory, it is definitely required to have the (read-only) data replicated.

7.3.4 Matrix Multiply using Heterogeneous OpenMP

So far, we have run the matrix multiplication example locally and achieved the expected speedup by the number of threads. How does this scale when we incorporate the Xeon Phi co-processors and distribute the work? The results of this experiment series can be seen on Section 7.3.4. The following subsections discuss the different outcomes.

Try 1: Host and one Xeon Phi

To the 10 threads on the host we add a Xeon Phi with its 55 additional threads. Each of the threads now has to do one 65th of the work (compared to a 10th of the local benchmark). Under normal conditions the expected speed up would be a factor of 6 to the local baseline. However, compared to the host-only benchmarks with 10 threads the task is executed significantly slower. Why is that the case?

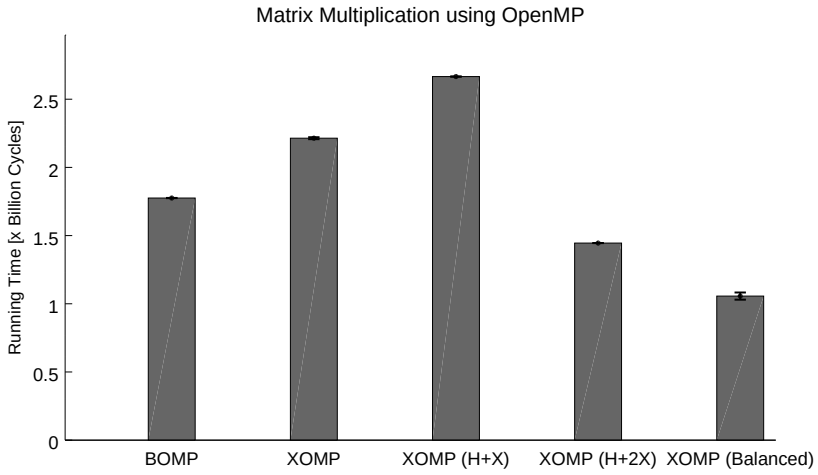


Figure 7.5: Matrix Multiplication on Heterogeneous System using OpenMP

Recall the Xeon Phi local benchmarks. To get an accurate estimation of the expected running time, we have to take the slowest element of the computation into consideration: 10 threads on the Xeon Phi had about 7 Million normalized cycles. Thus, doing just about one 6.5th of the work results in approximately 1.08 Million normalized cycles. Multiplying hat with the normalizing factor of the host we get about 2.6 Billion cycles which is in the range of the measured result. Doing the same calculations backwards we can estimate the number of threads needed such that we acheive a speed up compared to host-only variant. We obtained a value of at least 90 threads again the factor the Xeon Phi was slower.

Try 2: Host and two Xeon Phi

Even though it is possible to have more than 228 hardware threads on a single Xeon Phi we restrict the number Barrelfish cores to the number of physical cores and go with 55 worker domains per Xeon Phi. This leaves us two cores for system services and the Xeon Phi driver. We have a total of $10 + 2 \times 55 = 120$ threads which is above the calculated threshold.

Doing the same calculation as in try 1 gives us an estimate of 1500 Billion cycles which is about that what we measured in the experiment. Even though we went from 10 threads to over 120 did not result in the performance increase we hoped to see: the Xeon Phi and its poor scalar integer performance seems to be the bottleneck. But can we do better?

Introduction of vTreads

Obviously, when increasing the total number of workers to distribute the task, not only the Xeon Phi threads have to do less work, but also the 10 local threads on the host. By estimating the time it takes for a tread on the host to finish its

work based on the local measurements we expect the host threads to be done after 184 Million cycles. Hence, the host is waiting for more than a Billion cycles for the completion of the workers on the Xeon Phi.

Depending on the offload-paradigm, this spare cycles could either be used for other tasks or we need to find a way to balance the work load more. We estimated that the Xeon Phi needs about 9x times more time to execute the same task. We introduce virtual threads (vThreads) to OpenMP that tries to balance out the compute-power differences between host and co-processor in a static OpenMP schedule. The idea behind vTreads is to have the faster workers for a mini-cluster which does the work of multiple threads: instead of just getting a single OpenMP thread ID a worker with enabled vTreads gets a contiguous range of thread IDs depending on the number of assigned vThreads. This can be implemented using the code in Listing 7.4.

```
for (int i = 0; i < num_vtreads; ++i) {
    omp_function(omp_data);
    thread_id++;
}
```

Listing 7.4: OpenMP vThreads

Try 3: Host and two Xeon Phi (Balanced)

We re-run the benchmark with two Xeon Phis and add 8 vThreads for each of the host worker domains to balance out the differences. The master domain does not get vThreads. This results in $1 + 9 \times (1 + 8) + (2 \times 55) = 192$ threads. Doing the same calculations as above we would expect around 930 Million cycles. However, the measured total runtime was clearly above 1000 Million cycles in average – the fastest run (minimum) was about 950 Million. The breakdown of the running times into remote and local revealed that the host threads are now taking longer than the ones on the Xeon Phi. An optimal value would be somewhere between 7 and 8 vThreads.

Future Try 4: Activating Special Hardware Features

As stated in the benchmark description, we did not activate special hardware features such as `AVX-512` of the Xeon Phi[5]. With the nature of matrix multiplications this is expected to give to a speed up. However, such hardware features may not always be usable in various workloads.

7.4 Conclusion

The experiments about distributing work among the host and Xeon Phi worker domains showed that even with vast amounts of additional cores the performance gain is not as expected. The Xeon Phi's integer calculation performances

[5] `AVX-512` is available as compiler intrinsics from version 4.9 and later

and memory latencies have a too big impact on the performance. Nevertheless, depending on the offload-paradigm the use of co-processors and accelerators still pays off as work can be split up and/or done more efficiently. Moreover, the utilization of the host resources is decreased.

In addition, we accounted only the raw execution times of the matrix multiplication OpenMP loop and left the initialization phase and the memory sharing phase aside which takes significantly more time than the actual execution. There may be an additional performance gain implementing the missing prefetch instructions and using the extra wide vector registers. This is left for future evaluations.

Moreover, implementing the hybrid approach reduces the required effort for distributing memory. The capabilities have only been transferred to the gateway domains – or the node-local masters in that case.

Another open problem is how can we make sure, that data updates from the worker domains are propagated to the other nodes and especially to the master domain. Explicit `scatter` and `gatter` operations may be needed.

To sum up, we have shown how our proposed system can be used to distribute a matrix multiplication using a non-shared memory implementation of OpenMP among heterogeneous processors using bulk transport and message passing.

Chapter 8

Future Work

The current state of the system left several research questions and engineering tasks open. This Section summarizes possible anchor points for future work such as a truly heterogeneous operating system, virtual devices and bulk transport implementations.

8.1 Towards One System

Intel's solution for Xeon Phi systems is to have a separate Linux instance running on each Xeon Phi which is treated as an independent execution environment. Hence, the resulting system architecture of a single machine looks like a small networked cluster or a virtual environment setup.

Our proposed system is slightly more integrated in terms of OS service accessibility: the co-processor instances make use of the system wide services such as the global name service. However, its integration is still rather loosely coupled. In the end, the Barrelfish's OS nodes should work more in a coupled way: for instance, the cores and memory of the Xeon Phi should be managed by the host as it is done with the local resources including memory management. In any case system services such as name service and `ramfsd` should be existing only once for the entire machine or alternatively in an appropriate distributed design.

Designing and developing of such a system spanning host and co-processors arises several questions. Some of which are purely an engineering task while others need a more in depth investigation of the problem:

- **Resource Management:** Some co-processors as the Xeon Phi may bring their own local memory. This leads to the question on how these local resources are managed and by whom. Basically there are two approaches: either the co-processor manages its local resources or its owner the host takes care of the resources. Both ways come with their own

trade-offs which have to be elaborated. With any possible design it must be feasible to get a locally valid representation of capabilities. In the end it should be possible to allocate memory in a specific address space.

- **Naming and Addressing:** In Barrelfish IREFs refer to exported services and are assigned by the kernel monitors. In a heterogeneous environment the service may be running on any of the nodes. Therefore, we need a way to seamlessly refer to the exported interfaces on a remote node. The runtime system must be aware of the location because setting up a communication channel may have to be done differently. This could be handled by an additional Flounder backend.

- **Task Migration:** Migrating a task (or domain) from one core to another is highly non trivial: several things need to be ensured such as the validity of the domain's CSPACE or the seamless operation of open Flounder channels[1] for instance. Saving energy by consolidating machines is a standard approach in data centers and therefore we may want to do the same with co-processors: shutting them down when they are no longer needed. This implies we will have to migrate all task from one co-processor to another or even to the host. Migrating a task between physical address spaces even architectures is a challenge which may even be impossible to some extent. The question that arises is how to properly encapsulate the state[2] of a domain and how to resume its execution in a different physical address space.

- **Multiple Address Spaces:** How does an operating system deal with multiple physical address spaces in general especially when they are non-static. Refer to Section 8.3.

- **Non-static Physical Addresses:** As with the SMPT of the Xeon Phi a physical address may no longer be static but be mapped onto another physical address and this mapping can change during runtime. What does a physical address really mean now? To what extend can hardware virtualization technology techniques such as extended page tables of virtual machine monitors be applied?

- **xNUMA:** Systems with NUMA nodes have been around for quite a while. With the Xeon Phi the resulting system can be seen as an extension to normal NUMA nodes as a new type of memory becomes available (Section 5.1). The system needs to be aware of the highly skewed architecture and memory access latencies.

- **Heterogeneous Scheduling:** Managing a system with asymmetric cores especially supporting different architectures poses several challenges on the operating system towards its scheduling decisions. The OS can no longer simple dispatch tasks to any of the available cores. Wrong scheduling decisions may lead to poor performance or even unsupported instruction traps in the worst case. The optimal schedule may even depend to the current program counter as certain parts of an algorithm may benefit from special hardware features.

[1] especially when the used backend has to be switched e.g. LMP → UMP

[2] including the stack, static variables, CSPACE, communication channels and so forth

- **System Architecture:** Having multiple physical address spaces poses the question on how the structure of the system looks like in particular whether a hierarchical decomposition may be favorable compared to a flat design. Is a clustering of resources (co-processors, NUMA nodes) with multi-hop communication beneficial or can every resource being treated equally (i.e. each core of the co-processor is equivalent to a core on the host). How does the system scale in terms of memory consumption, message channels and state replication?

- **Execution Targets:** In a heterogeneous system an architecture specific version of each executable must be available either separately or integrated into one single file. It's the task of the operating system and/or the runtime environment to select the correct code to execute. How can such a multi-architecture execution environment be supported in case of spanning between co-processors and host? Further, how can multi architecture support be integrated into executables and how can the control flow be made hardware dependent?

To sum up, designing a heterogeneous system with multiple physical address spaces is highly non-trivial and poses may open research questions. Techniques used in architecture independent languages such as Java may be worth investigating.

8.2 Xeon Phi as a Collection of Cores

This topic is related to the previous one but we would like to stress some additional points about processor cores here. In the current state, the host treats the Xeon Phi as an ordinary PCI Express device with its driver and programming model. Hence, the host is not directly aware of the cores available on it: The Xeon Phi is treated as an independent compute node which manages its resources on its own. With the points mentioned in Section 8.1 the available cores should be exposed to and managed by the host. This implies that the host system must be able to handle late discovery of new cores of a different architecture. Further, the way the cores are managed must be highly scalable as the amount of cores[3] literally explodes. Obviously accessing shared data structures do not scale as remote memory access involves huge penalties.

A word on Simultaneous Multi-Threading Intel introduced simultaneous multi-threading (SMT, or *hyper threading*) to keep the pipeline full in the event of stalls caused by cache misses or other events preventing the processor from executing the next instruction. Current server and desktop CPUs have two hardware threads per core showing up as two logical cores to the operating system. The Xeon Phi has four such logical cores per physical core. With the design of Barrelfish each of the cores would have its own CPU driver running. This arises the question to what extent such logical cores should be treated real cores or if they should be better ignored. Alternatively having a single CPU

[3] On our system, the two Xeon Phi co-processors bring $2 \times 4 \times 57 = 456$ new cores / threads

driver per physical core managing all of its local hardware threads. Currently Barrelfish is not capable of booting all cores of the Xeon Phi (see Section 8.4).

8.3 Multiple Address Spaces

With the design of the Xeon Phi the assumption that a physical address is unique is no longer true. Recall the points stated in Section 4.3.3: a physical address may now refer to two different memory locations and on the other hand two different physical addresses may point to the same memory location. A physical address space thus behaves similar to a virtual address space to some extent. The following points arise when we try to unify physical address spaces:

- **Addressing:** How can a single memory location be addressed uniquely such that the address is unambiguous at any time and can be transformed into a locally valid representation.
- **Address Spaces:** What forms a physical address space and how is it identified. Who is assigning the potential identifier to the address space?
- **Translation:** How are addresses of a remote address space translated into the local address space. What happens when such a mapping changes during runtime? How about security: should every address space be able to translate or should the translating function better be protected by means of capabilities?
- **RDMA** With technologies such as RDMA how is the memory of a remote address space[4] referred to especially how are possible address space identifiers assigned to independent, networked machines. Is there any difference to multiple machine-local address spaces?

8.4 Barrelfish: Explore Scaling Behavior

Booting Cores: Hard Limit With the Xeon Phi we had the finally the possibility to boot Barrelfish on a *big* machine with up to logical 228 cores. However, the attempt to boot all of the 228 hardware threads failed due to memory constraints: The total amount of RAM consumption for kernel, Monitor and `spawnd` for any additional cores is about 30 MB[5]. The created messaging channels between the monitors are not included in this 30 MB. This implies, that after about 138 cores, the memory consumption is near 4GB and hence allocating memory in the low 4GB address segment is no longer possible. Further, with that memory consumption it is not possible to boot all of the 228 cores as it would require about 6.5 GB of memory just to hold the basic system domains and with only a total 6 GB of memory available. Therefore we need to investigate what uses so much memory.

[4] in this case the one of another machine

[5] there is some fragmentation to be expected, as capabilities have to be a power of two.

Booting Cores: Setup Time A brief evaluation of the Monitor initialization times (Section C.2) suggests a quadratic growth of the boot up time[6] for each additional Monitor present in the system. It may be debatable whether or not such a growth in initialization time for large core counts is acceptable or if the whole process needs to be partially revised by clustering the Monitors into groups and have multi-hop communication between the clusters.

Boot Modules Currently all the modules to be booted are specified in the `menu.lst` file. Thus, every new module increases the total size of all modules to be loaded especially with huge ones such as the Xeon Phi multiboot image. After a certain total size of the modules exceeds the machine goes into a boot loop, which could be replicated on various machines. To avoid this problem, the `menu.lst` should only contain the core domains needed for providing essential system services plus the network stack. Additional domains should be loaded on demand over NFS for instance as it is already done with the Xeon Phi image.

Memory Mappings The benchmarks have revealed a monotonically increasing mapping time with the number of mapped virtual memory regions in the virtual address space. With any new mapping, the number of VREGIONS increases and so does the representation of the page tables in user space. In addition to that the capability tree is also getting bigger. All this results in a steadily increasing mapping time with every new mapping. The effects are expected to get even worse if the data structures no longer fit in the L1 cache.

8.5 Networking to the Xeon Phi and Host VFS

As explained in Chapter 6 the Xeon Phi multiboot image contains all the modules which can be spawned on the co-processor. Obviously the number of modules determine the size of the image. A reasonable approach would be to have only the core modules inside this multiboot image and have additional modules requested form the host[7]. This can be achieved with various approaches:

- **Networking:** The host exports a networking service or a virtual network device to the Xeon Phi. The local `ramfsd` service requests the additional modules over NFS. That way the host is not directly in control of the modules to be loaded.
- **ramfsd:** The `ramfsd` service of the host exports its interface to the Xeon Phi and allows requests from the Xeon Phi. This involves adapting the bulk transfer mechanisms to work over PCI or to do a pre-allocation of the buffer on the Xeon Phi and directly DMA into it.

[6] from call to `main()` until entering the messaging loop

[7] note that this applies only to the current system. Having a single system as described in Section 8.1 makes this no longer needed. The following points will have to be solved anyway

8.6 Extending Flounder

With our solution to bootstrap Flounder over PCI Express we lose the ability to send a capability. Even though we provide means to transfer capabilities between address spaces automated transfers such as in standard Flounder are not supported. In addition to that integrating DMA based bulk transport mechanism into Flounder may lead to interesting use cases for copying larger buffers.

8.7 DMA Library

The current DMA library provides only mechanism for memory copy operations and polled operation mode. Bringing interrupts for larger transfers, 1MB and more, may be a better option. Further implementing the missing transfer types may enables for faster `memset()` implementations.

8.8 OpenMP

The support of OpenMP in Barrelfish is not at all complete. There are several open points to discuss and/or implement concerning the OpenMP library.

- **OpenMP Library:** The current OpenMP implementation of Barrelfish's `libbomp` lacks the support of most schedule types except for static schedules. Especially the dynamic schedule is of need, as the processing capabilities of host and co-processor differ. Further other language features such as a teams, nested parallel sections and the internal control variables need to be properly implemented according to the OpenMP specification. With the new OpenMP 4.0 standard, explicit offload pragmas are introduced which could be a handy feature to have for controlling the execution and attack the problem with different architectures.
- **Implicit Sharing:** It may be debatable whether or not sharing of state should done explicit. The problem with non-shared address spaces is mostly caused by having pointers to memory which is not mapped in the domain. A hook in `malloc()`'s morecore may do the trick by sharing the requested memory every time a new frame gets allocated. The sharing itself could be deferred but OpenMP parallel sections must not start before the sharing is completed. Despite possible drawbacks such as lack of replication control, this would solve the problem of the void pointer argument of the generated work function.
- **Replication:** Currently a frame has to be replicated explicitly. However, it is desirable that only the parts are replicated that are really accessed by the workers on the node. Further processed data should also be merged back to the master node correctly. This involves additional compiler intrinsics to figure out which data was touched and has to be updated on the other nodes. Using a double buffer approach data could be transferred in background using DMA while processing is done on the secondary buffer.

- **Distributing Memory and Capabilities:** Since master and workers theoretically should have access to each other's resources it may be beneficial to have a way to remotely control the worker's CSPACE. This would enable the master to update virtual memory mappings and so forth. However, as workers are full-blown domains this would stress the notion of capabilities to its extreme.
- **Control Channels:** In the current implementation the master domain has a control channel to each of the worker domains. Hence, the number of channels grows linearly in the number of workers. To reduce the polling overhead a hierarchical structure may be favored: for each compute node[8] a dedicated worker acts as a gateway to the master. The gateway will receive a certain thread range which is the further distributed among the node-local workers which may even use a shared memory approach.
- **vThreads:** Using various kinds of processors simultaneously inevitably leads to the case that some cores are much faster in executing tasks than others. In our approach we balanced the difference by introducing of vThreads which solved the problem partially. However, the estimation of the vThreads count should be done automatically during library initialization rather than hard coding the value. Alternatively, this can be done either by implementing a dynamic schedule which tries to distribute the work evenly.

8.9 Bulk Transport over PCI Express

With the possibility to share frames and do DMA transfers the system provides a basic support for bulk transfers. However, extending the bulk transfer framework as proposed in [1, 11] would provide more flexibility and security enforcements as described in their trusted/non-trusted model. Based on the findings in Chapter 5 the DMA engine certainly will have to be used leading to a similar model as with the bulk transport over Ethernet backend. In contrast to the Ethernet backend, there will be more control where to place the buffer and the problems with receive buffers as stated in [1] may disappear. In any case, a protocol needs to be applied to make sure that the buffer contents are not changed while the move/copy operation has not finished yet.

8.10 Xeon Phi Hardware Features

Currently, special hardware features such as `AVX512` are not used. The Xeon Phi may be considered as a vector processing device and hence the extra wide vector operations should be used for optimal throughput. Additionally, the Xeon Phi is able to send MSI-X interrupts to the host and other Xeon Phi co-processors. Enabling interrupts to enhance message passing may be beneficial for PCI Express crossing channels.

[8] Host and Xeon Phi, possible also NUMA node

8.11 VirtIO

The implementation attempt of a VirtIO conforming support library showed several issues (recall Section 6.10.1). Certain elements of the programming model and its assumptions seem not to be the best option in a multikernel operating system. Based on these insights we may propose an additional message passing based transport layer in addition to the ones specified in [78]. Essentially the device register representation has to be transformed into a message passing interface. In addition to that whether or not parts of the virtqueue are also transformed into message passing is left as an open question at this point.

Appendices

Appendix A

Memory Access Benchmarks

This appendix contains additional information about the conducted memory access benchmarks of Chapter 5.

A.1 Xeon Phi Caches

The total combined cache size on the Xeon Phi is $512kB \times \#cores$. This adds up to 28.5MB in our case. However, due to its design that total value is only available if every core runs different applications which access distinct parts of memory. In the worst case, if all cores execute the same program and access the very same data the effective total combined cache size for all cores is 512kB. To sum up, a cache line can only be present in one of the cores. The Xeon Phi Systems Developer Manual [20] gives more information about the caches and its protocols.

A.2 Memory Latency

A.2.1 Experiment Description

The experiment setup contains circular linked lists of different sizes residing in the target memory regions. There are two types of lists as shown in Listing A.1:

1. **Cache Friendly:** The list is basically an array of pointers and initialized as such: the next element is located right next to its predecessor.
2. **Randomized:** The list contains cache line sized elements linked in a random way such that every access to the next element is expected to miss

the last level cache and the value has to be fetched from main memory. The list is randomized using Knuth shuffle.

```
/// cache line sized list element for randomized access
struct elem {
    struct elem *next;
    uint8_t pad[CHACHE_LINE_SIZE - sizeof(void *)];
};

/// pointer sized list element for cache friendly access
struct celem {
    struct celem *next;
};
```

Listing A.1: List Element Description

To minimize the loop overhead, each loop is manually unrolled with preprocessor macros. Each loop iteration accesses 1000 elements. The time measured for the entire loop execution is then divided by the number of elements traversed in the list to get the value of a single access. The implementation can be seen in Listing A.2. A the entire code of the benchmark can be found in `usr/bench/mem_latency`.

```
/**
 * perform a single run of the benchmark on a given buffer
 */
static cycles_t run_benchmark(void *buffer,
                              volatile void **ret_elem)
{
    volatile struct elem *e = buffer;

    cycles_t tsc_start = bench_tsc();
    for (uint32_t i = 0; i < LOOP_ITERATIONS; ++i) {
        NEXT_1000(e);
    }
    cycles_t tsc_end = bench_tsc();

    *ret_elem = e;

    return calculate_time(tsc_start, tsc_end)
                        / (LOOP_ITERATIONS * 1000);
}
```

Listing A.2: Measuring Memory Latency

A.2.2 Experiment Parameters

The used benchmark parameters can be seen on Table A.1 below. The chosen values are based on the hardware specifications of the described system (Section 4.1. We have chosen the working set sizes to be based on the available cache

sizes: less than L1 cache for the cache-friendly access pattern and a multiple of the last level cache for the randomized access pattern.

Parameter	Value
Working-set Size (Random)	200MB (Host) or 228MB (Xeon Phi)
List Size (Random)	3.2M (Host) or 3.7M (Xeon Phi)
Working-set Size (Friendly)	8KB
List Size (Friendly)	1024
Repetitions	1000
Loop Unrolling Factor	1000

Table A.1: Experiment Parameters: Memory Access Latency

A.3 Memory Throughput

A.3.1 Experiment Description

The benchmark consists of copying a data from one buffer into another buffer while we measure the time to copy and calculate the throughput based on the buffer size divided by the time. In order to have consistent experiments we used the benchmarking functionality of `libdma` which supports `memcpy()` and DMA based transfers. The benchmarks can be enabled by setting the switch in the `dma_benchmark` header file:

```
1  /* dma/dma_benchmark.h*/

3  /// DMA Benchmark control
   #define DMA_BENCH_RUN_BENCHMARK 1
```

A.3.2 Experiment Parameters

The used experiment parameters can be seen in Table A.2 below. DMA transfers had to be done at cache line granularity and hence the minimum transfers size is 64 bytes.

Parameter	Value
Working-set Size	64 Bytes to 128 MB
Repetitions	500

Table A.2: Experiment Parameters: Memory Throughput

A.4 Memcopy Closeup

Compared to DMA transfers `memcpy()` was too slow to see a potential difference on the same graph. We show a closeup of only `memcpy` transfers (Figure A.1).

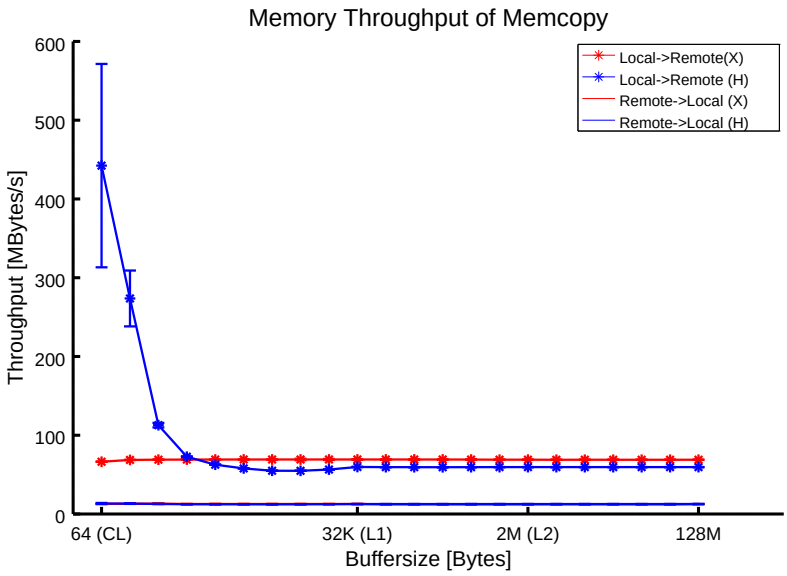


Figure A.1: Memcopy Throughput

The graph displays the throughput of copying buffers over PCI Express either local to remote or vice versa measured on the host and on the Xeon Phi.

First of all, we notice the peak of the host copying buffers from its local main memory onto the co-processors GDDR. The peak may be caused by a cached write in, other words, the actual write was not always done immediately resulting in a higher perceived throughput and standard error. With more data to copy the, write cache gets filled up and hence has to be executed more often resulting in a lower transfers rate.

Comparing the transfer speeds with bigger buffer sizes we get a more stable result and clearly see differences in the copy direction. Transferring data from local to remote memory is always faster even though the actual copy direction changes depending on who initiates the transfer. This can be explained in the way it gets translated into PCI Express transactions. A remote write involves sending a write message with the payload while reading is like sending a read request and waiting for the data.

A.5 DMA Driver Overhead

In our benchmarks we have considered the raw DMA performance by measuring directly at the driver. Because domains need to use the DMA service offered by the driver domain to issue requests there is a certain overhead introduced for the message passing. Figure A.2 shows the comparison between the time at the driver (dotted line) and the time at the client domain (solid line) for each of the tree DMA services. We get a almost constant overhead which is introduced by Flounder, request handling and book keeping of about 30k cycles. With an

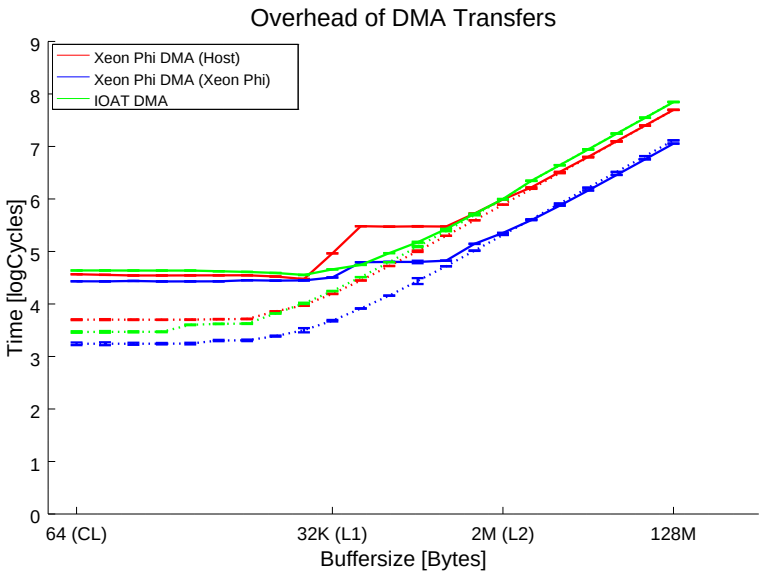


Figure A.2: DMA Driver Overhead

increased transfer size the actual transfer dominates hence after about 512kB we see that the dashed and solid lies match.

A.6 Message Passing

A.6.1 Experiment Description

The message passing latencies were measured using the `ump_latency` for local measurements and `xphi_ump_bench` for PCI Express crossing benchmarks. The Xeon Phi benchmarks can be found `usr/bench/xeon_phi_ump`.

A.6.2 Experiment Parameters

Buffer placements can be controlled by enabling the preprocessor defines in `benchmark.h` of the Xeon Phi UMP benchmark. Table A.3 shows the relevant benchmark parameters.

Parameter	**Value**
Backen	UMP (polling)
Payload	0 Bytes
Repetitions	10000
Core Number	6 (host), 5 (Xeon Phi)

Table A.3: Experiment Parameters: UMP Latency

A.6.3 UMP Latency Distribution

Because message passing is of such importance we are interested if there is a difference in the roundt trip times between different cores. Figure A.3 shows the distribution of the measured RTTs between core 0 and any other core in the system. On the host we nicely see the increase in the round trip time when we cross the NUMA boundary. With the ring-like interconnect on the Xeon Phi we expect to have varying latencies with respect to the distance between the cores. However, the measured latencies have no statistical significant difference which may correlate to the core distances of the die. Where as on the host we get a better latency when we are on the same core sending to a different hardware thread.

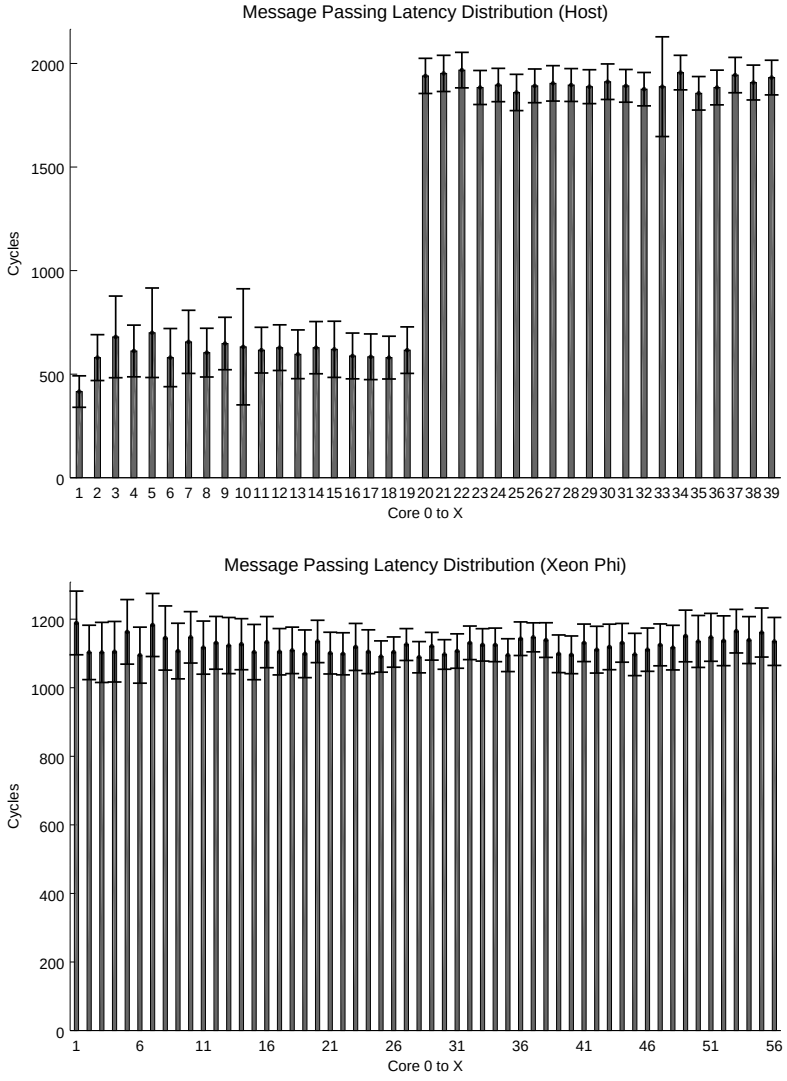


Figure A.3: UMP Latency Distribution

Appendix B

OpenMP Benchmark

B.1 Library Initialization

The library initialization benchmarks were conducted using the library integrated event tracing. The library measures the times for each worker and it also aggregates the measurements for local and remote workers.

The benchmark facility can be activated by compiling the library with the respective switch and a call to the enable function as shown in Listing B.1. Notice, that the number of threads and the number of runs must match the benchmark settings. The different events to measure can be activated by passing the appropriate flags.

```
/* <xomp/xomp.h> */

  /// enable the benchmarking facility
  #define XOMP_BENCH_ENABLED 1

/* <xomp/xomp_master.h> */

  /// flags which events to measure
  #define XOMP_MASTER_BENCH_SPAWN   (1 << 0)
  #define XOMP_MASTER_BENCH_MEM_ADD (1 << 1)
  #define XOMP_MASTER_BENCH_DO_WORK (1 << 2)

  /**
   * \brief enables basic benchmarking facilities
   *
   * \param runs    the number of runs of the experiment
   * \param flags   flags which benchmarks to enable
   *
   * \returns SYS_ERR_OK on success
   */
  errval_t xomp_master_bench_enable(size_t runs, size_t nthreads,
                                    uint8_t flags);

  /**
   * \brief prints the results of the enabled benchmarks
   */
  void xomp_master_bench_print_results(void);
```

Listing B.1: Enabling the BOMP Benchmark

B.2 Matrix Multiplication

The matrix multiplication was conducted using a slightly cache optimized version of the $O(N^3)$ nested loops implementation as shown in Listing B.2. The entire code can be found in `usr/bench/bomp_mm`.

```
static void mm_omp(MATRIX_TYPE *a,
                   MATRIX_TYPE *b,
                   MATRIX_TYPE *c,
                   struct mm_frame *mm_frame)
{
#pragma omp parallel
    {
        uint64_t nrows = mm_frame->rows;
        uint64_t ncols = mm_frame->cols;
        uint64_t counter = 0;
        MATRIX_TYPE sum = 0;

#pragma omp for nowait schedule (static, 1)
        for (int i = 0; i < nrows; i++) {
            counter++;
            uint64_t row = i * nrows;
            MATRIX_TYPE *c_row_i = c + row;
            MATRIX_TYPE *a_row_i = a + row;
            for (int k = 0; k < nrows; ++k) {
                MATRIX_TYPE *b_row_k = b + k * nrows;
                MATRIX_TYPE a_elem = a_row_i[k];
                for (int j = 0; j < ncols; ++j) {
                    c_row_i[j] += a_elem * b_row_k[j];
                }
            }

            for (int j = 0; j < ncols; ++j) {
                sum += c_row_i[j] > 0;
            }
        }

        __sync_fetch_and_add(&mm_frame->sum, sum);
    }
}
```

Listing B.2: OpenMP Version of Matrix Multiply

Appendix C

Barrelfish

This appendix contains additional benchmarks and information about the Barrelfish operating system conducted during the development of the system.

C.1 Memory Mapping in Barrelfish

The sharing of frames among worker domains requires mapping the frame in the respective local address spaces. Depending on the mapping size several frames for the needed page tables have to be allocated and their respective VSPACE and page table structures initialized. The results of the measured time used for the mapping operation can be seen on Figure C.1.

C.1.1 Benchmark Discussion

The experiments have been conducted on the host machine with purely local worker domains. The local measurements were conducted on the master on core 0, the core mem server is running on. The number of threads is the sum of 1 master and n additional workers. Hence the difference between local and the `2T` is only the location where the mapping measurement is conducted (core 1 in this case)

As shown in the graph, the performance characteristics differ by buffer size hence we distinguish the discussion between those buffer sizes.

4kB Buffers Recall a single 4 kB buffer occupies a single slot in a x86_64 page table and hence the additional resources needed for a mapping are low. This results in a quite stable mapping performance compared to a local mapping, or with 2 or 40 worker domains.

32 MB Buffers In contrast to the 4 kB buffers, mapping larger buffers, 32 MB in our case, is getting slower because of the additional page tables that need to be allocated. The resulting overhead is expected to be emphasized with larger buffer sizes. The additional resources have to be allocated which needs

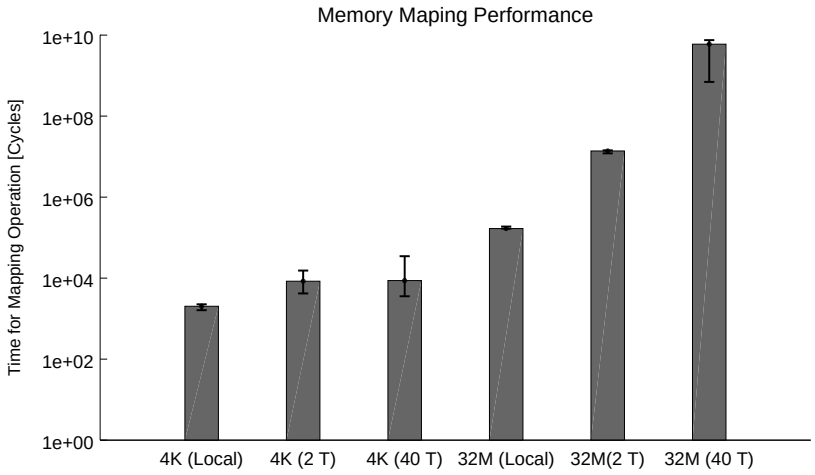


Figure C.1: Memory Mapping Performance with Multiple Domains

additional messages this results in an quite big increase of the running time by two orders of magnitude for 40 threads.

C.1.2 Side Effects

For each mapping a new frame and virtual address range is allocated hence increasing the number of total mapped capabilities and VREGIONS with every new mapping. Even though the calculated standard errors are rather small, the measured time highly depends the size of the VSPACE data structures. As many of the data structures are basic lists the access time to those is $O(N)$. The benchmarks have shown that the time required for mapping the buffer linearly increased with the number of total mappings.

C.1.3 Conclusions

Having a single mem server is clearly no good option when having a work load that issue a lot of requests to the single mem server. The effects are amplified when the buffer sizes increases.

C.2 Boot Performance

The architecture of Barrelfish as described in 3.1 requires explicit updating of the replicated state. This is done purely in user space by the Monitors while the kernel itself does not have state. This requires the Monitors to have communication channels between each other which need to be initialized. Figure C.2 shows the initialization times for each Monitor during the boot process. The elapsed time measured is from the point where `main()` is called to just before the message handling loop. The initialization time per monitor grows with each additional monitor running in the system.

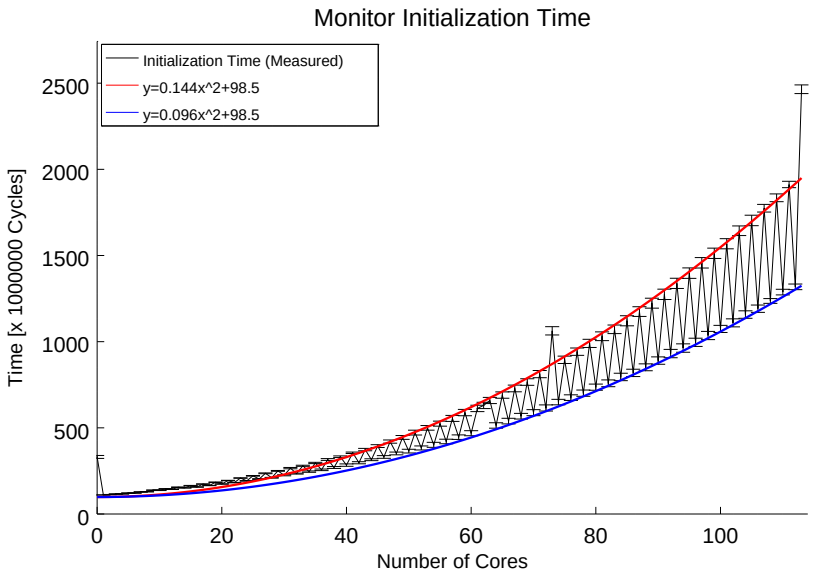


Figure C.2: Initialization Time per Monitor

Quite surprisingly there is a high variance if we see compare the initialization times of two consecutive cores resulting in a zic-zac representation in the graph. We expect that this is caused by an already initialized cache when the second hardware thread is initialized.

Besides the interesting zic-zac pattern one clearly sees the the growth in the initialization time from about 110 Million cycles up to about 2000 Million cycles per monitor. There are few exceptions such as the bootstrap monitor of core 0. The graph itself suggests a quadratic growth of the initialization times (shown in red and blue).

C.3 Querying and Registering Symbols

If a binary is to be spawed with the `SPAWN_FLAGS_OMP` bit set in the supplied flags during the architecture specific loading `spawn_parse_omp_functions` gets invoked which scans through the symbol table of the ELF binary and extracts the information about the OpenMP generated functions. Without the appropriate flags, this data is not extracted and hence not available for later use.

C.3.1 Interface Specification

An extract of the interface can be seen in Listing C.1. In order to use it the domain has to link with `libspawndomain`. The interface is declared in:

```
#include <spawndomain/spawndomain.h>
```

Listing C.1 below shows the functions used in our heterogeneous OpenMP backend to register and obtain the symbol values.

```
/**
 * \brief executes a lookup query on octopus to obtain the symbol
 *
 * \param binary     name of the binary to query
 * \param idx        index of the symbol to query
 * \param retname    returns the name of the symbol
 * \param retaddr    returns the address of the symbol
 */
errval_t spawn_symval_lookup(const char *binary, uint32_t idx,
                             char **retname, genvaddr_t *retaddr);

/**
 * \brief registers a found symbol with octopus for later retrieval
 *
 * \param binary     the name of the binary
 * \param idx        index of the symbol to insert
 * \param symname    name of the symbol to insert
 * \param address    address of the sybol to insert
 */
errval_t spawn_symval_register(const char *binary, uint32_t idx,
                               char *symname, genvaddr_t address);

/**
 * \brief looks up the symbol information based on its address
 *
 * \param addr       the address to lookup
 * \param retidx     returns the symbol index
 * \param retname    returns the symbol name
 */
errval_t spawn_symval_lookup_addr(genvaddr_t addr,
                                  uint32_t *retidx, char **retname);

/**
 * \brief looks up the symbol by a given index
 *
 * \param idx        the index of the symbol to look up
 * \param ret_name   returns the name of the symbol
 * \param ret_addr   returns the address of the symbol
 */
errval_t spawn_symval_lookup_idx(uint32_t idx, char **ret_name,
                                 genvaddr_t *ret_addr);
```

Listing C.1: Interface to obtain Symbol Information

Bibliography

[1] Reto Achermann and Antoine Kaufmann. Bulk Transfer over Network. Distributed systems lab, ETH Zurich, February 2014.

[2] Inc. Advanced Micro Devices. *Heterogeneous System Architecture: A Technical Review.*

[3] Inc. Advanced Micro Devices. *The OpenGL Graphics System: A Specification.* The Khronos Group Inc.

[4] Inc. Advanced Micro Devices. What is Heterogeneous System Architecture (HSA)? Technical report, Advanced Micro Devices, Inc., October 2014.

[5] Antonio Barbalace, Alastair Murray, Rob Lyerly, and Bino Ravindran. Towards Operating System Support for Heterogeneous-ISA Platform. In *4th Workshop on Systems for Future Multicore Architectures*, SFMA '14, 2014.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[7] Andrew Baumann. Inter-dispatcher communication in Barrelfish. Technical Note 011, ETH Zurich, December 2011. `www.barrelfish.org/TN-011-IDC.pdf`.

[8] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *In Proceedings of the 22nd ACM Symposium on OS Principles*, Big Sky, MT, USA, October 2009. ACM.

[9] Andrew Baumann, Chris Hawblitzel, Kornilios Kourtis, Tim Harris, and Timothy Roscoe. Cosh: Clear OS Data Sharing In An Incoherent World. In *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*, Broomfield, CO, October 2014. USENIX Association.

[10] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. Why isn't your OS? In *In Proceedings of*

the 12th Workshop on Hot Topics in Operating Systems, Monte Verità, Switzerland, 2009.

[11] Jeremia Bär and Claudio Föllmi. Bulk Transfer over Shared Memory. Distributed systems lab, ETH Zurich, February 2014.

[12] Melvin E. Conway. A Multiprocessor System Design. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS '63 (Fall), pages 139–146, New York, NY, USA, 1963. ACM.

[13] Intel Corporation. *OpenCL Optimization Guide for HPC Systems*, coding for intel xeon phi coprocessors edition. Saturating the Memory Bandwidth: `https://software.intel.com/en-us/node/515344`.

[14] Intel Corporation. The Single-chip Cloud Computer. Online, 2010-04-26. http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-cloud-article.pdf.

[15] Intel Corporation. Intel Unveils New Product Plans for High-Performance Computing. Online, 2010-05-31. http://www.intel.com/pressroom/.

[16] Intel Corporation. *Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual*. Intel Corporation, 2200 Mission College Blvd, Santa Clara, CA 95054-1549, USA, September 2012. Reference Number: 327364-001.

[17] Intel Corporation. *Intel Manycore Platform Software Stack (Intel MPSS) - User's Guide*, July 2014. Document Number: 330076-001US, Version 3.3, `http://registrationcenter.intel.com/irc_nas/4741/MPSS_Users_Guide.pdf`.

[18] Intel Corporation. *Intel MPI Library for Linux - User's Guide, Document Number: 315398-012*, 2014. `https://software.intel.com/sites/default/files/managed/33/d6/User_Guide.pdf`.

[19] Intel Corporation. *Intel Xeon Phi Coprocessor Datasheet*. Intel Corporation, 2200 Mission College Blvd, Santa Clara, CA 95054-1549, USA, April 2014. Document ID Number: 328209 003EN.

[20] Intel Corporation. *Intel Xeon Phi Coprocessor System Software Developers Guide*. Intel Corporation, 2200 Mission College Blvd, Santa Clara, CA 95054-1549, USA, March 2014.

[21] Intel Corporation. *Symmetric Communications Interface (SCIF) For Intel® Xeon Phi™ Product Family*, February 2014. User's Guide Revision 1.03, `http://registrationcenter.intel.com/irc_nas/4741/SCIF_UserGuide.pdf`.

[22] Intel Corporation. Intel Re-architects the Fundamental Building Block for High-Performance Computing. Online, 2014-06-23. `http://newsroom.intel.com/community/intel_newsroom/blog/2014/06/23/`.

[23] Intel Corporation. Intel Xeon Phi Coprocessor. Online, Accessed: 2014-04-14. `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`.

[24] Intel Corporation. Intel Turbo Boost Technology 2.0 - Quick Overview. Online, Accessed: 2014-10-10. `http://www.intel.com/technology/turboboost/`.

[25] Intel Corporation. ARK | Intel Xeon Phi Coprocessors. Online, Accessed 2014-10-13. `http://ark.intel.com/products/family/71840/`.

[26] Intel Corporation. Intel Many Integrated Core Architecture. Online, Accessed 2014-10-13. http://www.intel.com/mic.

[27] Microsoft Corporation. Microsoft Windows. Online, accessed 2014-10-10. `http://windows.microsoft.com/`.

[28] Microsoft Corporation. Barrelfish - Microsoft Research. Online, Accessed 2014-10-13. `http://research.microsoft.com/en-us/projects/barrelfish/`.

[29] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burgerhor. Power Challenges May End the Multicore Era. *Communications of the ACM*, 55(2):93–102, February 2013. `http://cacm.acm.org/magazines/2013/2/160168`.

[30] Claudio Föllmi. Applying the Multikernel Approach to a Heterogeneous OMPA4460 SoC. Bachelor's thesis, ETH Zurich, September 2013.

[31] K. Geihs, B. Schoener, U. Hollberg, H. Schmutz, and H. Eberle. An architecture for the cooperation of heterogeneous operating systems. In *Computer Networking Symposium, 1988., Proceedings of the*, pages 300–312, 1988.

[32] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM.

[33] R. Grindley, T. Abdelrahman, S. Brown, S. Caranci, D. DeVries, B. Gamsa, A. Grbic, M. Gusat, R. Ho, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, P. McHardy, S. Srbljic, M. Stumm, Z. Vranesic, and Z. Zilic. The NUMAchine Multiprocessor. In *Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, ICPP '00, pages 487–, Washington, DC, USA, 2000. IEEE Computer Society.

[34] Khronos OpenCL Working Group. *The OpenCL Specification - Version: 2.0*. Khronos Group. `https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`.

[35] S. Gupta. Computing with Green Responsibility. In *Proceedings of the International Conference and Workshop on Emerging Trends in Technology*, ICWET '10, pages 234–236, New York, NY, USA, 2010. ACM.

[36] Red Hat. newlib C library. Online, 2014-10-14. `https://sourceware.org/newlib/`.

[37] Jonas Hauenstein, David Gerhard, and Gerd Zellweger. Ethernet Message Passing for Barrelfish. Distributed systems lab, ETH Zurich, July 2011.

[38] Jim Held, Jerry Bautista, and Sean Koehl. From a Few Cores to Many: A Tera-scale Computing Research Overview. Whitepaper, Intel Corporation, 2006.

[39] Free Software Foundation Inc. OpenMP - GCC Wiki. Online, Accessed 2014-08-10. `https://gcc.gnu.org/wiki/openmp`.

[40] Free Software Foundation Inc. *Multiboot Specification*, version 0.6.96 edition, Accessed: 2014-10-15. `http://www.gnu.org/software/grub/manual/multiboot/multiboot.html`.

[41] Texas Instruments Incorporated. OMAP 4 Applications Processors. Online, October 2014. `http://www.ti.com/product/OMAP4430`.

[42] Intel Corporation. *Intel Xeon Processor E5-1600/E5-2600/E5-4600 v2 Product Families Datasheet*, volume one of two edition, March 2014. Reference Number: 329187-003.

[43] Intel Corporation. *Intel Xeon Processor E5 v2 Product Family Datasheet*, volume two: registers edition, March 2014. Reference Number: 329188-003.

[44] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):186–197, June 2007.

[45] Tim Jones. Virtio: An I/O virtualization framework for Linux. online, January 2010. www.ibm.com.

[46] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 201–216, Broomfield, CO, October 2014. USENIX Association.

[47] F. Kon, R.H. Campbell, M.D. Mickunas, K. Nahrstedt, and F.J. Ballesteros. 2K: a distributed operating system for dynamic heterogeneous environments. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*, pages 201–208, 2000.

[48] Kornilios Kourtis. Barrelfish Release Notes 2014-05-22. Online, Accessed: 2014-10-03 May 22, 2014. `https://lists.inf.ethz.ch/pipermail/barrelfish-users/2014-May/001201.html`.

[49] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 64–, Washington, DC, USA, 2004. IEEE Computer Society.

[50] ARM Ldt. ARM big.LITTLE Technology. Online, October 2014. `http://www.arm.com/products/processors/technologies/`.

[51] Tong Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, Jan 2010.

[52] Bellevue Linux. vmlinuz Definition. Online, Accessed: 2014-10-15. Created 2005-03-09, `http://www.linfo.org/vmlinuz.html`.

[53] Dominik Menzi. Support for heterogeneous cores for Barrelfish. Master's thesis, ETH Zurich, Juli 2011.

[54] G.E. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998.

[55] Patrick Moorhead. HSA Foundation - Purpose and Outlook. Technical report, Moor Insights and Strategy, November 2014.

[56] Mark Nevill. An Evaluation of Capabilities for a Multikernel. Master's thesis, ETH Zurich, May 2012.

[57] Edmund B. Nightingale, Chris Hawblitzel, Orion Hodson, Galen Hunt, and Ross Mcilroy. Helios: Heterogeneous multiprocessing with satellite kernels. In *In Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[58] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable SoC. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 7 pp.–, April 2003.

[59] nVidia Corporation. *nVidia Tesla Kepler GPU Computig Accelerators*, October 2013. Datasheet.

[60] nVidia Corporation. online, September 2014. `https://developer.nvidia.com/cuda-tools-ecosystem`.

[61] nVidia Corporation. *CUDA Documentation*, August 2014. `http://docs.nvidia.com/cuda/index.html`.

[62] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, version 4.0 edition, July 2013. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[63] Linux Kernel Organization. The Linux Kernel Archives. Online, accessed 2014-10-10. `https://www.kernel.org/`.

[64] The Linux Kernel Organization. The Linux/x86 Boot Protocol. Online„ accessed 2014-05-10. `https://www.kernel.org/doc/Documentation/x86/boot.txt`.

[65] PCI-SIG. PCI Specifications. Online, Accessed: 2014-10-15. `https://www.pcisig.com/specifications/`.

[66] Fred J. Pollack. New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies (Keynote Address)(Abstract Only). In *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society.

[67] Barrelfish Project. Mackerel User Guide. Technical Note 2, ETH Zurich, March 2013. `www.barrelfish.org/TN-019-DeviceDriver.pdf`.

[68] Barrelfish Project. Barrelfish. Online, 2014. `www.barrelfish.org`.

[69] Barrelfish Project. Online, Accessed: 2014-10-15. `http://hg.barrelfish.org/barrelfish/`.

[70] Timothy Roscoe. Hake. Technical Note 003, ETH Zurich, April 2010. `www.barrelfish.org/TN-003-Hake.pdf`.

[71] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[72] Rusty Russell. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.

[73] Adrian Schüpbach, Andrew Baumann, Timothy Roscoe, and Simon Peter. A declarative language approach to device configuration. In *In Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, USA, 2011.

[74] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *In Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[75] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-core x86 Architecture for Visual Computing. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[76] Akhilesh Singhania, Ihor Kuz, and Mark Nevill. Capability Management in Barrelfish. Technical Note 013, ETH Zurich, December 2013. `www.barrelfish.org/TN-013-CapabilityManagement.pdf`.

[77] Herb Sutter. The Free Lunch Is Over - A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005. `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[78] OASIS Virtual I/O Device (VIRTIO) TC. *Virtual I/O Device (VIRTIO) Version 1.0*, committee specification draft 01 / public review draft 01 edition, December 2013. `http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.html`.

[79] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multithreaded System. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 156–166, New York, NY, USA, 2007. ACM.

[80] Gerd Zellweger. Unifying Synchronization and Events in a Multicore Operating System. Master's thesis, ETH Zurich, March 2012.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

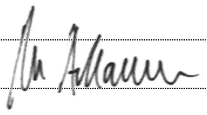
Message Passing and Bulk Transport on Heterogeneous Multiprocessors

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Achermann	Reto

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Zürich, 17. Oktober 2014	

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.