Imperial College London

IMPERIAL COLLEGE LONDON

MENG COMPUTING INDIVIDUAL PROJECT

Customized OS kernel for data-processing on modern hardware

Author:
Daniel GRUMBERG

Supervisor:
Dr. Jana GICEVA

An Individual Project Report submitted in fulfillment of the requirements for the degree of MEng Computing

in the

Department of Computing

June 18, 2018

Abstract

Daniel Grumberg

Customized OS kernel for data-processing on modern hardware

The end of Moore's Law shows that traditional processor design has hit its peak and that it cannot yield new major performance improvements. As an answer, computer architecture is turning towards domain-specific solutions in which the hardware's properties are tailored to specific workloads. An example of this new trend is the Xeon Phi accelerator card which aims to bridge the gap between modern CPUs and GPUs. Commodity operating systems are not well equipped to leverage these advancements. Most systems treat accelerators as they would an I/O device, or as an entirely separate system. Developers need to craft their algorithms to target the new device and to leverage its properties. However, transferring data between computational devices is very costly, so programmers must also carefully specify all the individual memory transfers between the different execution environments to avoid unnecessary costs. This proves to be a complex task and software engineers often need to specialise and complicate their code in order to implement optimal memory transfer patterns.

This project analyses the features of main-memory hash join algorithms, that are used in relational databases for join operations. Specifically, we explore the relationship between the main hash join algorithms and the hardware properties of Xeon Phi cards. We found that non-partitioned hash join implementations suit the Xeon Phi better and that this conclusion should hold for other accelerator technologies as well. This project then tackles the problem of providing system stack abstractions to distribute computation within the available hardware execution contexts. The proposed framework aims to infer efficient data transfer patterns, allowing developers to focus on their algorithms and data flow at a higher level of abstraction than current frameworks, such as Intel's SCIF. Moreover, the framework is evaluated using a set of microbenchmarks designed to emulate common offloading scenarios.

Acknowledgements

Firstly, I would like to take this opportunity to thank my supervisor, Dr. Jana Giceva, for introducing me to this topic, and for the valuable feedback she has provided me with throughout the course of this project. Secondly, I would like to acknowledge the Systems Group at ETHZ, especially Reto Achermann, for generously letting me use their infrastructure and for answering my questions about Barrelfish.

I would like to express my gratitude towards my friends, Franklin, Saurav and Amin, for listening to my incessant rants over the past four years.

I am also grateful for the unconditional love and support always given to me by my parents and younger sister.

Last but not least, I thank my girlfriend, Joanna, for always being willing to proof-read my work, and for always being there at my side.

Contents

Chapter 1

Introduction

1.1 Motivations

With the ubiquity of large scale web applications, internet companies go to extreme lengths in order to improve data centre technology to meet ever-increasing user demands. Corporations stop at nothing to improve performance and reduce data centre operation costs, for example, by designing in-house server units which is nowadays commonplace [1]. Modern hardware innovations can improve data processing application performance, which constitutes a key element in the operation of large scale web applications. Hash join algorithms are particularly important as they implement joins, which are some of the most commonly performed operations in relational databases. However, commodity operating systems and their APIs are becoming increasingly awkward and inefficient in leveraging modern hardware features. This stems from the fact that the architectures underpinning mainstream operating systems are rooted in the 1970s UNIX operating system and its variants. The available hardware has greatly evolved since, and it feels as though a major redesign of the way we think of computer systems is long overdue. In particular, the proliferation of accelerator hardware and the offloading of computation to devices traditionally reserved for I/O purposes is challenging the single monolithic kernel approach, as well as the "everything is a file descriptor" philosophy [2] found in traditional operating system design.

Since the mid-late 2000s [3], there has been renewed interest in creating new operating systems from scratch to take advantage of advances in hardware components and the rise of multi– and many–core heterogeneous architectures. In this project, we aim to explore and evaluate new operating system abstractions to leverage coprocessor technology, in the context of data processing applications. The main challenges and questions this project investigates are:

- What makes a hash join algorithm suitable for execution on Xeon Phi cards or other similar accelerator technologies?
- Can we adapt the existing infrastructure and/or create new abstractions for executing programs or program fragments on accelerators, whilst minimising the impact of data transfers between the disjoint physical address spaces of the host processor and the target accelerator? Can we achieve this goal without developers having to specify the optimal transfer patterns?
- How can we write applications targeting multiple accelerators in a portable manner?

1.2 Challenges

I faced multiple technical challenges during this project, I list the most important ones below:

- **Lack of access to tools**: The compiler and supported frameworks provided by Intel for programming Xeon Phi cards are under a restrictive commercial license. The license I was using belonged to the Systems Group at ETHZ in Switzerland. It expired halfway through my project, and the Systems Group could not justify extending it. This complicated my evaluation and made it impossible to compare my framework to Intel's one.
- **Programming in a research operating system**: I implemented my framework within the Barrelfish operating system. Barrelfish is a research project and as such, is not thoroughly documented. Furthermore, it provides limited debugging facilities. These two factors made it harder to implement my framework. Moreover, Barrelfish's booting infrastructure required restarting the target machine and loading the new OS image over the network. This process would take multiple minutes for every run, thus severely hampering my development speed.

1.3 Contributions

This project's main aim was to provide suitable abstractions for offloading program fragments to accelerators within the context of hash join algorithms. In order to achieve this, I needed to understand the properties of these algorithms and how they relate to the Xeon Phi's architectural features. I then derived a usable interface that would allow programmers to leverage accelerators without worrying about individual memory transfers in the context of these algorithms. To narrow the scope of this project, I chose to restrict my analysis to main-memory hash join algorithms, and to Xeon Phi hardware. The main contributions of this project are listed below:

1. **Hash join analysis**: I provide an analysis of state-of-the-art, main-memory, hash join algorithms, which can be found in Chapter 3. This includes a port of some state-of-the-art algorithms to Xeon Phi cards and a detailed analysis of the architectural features of the Xeon Phi, as well as their impact on the observed performance of these algorithms. I show that non-partitioned hash join algorithms are preferable for execution on Xeon Phi hardware and I justify why this the case for other similar accelerators.
2. **Novel offloading framework**: I propose a new API and associated framework that lets the programmer specify the data flow of their algorithms in terms of a directed acyclic graph of computations. This lets the programmer specify the high level data flow of the application without worrying about individual memory transfer operations. Details can be found in Chapter 4.
3. **Framework evaluation**: I evaluate this framework in Chapter 5, using a set of carefully crafted microbenchmarks, which emulate common offloading scenarios. This shows the various data transfer optimisations the framework performs automatically during the program's execution.

Chapter 2

Background And Related Work

2.1 Current Hardware Landscape

The hardware landscape has changed dramatically in the last few years. Indeed, with growing concerns about Moore's Law's health status and stronger demands on power efficiency, chip designers have been turning away from single-core, high-clock rate processors. Commodity high-performance general purpose microprocessors now feature elaborate techniques such as out-of-order execution, SIMD extensions and hyper-threading. Furthermore, multi-core architectures are now commonplace. The trend is clear — if it is impossible to improve the speed of straight-line sequential execution, application performance can be maximised by extracting as much parallelism as possible from our software.

Beyond this many application domains require performance and power efficiency characteristics not achievable by general purpose microprocessors. These industries are pivoting towards heterogeneous hardware systems, where different components are valued for the different characteristics they offer. This trend is ever-growing and can be readily observed in the smartphone market with power efficient cores [4], as well as in the data centre industry with the advent of FPGAs and smart NICs [5].

2.1.1 Coprocessors and Accelerators

As was mentioned earlier, there is a trend towards building specialised computer hardware to achieve certain tasks with better performance characteristics. We define a coprocessor as a discrete processing unit capable of some form of general purpose computation. The most common type of coprocessor is undeniably the discrete graphics processing unit, although other kinds exist.

General Purpose Graphics Processing Unit

The invention of GPU hardware and its usage in general purpose applications as a means of achieving better performance stems from two observations [6]:

- ILP is difficult to extract from a sequential instruction stream. In a classical microprocessor this is aided by out-of-order execution mechanisms. However, it requires maintaining expensive control state, which ensures that program execution seems sequential. This can not be exposed to programmers, thus they can not fully saturate existing ALU components. In order to create explicit ILP abstractions, modern processors expose SIMD extensions to their ISAs. However, they introduce a lot of extra hardware — wider registers, SIMD processing units in the ALU, as well as the added control logic required for decoding the new instructions. Furthermore, the SIMD execution model does not map very well to control heavy applications e.g. a parser, in which case the extra register width is wasted.

- Control hardware dominates microprocessors and it does not directly contribute to the instruction execution rate. Indeed, Pollack's Rule [7] states that microprocessor performance increase is roughly proportional to the square root of the increase in complexity, that is the number of transistors. Modern designs go to extreme lengths to speed up a sequential instruction stream, which leads to a large increase in the number of transistors. The proportion of control hardware to computational units is highlighted in Figure 2.1.

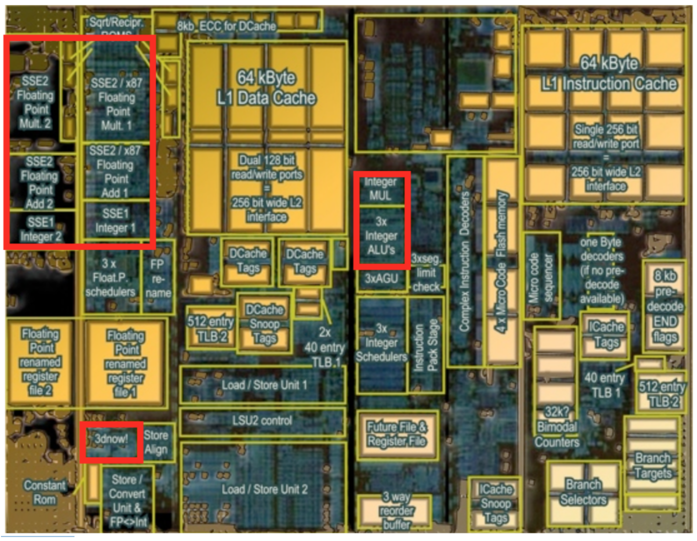

Sqrt/Recipr
8kb ECC for DCache
SSE2 Floating Point Mult. 2
SSE2 / x87 Floating Point Mult. 1
64 kByte L1 Data Cache
64 kByte L1 Instruction Cache
Single 256 bit read/write port
256 bit wide L2 interface
SSE2 Floating Point Add 2
SSE2 / x87 Floating Point Add 1
Dual 128 bit read/write ports
256 bit wide L2 interface
Integer MUL
3x Integer ALU's
SSE1 Integer 2
SSE1 Integer 1
3 x Float.P. schedulers
FP re-name
DCache Tags
DCache Tags
3xAGU
3xseg. limit check
Complex Instruction Decoders
4 x Micro Code Flash memory
Micro code sequencer
one Byte decoders (if no pre-decode available)
8 kb pre-decode END flags
512 entry TLB 2
DCache Snoop Tags
2x 40 entry TLB 1
3x Integer Schedulers
Instruction Pack Stage
ICache Tags
Floating Point renamed register file 2
Floating Point renamed register file 1
Load / Store Unit 1
40 entry TLB 1
512 entry TLB 2
LSU2 control
32k? Bimodal Counters
3dnow!
Store Align
Future File & Register File
Branch Targets
Constant Rom
Store / Convert Unit & FP<>Int
Load / Store Unit 2
3 way reorder buffer
ICache Snoop Tags
Branch Selectors

FIGURE 2.1: AMD "Deerhound" K8L die, red boxes show components performing arithmetic. John Owens, *GPU Architecture Overview*, `http://gpgpu.org/static/s2007/slides/02-gpu-architecture-overview-s07.pdf`, [Online; accessed January 23rd 2018], 2007.

These two observations have lead to a simple design principle. To maximize instruction execution rate, it is necessary to maximise the amount of SIMD *lanes* whilst sharing as many of the control components as possible. Modern GPUs take this principle as far as possible, as is shown in Figure 2.2. This particular modern GPU provides 5376 32-bit floating point cores, 5376 32-bit integer cores, 2688 64-bit floating point cores and 672 Tensor Cores (optimised matrix multiply and add), which allow it to achieve 15.7 TFLOP/s. It achieves this impressive performance by effectively creating 1024-bit wide SIMD processing lanes that share only a minimal amount of local memory, known as warps in NVIDIA terminology. The warps are aggregated into 84 streaming multi-processors which contain the first level caches and fetch-and-decode logic. However, they are very different from traditional microprocessors, as they do not perform branch prediction and run at much lower clock rates, which allows them to exhibit great power-efficiency.

The trade-off GPUs make is to provide very high throughput in exchange for single thread performance. This maps particularly well to "embarrassingly" parallel

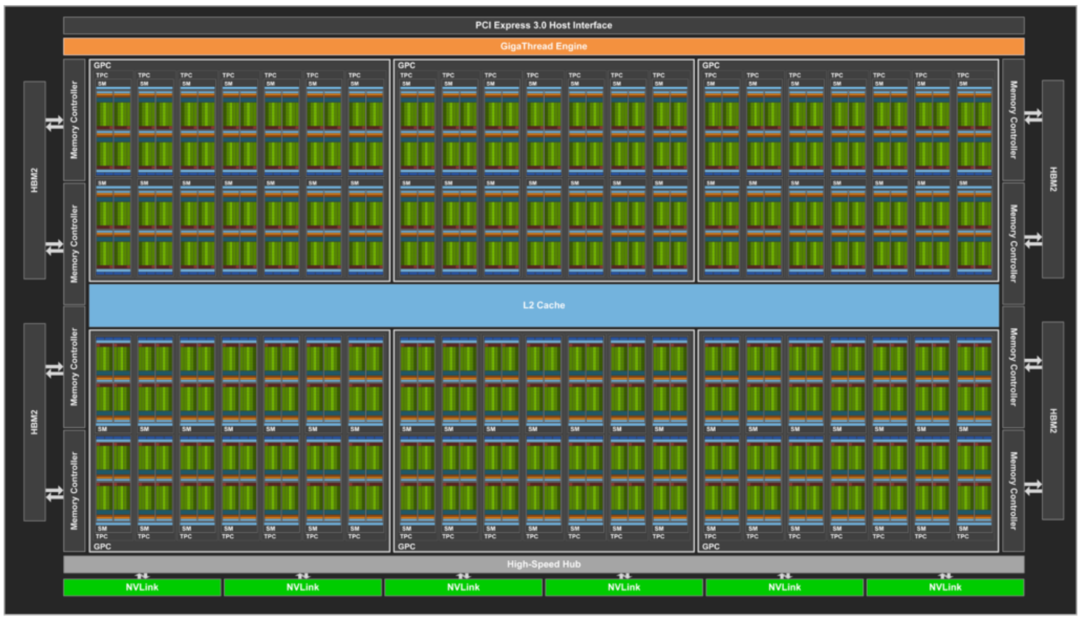


FIGURE 2.2: Architecture diagram for NVIDIA Volta architecture, small green boxes represent arithmetic units. Luke Durant and Olivier Giroux and Mark Harris and Nick Stam, *Inside Volta: The World's Most Advanced Data Center GPU*, `https://devblogs.nvidia.com/inside-volta/`, [Online; accessed January 23rd 2018], 2017.

workloads found in graphics and certain graph-processing applications. Indeed, being able to process thousands of identical work items in parallel enables the full utilisation of the throughput available on a GPU. On the other hand, programs that do not exhibit these kinds of levels of parallelism will suffer from higher latencies and from a severe lack of resource utilisation which renders them power-inefficient.

In conclusion, getting rid of the requirement to perform well for all applications enables specialised architectures to provide exceptional throughput. GPUs are thus suited to be utilised as coprocessors and have computation *offloaded* over the PCIe bus onto them when appropriate. However, the data transfer between main memory and the GPU memory is extremely expensive and thus needs to be carefully managed to prevent losing all the performance benefits the GPU affords in the first place. The problem is compounded by the overhead of copying data from userspace across to the kernel in order for the GPU's device driver to send it safely across the system bus. Furthermore, the parallelism of the platform is directly exposed to the programmer, through new languages and APIs to communicate with the card. Thus, porting applications requires significant programming effort and re-engineering to adapt existing software to the platform.

Intel Many Integrated Core Architecture

Often, it is not possible to extract the necessary ILP out of applications to make them suitable for GPU execution. Nevertheless, many applications exhibit high task parallelism. This means that many mostly-independent tasks can be run in parallel. However, they would benefit from low latency in their sequential segments.

Also, GPUs do not implement efficient caching and thus shared-memory applications tend to be unsuitable for GPUs. Intel is trying to provide an offering for parallel coprocessors that allows for efficient execution for a wider variety of workloads, whilst maintaining compatibility with familiar programming models. The first instalment of Intel MIC was the Intel Xeon Phi X100 nicknamed Knights Corner [9] released at the end of 2012. I will be basing this section upon its architecture because it is the hardware platform I have access to. I will be referring to it as Xeon Phi.

The Xeon Phi is targeted at high performance computing facilities and data centres with rigid power consumption constraints. It aims to provide programmers with a familiar development environment. To this effect, it runs a full service Linux operating system (albeit patched), supports x86 memory order model and IEEE 754 floating-point arithmetic, and offers compiler support for Fortran, C, C++, and various Intel extensions to these languages. The Xeon Phi coprocessor connects to a *host* Xeon processor over the PCIe bus. The provided tools and libraries implement a TCP/IP stack over this link to allow communication through standard network protocols such as `ssh`. Beyond this, multiple Xeon Phis can be connected to the same system through standard PCIe expansion slots and communicate through a peer-to-peer protocol without intervention from the host [9]. Furthermore, programming is simplified by allowing the host microprocessor to access the memory of the Xeon Phi through a memory mapped region referred to as *aperture*. This enables the use of traditional virtual addressing. Communication in the reverse direction can be achieved with the same technique. This allows for a very simple and familiar way of communicating between the two address spaces, which is especially convenient for shorter messages. For larger memory transfers, asynchronous hardware-managed transfers using DMA are still preferable [10].

The microarchitecture features 60 cores, memory controllers and caches (notably a distributed L2 cache), PCIe client hardware, and a very high bandwidth bidirectional ring interconnect. The cores themselves implement a simple short **in-order** pipeline, supporting up to 4 hardware threads, running at a base clock rate of 1.24GHz [11]. All these architectural features help improve power-efficiency. Moreover, they support almost all legacy features of x86 ISAs. The exciting feature of the cores is their vector processing units, which provide 512-bit wide SIMD extensions that include fused-multiply-and-add, gather-scatter instructions, as well as transcendental operations, such as logarithm and square root [12]. Thus, despite being in-order cores, they are able to support ILP through SIMD and SMT, not unlike a GPU.

Each direction of the bidirectional interconnect features three separate rings:

- The data block ring is 64-bit wide and supports reads and writes to the L2 cache and wider memory system. Its width is necessary in order to be able to support all the cores efficiently.
- The command and address ring is much smaller. It is used for transmitting read/write commands and memory addresses.
- The acknowledgement ring is the smallest and is used for flow control and coherence messages.

When a core tries to access its L2 cache and misses, it broadcasts an address request to the tag directories. If the line exists on another core's L2 a forwarding request is sent to it on the address ring, and the requested block is then forwarded on the data block ring. If the line is not found, the tag directory sends a request to the appropriate memory controller. The memory addresses are uniformly distributed amongst the tag directories. Furthermore, the tag directories track cache lines in all L2 caches. All tag directories keep track of addresses from all memory controllers in order to prevent hotspots in the interconnect [9]. A diagram of the interconnect layout is shown in Figure 2.3.

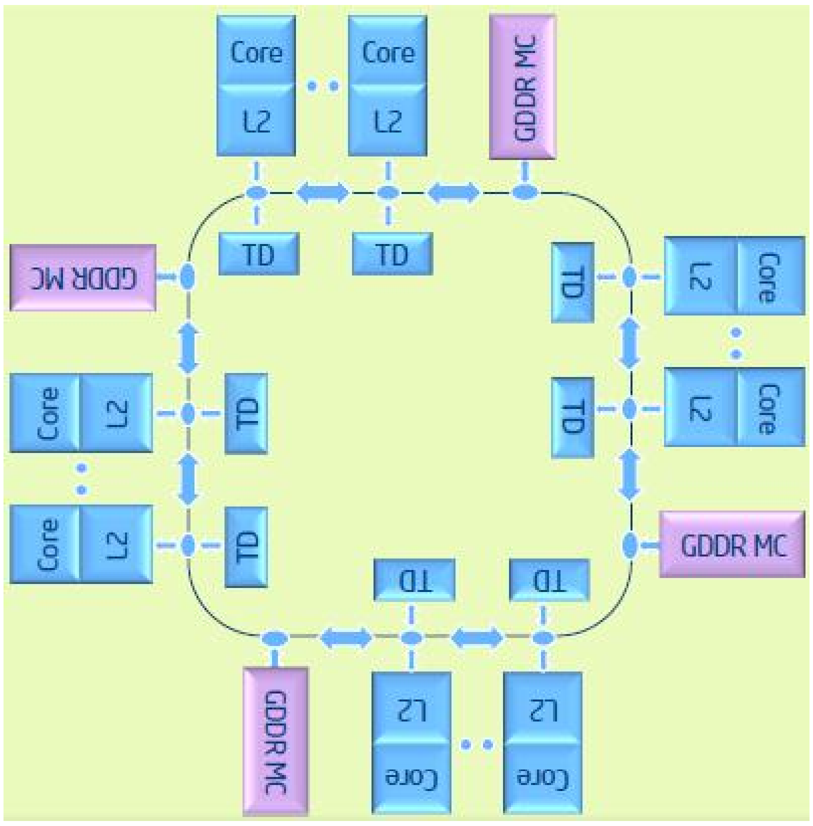


FIGURE 2.3: Interconnect layout in the Xeon Phi George Chrysos, *Intel® Xeon Phi™ X100 Family Coprocessor - the Architecture*, `https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner`, [Online accessed January 23rd 2018], 2012.

Through relying more heavily on caches, the Xeon Phi can handle successfully a wider variety of workloads than a GPU. However, the ring architecture of the card prevents fine-grained control of the locality of the L2 cache which seriously limits the ability of the Xeon Phi to handle a wider variety of workloads. Again, there still are applications where the cost of transferring data across the PCIe bus prohibits offloading the computation to it. It is much better to run these workloads entirely on the host, from a performance standpoint. In these situations, the Xeon Phi would still need to be left powered-on. To mitigate this undesirable excess power consumption, Xeon Phi implements sophisticated power management features. There are four power-saving states the coprocessor can find itself in:

1. When on a single core's all four threads are halted, the core's clock automatically power-gates itself.

2. When a core's clock has been power gated for a configurable amount of time, the core power-gates itself.
3. When all the cores are power-gated and there is no more activity detected by the system agent, the tag directories, the L2 caches, and the interconnect become power-gated.
4. Now, the host driver can put the entire coprocessor into an idle state. The system agent is power-gated, the PCIe hardware is put in a wait state, and the memory units are put in a self-refresh mode that consumes little power.

Intel claims that these features make the Xeon Phi a viable accelerator solution for large data centres with strict power consumption requirements [9].

FPGA and ASIC in Smart Network Interface Controllers

Microprocessors with offloads to coprocessors such as GPUs and Intel MICs are attractive solutions to improve application logic performance. Therefore, we want to prioritize their usage for application logic, and not use them up for simple menial tasks. For example offloading the entire TCP protocol to a smart NIC is an increasingly common occurrence. It allows faster inter-cluster networking inside data centres needing to serve huge amounts of data to a large number of clients, whilst still relying on standard Ethernet switches [13].

In order to implement a partial or full network protocol, the NIC needs to include additional circuitry beyond the usual MAC layer implementation found in traditional network cards. There are essentially two ways of achieving this:

- Including some form of support for a programmable microprocessor inside the NIC [14]. The advantage of this approach is that it allows a lot of flexibility. This approach is slowly starting to become popular as SDNs are becoming increasingly adopted in data-centres. These are often integrated into the data centre's switches, which allows them to implement custom data-transfer protocols over Ethernet in a WAN.
- A more common system is to offloads parts of existing protocols, most notably TCP [15], into fixed function hardware components implemented as ASICs or FPGAs. The latter is very popular in data centre environments because of its ability to be field-reprogrammable and to support custom protocols if required [16].

2.1.2 Rack-scale Computing

Rack-scale computing refers to a recent shift in data centre management and deployments, as well as a research area focused on building large scale machines. The idea is to fill the gap between single server deployments and server clusters. There seems to be a missed opportunity in filling such machines with plain CPUs connected through standard Ethernet networks. The current research trend in this area is to view these as resource-rich deployment units, that provide heterogeneous compute abilities through the use of standard CPUs, as well as accelerators and power-efficient cores. They also aim to provide large and varied amounts of storage, in the form of RAM, Flash based memory, SSDs, and more standard hard-drives. However, internal communication can not happen over Ethernet for latency reasons, so many different types of interconnect with efficient SDNs are being investigated. In a way, they aim to neatly package all the available hardware innovations described in previous subsections [17].

This allows large data centre resource management systems to still deal in homogeneous units, which greatly simplifies their jobs, whilst retaining the ability to leverage the attractive features of heterogeneous hardware platforms. However, this raises significant questions:

1. How are applications implemented for a rack-scale computer? Is it possible to treat them as a uniform shared memory architecture would be? As a system with many NUMA nodes accessed through RDMA? As a distributed system?
2. Is the efficient management of these varied resources a possibility?
3. What is the failure model? Is fault-tolerance achievable?

These are still open questions in this field.

2.2 State of the Art Operating System Design

In a computer system, the operating system is responsible for providing the programmer with abstractions of the underlying hardware components and for exposing them through varied APIs. Currently, the most common operating systems are Android/Linux (on smartphones) and Windows (on laptops and desktops). Both systems are quite old and find their roots in the era of single core x86 processors. This legacy prevents them from elegantly leveraging the recent hardware innovations presented in the previous section to their full potential. Furthermore, the way computer systems are thought about has outgrown the shared uniform memory model. New abstractions and programming models are necessary to encode this paradigm shift in order to enable a simple, portable development environment suitable for high-performance applications.

2.2.1 Scalable Operating System Architecture for Heterogeneous Multi-Core Systems

Handling multi-core systems efficiently and elegantly seems to require a major redesign from the monolithic kernel architectures of old, for scalability reasons. The issue stems from the fact that modern hardware systems involve multiple discrete devices capable of arbitrary computation within their own memory address space. Currently, the only way to communicate with such devices is through device drivers. This is inefficient as the device driver interface is designed for writing and reading data from a bus-connected device, as opposed to scheduling arbitrary computation on it [18], which introduces significant complexity into device driver code. For example, most CUDA drivers implement a JIT compiler. A new promising approach to this problem is multikernels. A multikernel system provides a separate kernel for every core or computationally capable device [3] [18] [19]. An example of such an architecture is shown in Figure 2.4.

This design is motivated by multiple factors, listed below:

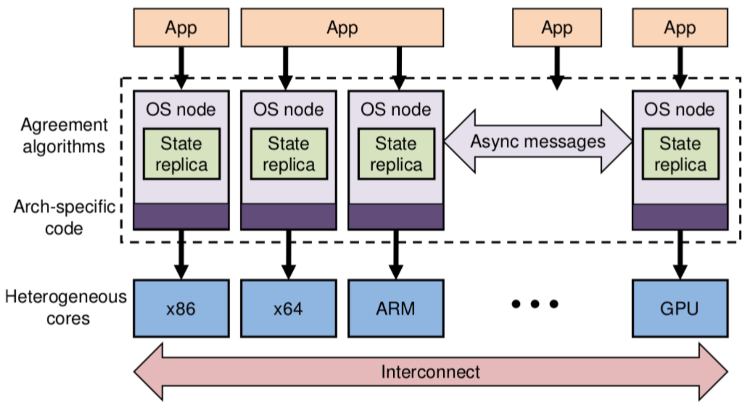


FIGURE 2.4: Barrelfish a multikernel architecture Baumann, Andrew and Barham, Paul and Dagand, Pierre-Evariste and Harris, Tim and Isaacs, Rebecca and Peter, Simon and Roscoe, Timothy and Schüpbach, Adrian and Singhania, Akhilesh, "The Multikernel: A New OS Architecture for Scalable Multicore Systems", in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09, Big Sky, Montana, USA: ACM, 2009, pp. 29–44, ISBN: 978-1-60558-752-3. DOI: 10 . 1145 / 1629575 . 1629579. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629579.

- Different cores cannot share a single kernel as they each could implement a different ISA.
- Interconnects between different cores and microarchitectural elements are implemented as message passing networks, as opposed to single shared buses [20] [9]. Many protocols are implemented as such in large scale multiprocessor architectures, most notably cache coherence protocols. In these devices networking effects related to routing and congestion are recurring concerns.
- Cache coherence and single virtual address space are models that are becoming impossible to efficiently implement. This is largely due to the many computationally-capable devices having discrete physical address spaces. Maintaining a coherent view of the resulting composite address space in a virtual memory environment would introduce difficulties related to pointer size, large variance in memory access delays, and the need for a software-managed cache coherency protocol. For example, a write to a cache line from a GPU could potentially trigger a cache invalidation in a NIC. Implementing such a mechanism efficiently seems nothing short of impossible.
- Efficient shared memory programming is inherently tied to the cache architecture and associated coherency protocols. In a sense, high-performance low-level applications are becoming increasingly difficult to port as they often rely on many architectural features of their chosen platform to achieve their required performance.

The Barrelfish operating system is the most mature such system which is open-source. This is why I decided to implement my project within it. It takes a very principled stance in viewing the OS as a distributed system which promises the best scalability. Its high-level design design decision are listed below [3]:

- **Explicit inter-core communication**: Cache coherence is not an achievable goal for future architecture, thus inter-core communications can not rely on it. This leaves explicit message passing as the only solution. A side-effect of message passing is the safety afforded by well-defined interfaces and protocols.
- **Hardware independent OS structure**: Implementing the system's behaviour through abstract distributed protocols enables the reduction of hardware dependent code. Only the implementation of the message passing interface for the available transport channels and the functionality that is tied to the actual operation of the hardware need to be specialised.
- **No shared state**: The above two points prevent the system from sharing state between any two cores. Thus, the state has to be replicated across the kernels and be modified through explicit message passing. These kinds of optimisations are already present in the Linux kernel in the form of per-core scheduler queues [21]. These kinds of design allow better scalability as explicit synchronisation is not required, thus affording the use of weaker consistency models.

None of the available systems implement extensive support for heterogeneous hardware. A proposed approach is to compile applications to an intermediate language and annotate their *affinities* to various supported devices. The loading and launching process then needs to compile the code down to an appropriate ISA for the chosen device and then load it into that device. An alternative scheme involves producing *fat binaries* that contain a variant of the application for each device as well as the affinity metadata. Nevertheless, the need to decide supported platforms ahead of time and the sheer size of the binaries seems prohibitive [18]. Another scheme is to infer the best platform for a code fragment at compile-time and to generate the code to perform the necessary offloads at runtime [22]. However, this approach requires extensive compiler support and places a lot of pressure on the associated cost model.

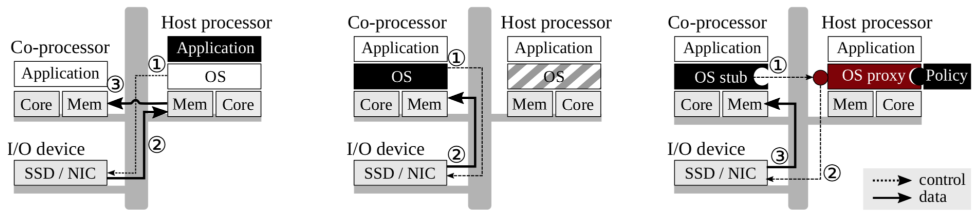
2.2.2 Unikernels and Library Operating Systems

The unikernel and library OS design philosophy is to only provide the minimal set of required abstractions for running an application. Typically these kinds of system allow the developer to pick and mix operating system components from a set of libraries that are then statically linked into the main application resulting in a single address space image [23]. They are typically not designed for multiplexing hardware across multiple processes, instead allowing single applications to achieve performance comparable to that of executing on raw hardware.

Recently, alternatives to IaaS clouds, such as FaaS, are being considered by cloud service providers. The FaaS model allows developers to express their application logic as event handlers in a high level language, without having to worry about configuring the underlying system. The cloud provider just needs to provide PaaS services and worry about instantiating and scheduling the different functions. Naïvely, one could spin up a new virtual machine for each new function invocation, which would lead to excessive start-up cost. At the other end of the spectrum, an implementation could use a single virtual machine hosting a runtime such as NodeJS and submitting function calls to it. Unfortunately, this presents obvious security concerns. The secure alternative is to maintain virtual machines dedicated to functions belonging to a single application. The challenge here is that maintaining too many virtual machines incurs excess cost, whilst having too few results in quality of service issues. The main challenge involves scaling the number of instances up and down very quickly. The unikernel design is particularly suited to this kind of task, as its light-weight nature promotes quick boots and shutdowns, whilst maintaining a low memory and resource usage footprint [24].

Similarly, this project aims to offload applications onto accelerators by separating the implementation effort into a *compute plane* embedded within the Barrelfish operating system, and a separate, much leaner, *data plane*. This scheme would allow the accelerator to remain almost fully idle whilst it is not being used. This is very attractive for accelerators such as the Xeon Phi, which implement advanced power management features highlighted in Section 2.1.1.

An example of this approach for Xeon Phi programming support is presented in the Solros [25] system. This system bases itself on the existing Linux kernel port to Xeon Phi, and reduces its size by delegating costly I/O operations, namely file-system manipulation and network communications, to the host kernel via the means of RPCs into the host OS. The host then coordinates data-delivery directly to the Xeon Phi from the I/O device, without going back to the host. This makes sense as the in-order nature, simplistic branch predictor, and slow clock rate of the Xeon Phi make it a poor candidate for running control flow heavy I/O stack code through *virtio* or *NFS*. The other option is having a host application mediate all I/O operations of the offloaded work. In this scenario, the application's performance would be unnecessarily hampered by excessive data buffering into the host applications address space. Solros achieves performance one order of magnitude better than the Intel-provided Linux kernel for file-system and network stack implementations. More specifically, the improvements were of 19x and 7x respectively both on micro-benchmarks and on simple applications whose data processing elements are well suited to the Xeon Phi [25].



(a) Host-centric architecture (b) Co-processor-centric architecture (c) Solros architecture

FIGURE 2.5: Solros architecture compared with alternatives. Min, Changwoo and Kang, Woonhak and Kumar, Mohan and Kashyap, Sanidhya and Maass, Steffen and Jo, Heeseung and Kim, Taesoo, "Solros: A Data-centric Operating System Architecture for Heterogeneous Computing", in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18, Porto, Portugal: ACM, 2018, 36:1–36:15, ISBN: 978-1-4503-5584-1. DOI: 10.1145/3190508.3190523. [Online]. Available: `http://doi.acm.org/10.1145/3190508.3190523`.

2.2.3 Memory Management in Non-Uniform Architectures

In traditional operating systems, memory management is mostly a solved problem, through virtual memory and paging. In this scheme, each process maintains its own address space where memory locations are represented as machine-word-sized integer addresses. Furthermore, each core has an address translation unit responsible for turning virtual addresses into physical memory locations. For the operating system, managing MMUs requires keeping track of the locations of page table roots, device registers e.t.c.. This technique is old, well-understood, elegant and quite performant, so why is it not suitable for the systems of the future? Virtual memory relies on three simplifying assumptions:

- All RAM and memory mapped I/O appear in a single physical address space.
- Each core's MMU is able to access the entire address space.
- There is no ambiguity — each processor interprets physical addresses in the same way.

However, in the new model of *islands* of computations, each individual computational device often includes on-board memory that is not directly accessible to the host system. The assumptions made above clearly do not apply any more, and instead such systems contain many *intersecting* physical address spaces. Common operating systems solve this problem by using virtual memory only for the host processor, whilst delegating the burden of managing memory in accelerators to the user and the device driver. Nevertheless, in order to increase the operating system's role in scheduling computation onto these accelerators it will be necessary to address this issue cleanly. I present the solution proposed by Simon Gerber and Gerd Zellweger and Reto Achermann and Kornilios Kourtis and Timothy Roscoe and Dejan Milojicic [26] at USENIX HOTOS'15.

This paper defines the concept of a physical address space, AS for short. Each AS has a unique identifier, a range of address values and a function from addresses to referents. There are two types of referents in this scheme:

- Actual memory locations (RAM or memory-mapped hardware), which can only be addressed in this AS by construction.
- An address referring to another AS.

ASes are split into contiguous *regions*, each of which can contain one of the above referent types. This is done to prevent interleaving of different referent types in an AS. Allowing arbitrary interleavings would require tracking the referent type per address, which would significantly increase memory usage. Regions of addresses referring to another AS require a function, which can be static or dynamic, mapping from addresses in the current AS to the next one. Note that this scheme does not prevent addressing loops, which is a limitation also present in the traditional virtual memory technique, as routing loops can be set up through memory maps regardless. A computing system can thus be seen as an arrangement of ASes and processors, as is shown in Figure 2.6.

However, this model does not address the high variability in memory access times between different ASes, as would be the case in an heterogeneous system. Replicating data on both sides is unfortunately still a necessary reality in programming accelerators. Data replication and maintaining coherency can be an expensive operation. Therefore, developing a cost model for data copying costs relative to the performance improvements afforded by the specialised hardware is paramount. This is a challenge that has yet to be solved.

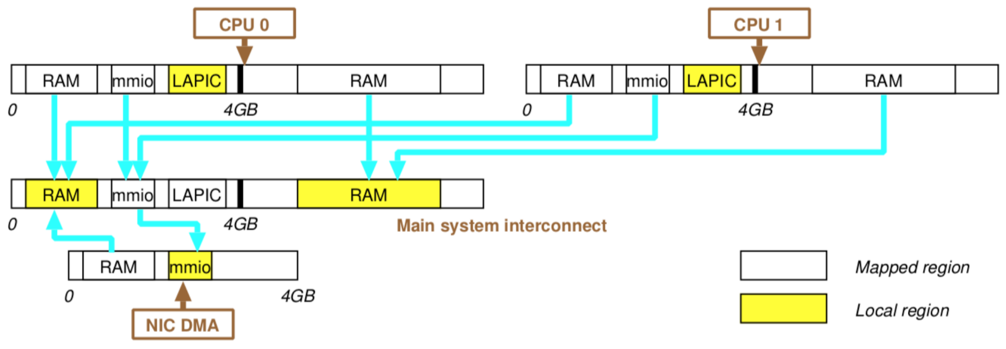


Figure 2.6: Barrelfish memory management example representing a dual-core 64-bit PC with a 32-bit DMA capable NIC. Simon Gerber and Gerd Zellweger and Reto Achermann and Kornilios Kourtis and Timothy Roscoe and Dejan Milojicic, "Not Your Parents' Physical Address Space", in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/hotos15/workshop-program/presentation/gerber.

2.2.4 Scheduling in an Heterogeneous Architecture

Scheduling computation in a heterogeneous system is a vastly different problem to classic scheduling in homogeneous systems. Traditionally, scheduling involves multiplexing oversubscribed hardware resources to a multitude of competing processes according to policies balancing fairness, speed of completion and individual application requirements. The problem now consists of scheduling varied hardware resources to a large amount of *cooperating* tasks, based on the performance of the entire application. In this context, the particular strengths and weaknesses of each accelerator, as well as the costs of data transfers between the different address spaces need to be taken into account when assigning tasks to computational devices. Given these constraints it is clear that preemptive multitasking is not an option as the tasks are cooperating, and a task-based approach needs to be employed. Unfortunately, this has been shown to be an **NP-complete** problem [27], as it is comparable to the knapsack-problem. This leaves only heuristic-based approaches to prevent expensive operations in the scheduling hot path. The approaches to this problem can be roughly classified as static or dynamic.

Gang Scheduling

Gang scheduling is possibly the first form of scheduling algorithm for parallel systems. Here, a gang refers to a group of tasks that need to communicate frequently. An application is composed of multiple gangs and is referred to as a bag of gangs (BoG). It is a desirable property to schedule all jobs in a gang together as it allows them to communicate without suffering the cost of a context switch. At specified time quanta, a multi-context switch happens, where all tasks are swapped out at once and allow another gang to be scheduled [28]. If a gang cannot be filled then it is essential to consider how to add unrelated tasks to the current time-slice or to co-schedule another gang simultaneously. It is not known ahead of time how long a job will take, therefore there is no optimal algorithm for gang formation and scheduling that can guarantee maximum processor utilisation in a preemptive system. Of course, in cooperative multi-tasked system the multi-context switch does not need to happen. Gang scheduling relies on information about the interactions between tasks, which is not available without manual annotation or extensive program analysis.

Work-stealing

Dynamic environments are characterised by tasks being created spontaneously without prior knowledge of their number, their dependencies, their durations or communication patterns. Such environments can be described as fork-join models — directed acyclic graphs of tasks with a start node and an end node representing the start and end of the application. In these environments the algorithm of choice is work-stealing. The model employed by work-stealing consists of assigning each processor with a queue of tasks. A processor can push work to its queue and pop work off it. When the execution of a task creates a new task, the processor places it at the back of its queue, then any idle processor can attempt to steal work of the back off the queues of other processors [29]. There are two possible implementations of work stealing:

1. **Child-stealing**: Here, the current processor keeps executing the current task and places the new task at the back of its queue. It then becomes available to other processors. This algorithm allows an unbounded explosion of the number of available tasks.
2. **Continuation-stealing**: Here, the current processor immediately switches context and executes the new task. Other processors, may then steal the current task. This prevents an unbounded explosion of tasks at the expense of additional context-switches. However, compilers can help eliminate some of these additional context switches.

Work-stealing augmented with programmable policies for specifying affinities is the mechanism employed by the StarPU platform for task offloading [30]. In StarPU, each accelerator device type is assigned a run-queue that is filled according to the task's affinity with this device. Each queue is then ordered according to the current scheduling policy. Devices can now pop tasks off their respective queues until the end of the application.

DAG Scheduling

Dynamic approaches are very convenient in that they free the programmer from scheduling considerations. However, they incur overhead as they make handling the cost of data transfers difficult, as all processors compete for all available work. As seen previously, most parallelism in applications can be viewed as a DAG. An alternative approach to scheduling parallel systems is to have the programmer specify a DAG representing the data flow of the computation performed by the application. Specifically, I explore the scheme proposed by Rossbach, Christopher J. and Currey, Jon and Silberstein, Mark and Ray, Baishakhi and Witchel, Emmett [31] for managing GPUs as computational devices at the operating system level.

The proposed **PTask API** essentially allows applications to construct a data-flow graph of their computation where vertices are code fragments, known as PTasks, to be executed on a compute resource, and where edges represent the flow of inputs and outputs of the PTasks.

PTasks can be in one of four states:

- Waiting for inputs.
- Queued (inputs available and waiting to be executed).
- Executing (currently running).
- Completed (waiting for its output to be consumed).

The implementation defines four scheduling disciplines:

- First available and FIFO: The host CPU threads of the PTasks compete for the accelerators, and access is arbitrated through locks on the data structures. The FIFO mode enhances this with queueing.
- Priority and Data-aware: PTasks are assigned a static priority. PTasks are scheduled from the run queue based on their priority, average wait time, current wait time, and host priority. The scheduler pops PTasks off the head of the queue and selects appropriate accelerators based on its affinity with them. The data-aware mode enhances this scheme by taking into account which accelerator's address space does the inputs of the PTask reside in. High-priority tasks need to be executed straight away, so the scheduler will just incur the cost of the data copy in order to finish, if the best fit available accelerator does not have the necessary data in its address space. However, if the task has a low priority the scheduler will keep it in its queue until the accelerator becomes available.

However, this approach limits the programmer in computations that can be expressed as a DAG statically, which prevents the use of recursion, loops and other such constructs.

The framework and associated API I propose in Chapter 4 is inspired by the PTask API. However, my system assumes that tasks cooperate. Furthermore, the scheduler's main goal is to minimise the cost of data transfer operations.

Chapter 3

Main Memory Hash Joins on the Xeon Phi

3.1 Hash Join Algorithms Explained

Joins are essential database operators, which allow one to find all entries in two tables that share a key. Over the years, much effort has gone into optimising them, as even small performance gains have a large impact, given the amount of join operations executed on most databases. There are two main lines of thought when it comes to designing these algorithms. The first approach, labelled *hardware-conscious*, postulates that such algorithms should be explicitly tailored to the architecture they are run on. Notably, the cache sizes, TLB size and miss latency, as well as memory bandwidth need to be taken into account [32]. The advocates of the *hardware-oblivious* approach claim that modern multicore CPUs and the many optimisations they implement successfully manage to hide cache and TLB miss latencies. Furthermore, it is commonly believed that such hardware-oblivious algorithms are more resistant to the data skew present in real-world workloads [32].

3.1.1 Hardware Oblivious Approaches

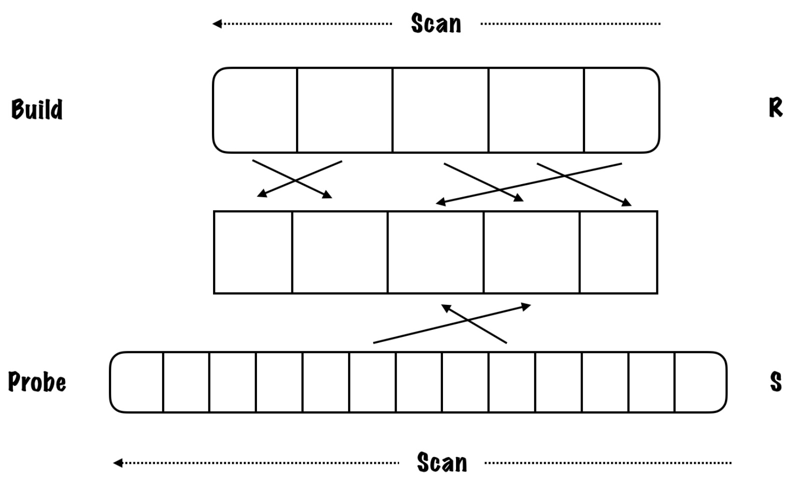


Figure 3.1: Canonical Hash Join Implementation

The hardware-conscious approach builds upon the hardware-oblivious algorithms presented in this section. The canonical algorithm takes two relations, R and S, as inputs, where $|R| \leq |S|$. The output of the algorithm can be either the number of matching entries, a join index, which is the key and record identifier for both relations, or the matching entries themselves. In its serial form, the algorithm is composed of the two steps detailed below and shown in Figure 3.1:

1. **Build**: Initially, a hashtable H of the entries in the smaller relation R is built using a well-chosen hash function. The chosen hash function is both fast to execute and achieves good key distribution across the buckets.

2. **Probe**: The relation S is traversed and, for every entry, the key is hashed to get the appropriate bucket of H. The key is then compared with every entry in the resulting bucket to find all the matches.

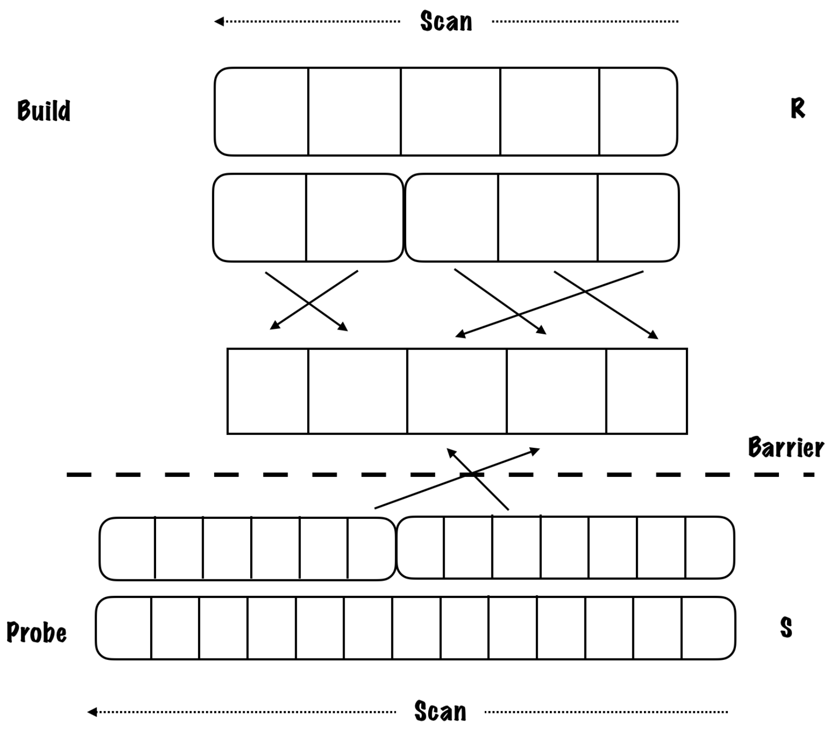


FIGURE 3.2: Simple Hash Join Implementation

The algorithm is parallelised using a straightforward scheme. In the build phase, R is divided into evenly-sized chunks between worker threads. Each worker processes its chunk by adding entries to a global hashtable. The worker threads then synchronise using a barrier before entering the probe phase. Again the relation S is split into equal chunks, that are processed in parallel by the workers following the serial algorithm detailed above. The control flow can be seen in Figure 3.2. It is important to note that the hashtable is a shared data structure between all the workers and as such, accesses to it need to be properly synchronised. In typical hashtable implementations this is achieved by having a lock per hash-bucket, which leads to low lock contention, since typical workloads involve millions of buckets and the critical section, which consist of inserting a new entry into the hashtable, is very small [32]. No workers modify the hashtable in the probe phase, meaning that no synchronisation is required.

3.1.2 Hardware Conscious Approaches

In terms of algorithmic complexity, the above hardware-oblivious algorithm is optimal without considering radically different approaches. However, Ambuj Shatdal and Chander Kant and Jeffrey F. Naughton [33] show that when the hashtable grows beyond the cache size of the considered architecture, most accesses will result in cache misses. This is due to the random nature of the memory accesses in the algorithm. To improve this behaviour, partitioned hash joins are preferred. The algorithm is extended with an extra partitioning step at the beginning of the build and probe steps. In the build phase, the relation R is split into equal cache-sized partitions using hash function h_1. For each of these partitions a separate hashtable is built, such that the resulting hashtable is now small enough to fit within the cache. In the probe phase, we use h_1 again to partition S. Each of these partitions is then independently probed using the appropriate hashtable constructed during the build phase. The different stages of the algorithm are displayed in Figure 3.3.

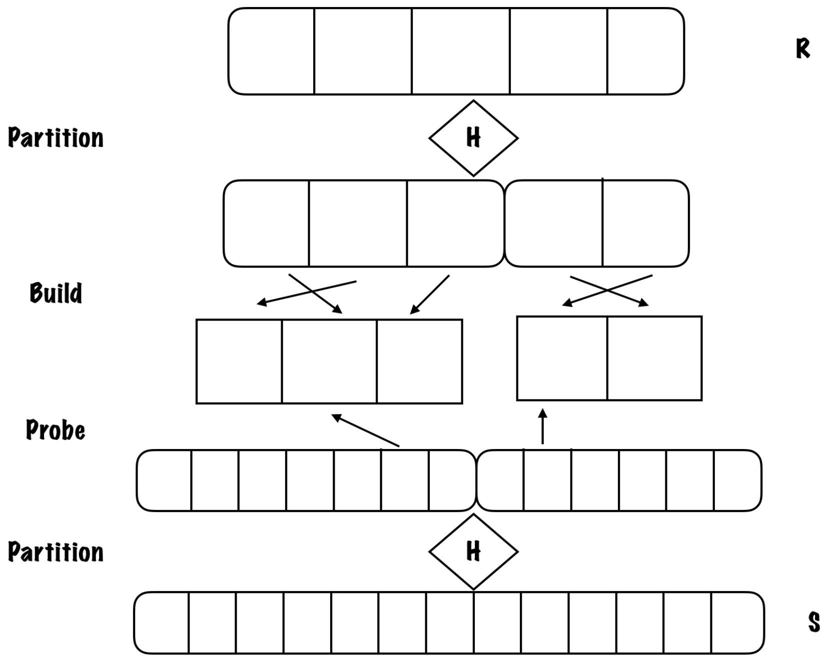


Figure 3.3: Partitioned Hash Join Implementation

The technique presented above performs extra partitioning in the hopes that it will prevent adverse effects related to the data caches. However, the algorithm is at the mercy of a different caching mechanism employed by modern architectures. Indeed, if too many partitions are created, they are likely to reside on many different pages. This means that if there are too many partitions, TLB misses will occur, and main memory will need to be accessed anyway in order to look up the page table. Effectively, the number of partitions that can be processed concurrently is limited by the number of TLB entries available in the system [34].

To prevent the negative effects associated with excessive TLB misses, radix hash joins can be employed. This scheme involves multiple partitioning passes, which make sure the partitioning fan-out never exceeds the number of TLB entries available.

This ensures that at most P pages are open at the same time. At the end of the algorithm, there should be 2^B independent roughly equally-sized partitions of the input relation, where B is chosen such that each partition fits in the second level cache for a representative key distribution. To prevent having more than P pages open at a given time, the partitioning is performed hierarchically. First the input relation is subdivided into P partitions by considering $B' = log_2(P)$ bits of the hash value and clustering entries that share the same radix together. The scheme is recursively applied to each resulting partition by considering the next most significant B' bits of the hashed key until 2^B partitions are generated. In the end, $\lceil B/log_2(P)\rceil$ levels are traversed. [34].

Kim, Changkyu and Kaldewey, Tim and Lee, Victor W. and Sedlar, Eric and Nguyen, Anthony D. and Satish, Nadathur and Chhugani, Jatin and Di Blas, Andrea and Dubey, Pradeep [35] show an algorithm to partition an input relation into 2^B subrelations using B radix bits, where the partitions are as described above. Each partitioning pass requires two scans of the input relation. The steps needed to perform one partitioning pass are described below, where P denotes the desired number of simultaneously open pages:

1. Construct a histogram counting the number of entries whose hashed key matches a particular combination of $log_2(P)$ bits considered in the current pass. The number of observations for each histogram bucket is the size of the resulting output partition. A contiguous memory space is then allocated for the whole reordered output relation.

2. Compute the global prefix sum of the histogram aggregated in the previous step. This way each entry of the prefix sum constitutes a cursor into the output memory space for the relevant partition.

3. The input is scanned again and the tuples are scattered into the correct place in the output relation, as indicated by the prefix sum. The relevant cursor needs to be incremented every time a tuple is written out in order to reflect the new entry in the partition.

Nevertheless, parallelising this algorithm requires some changes when performing the first pass as the histogram, the prefix sum and the output partitions are shared amongst the threads, leading to high contention. Following passes do not raise this issue as there is now a sufficient number of partitions to provide enough independent work in the system for a typical multi-core machine. Kim, Changkyu and Kaldewey, Tim and Lee, Victor W. and Sedlar, Eric and Nguyen, Anthony D. and Satish, Nadathur and Chhugani, Jatin and Di Blas, Andrea and Dubey, Pradeep [35] provide a modified version of the above algorithm to parallelise the first pass more efficiently using task queues:

1. Divide the input relation equally amongst T tasks. Each task computes its own local histogram, as described in the first step of the serial algorithm.

2. Each task aggregates its own version of the prefix sum. Consider the j^{th} index of the local prefix sum. Each task fills this index with the sum of all indices i of all the task-local histograms where $i < j$ and the j^{th} indices for earlier tasks. Each task thus obtains its own cursor into the output relation, which is not shared with any other task.

3. The tasks can now partition their assigned tuples independently, without interfering with each other. The resulting clustering is equivalent to that of the serial algorithm.

3.2 Experience Porting Hash Joins to Xeon Phi

I evaluated a state-of-the-art hash join implementation by porting it to the Xeon Phi, using Intel's MPSS development stack [36]. This allows measurement of any relevant performance improvements that could be achieved through the use of Xeon Phi features.

As a starting point, I chose the implementation provided by Teubner, Jens and Alonso, Gustavo and Balkesen, Cagri and Ozsu, M. Tamer [32], as it includes highly-optimised versions of both the parallel hardware-oblivious and the parallel hardware-conscious algorithms detailed previously. The hardware-oblivious approach is implemented using plain C, whilst the hardware-conscious approach uses SSE vector instructions in the probe phase to improve the performance in a non-portable way.

3.2.1 Porting to Xeon Phi

The advantage of the hardware-oblivious approaches in the previous section is that there is no effort involved whatsoever in porting them to a different system that supports standard C and POSIX features. Porting the code of the non-partitioned join requires no source code changes, apart from building the code using Intel's proprietary toolchain.

The parallel radix join is implemented with the histogram method [35] described previously. Furthermore, the used implementation makes significant use of SSE vector instructions in the probing phase in order to reap even more benefits from the customisation to the underlying hardware. However, the KNC variant of the Xeon Phi only supports its own set of vector instructions and drops support for earlier extensions, such as SSE. I rewrote the probe phase of the parallel radix join in order to measure its performance on the card. This does not uphold Intel's promise to provide a familiar programming environment, even more so as the new vector extensions do not map one-to-one with older formats such as SSE. This observation is only true for this particular variation of the card. Later models, starting from Knights Landing (KNL) support all the other legacy vector instruction formats, as well as the new AVX-512 extensions [37], requiring significantly less effort when porting software.

In line 19 of Algorithm 1, which details the SSE implementation of the probe phase, it may seem odd that only 2 keys are loaded when there are 4 SIMD-lanes available. This is because the vector instruction set used does not have scatter-gather abilities and as such, the payloads of the entries are loaded into lanes .1 and .3. They are not important to the algorithm, since they are unused, so I did not show them.

The Xeon Phi introduces a new set of vector instructions, which features 512 bit-wide vector registers. I optimised Algorithm 1 to make full use of the available vector register width. Now, `match_vec` contains 16 separate counters instead of the 2 that are used in Algorithm 1. Similarly, `key_vec` contains 16 copies of the key that is being searched for. The big improvement comes in the usage of `search_vec`. Beyond

Algorithm 1 Vectorised probe phase

```
def probe(partition S, prefix_sum H, partition R):
    vec4 match_vec = { .0: 0, .1: 0, .2: 0, .3: 0 }
    for(int i = 0; i < S.size/4; i += 4):
        int key_buf[4]
        int hash_buf[4]
        for (int j = 0; j < 4; j++):
            int key = S[i * 4 + j].key
            int idx = hash(key)
            prefetch(R[H[idx]])
            key_buf[k] = key
            hash_buf[k] = idx

        for(int j = i; j < i + 4; j++):
            int k = H[hash_buf[j]]
            int end = H[hash_buf[j+1]]
            vec4 key_vec = load_all(key_buf[j])

            for (; k < end; k += 2):
                vec4 search_vec = { .0: R[k].key, .2: R[k+1].
                    key }
                match_vec += cmp(key_vec, search_vec)

    return add(match_vec.0 + match_vec.2)
```

In the above pseudo-code, vector register initialisation uses a syntax similar to named initialiser lists from C99, where each 32-bit component is indexed starting from 0.

extending it to hold 16 32-bit values, I made use of the scatter-gather instructions available to load the keys for 16 tuples. Now, each loop iteration can process 16 tuples at a time rather than only two.

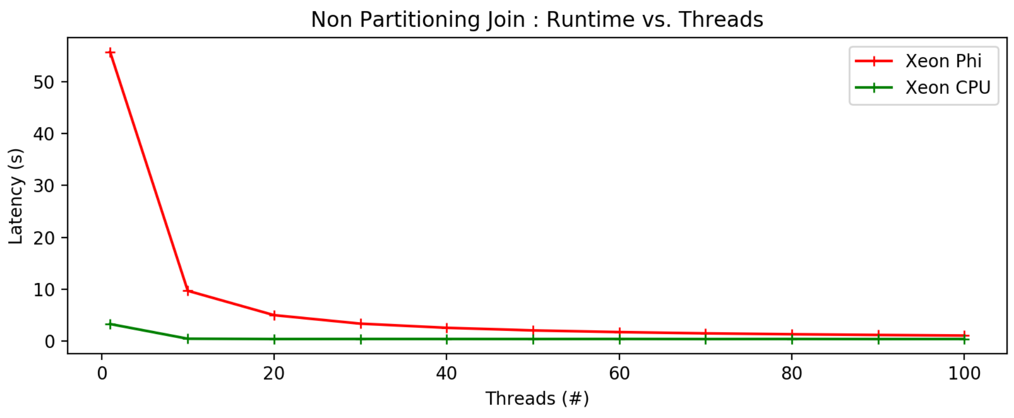
3.2.2 Architectural Features and their Effect

I benchmarked the performance of the two implementations by measuring their runtime on both a native Xeon CPU and on the Xeon Phi, using the largest working set that would fit into the Xeon Phi's main memory. This allowed the entire algorithm to run without data transfers from system main memory, as this gives the most accurate reflection of the performance achievable on both devices. The hardware setup used in this experiment is shown in Table 3.1. I used relations of 10M and 100M tuples as inputs with a uniform key distribution for both algorithms. The radix join implementation uses 18 radix bits and performs two partitioning passes. My findings are shown in Figure 3.4.

Processor	Cores	Clock Frequency	L1 size	L2 size	L3 size	Memory Size
Xeon Phi	57	1.10 GHz	32 KB	512 KB/core (28.5 MB)	N/A	6 GB
Xeon	8	2.60 GHz	32 KB	256 KB/core (2MB)	20 MB	384 GB

Table 3.1: Hardware specifications used in benchmarking

The first notable finding is that, regardless of the chosen algorithm, the performance of the Xeon Phi is very poor with low thread counts. The Xeon Phi is not designed to perform well on single-threaded workloads due to its short in-order pipeline and low clock rate, which definitely showed here. Also, the performance observed on the Xeon CPUs plateaus at around 10 threads, which represents the amount of parallelism available on this platform.



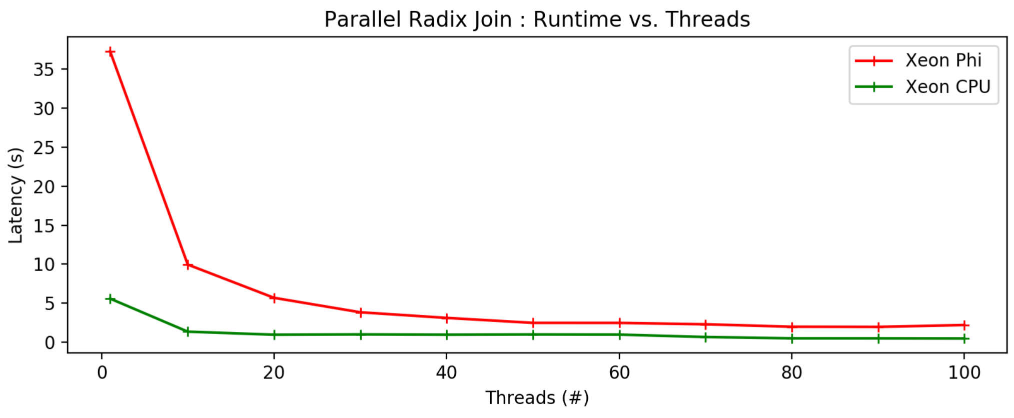


FIGURE 3.4: Hash Join performance for input relation R with 10M tuples and input relation S with 100M tuples. The keys are uniformly distributed in both relations.

However, for both algorithms, the observed runtime on the Xeon Phi approached that of the native Xeon processors when many threads were used.

For non-partitioned joins, high thread counts lead to virtually no performance difference between the two devices. The poor performance characteristics of the individual Xeon Phi cores were completely hidden by the achieved processing throughput in this workload. I believe the slight performance degradation is due to cache miss latency. As explained earlier, non-partitioned algorithms suffer from many cache misses and as such, they rely on the processor being able to mask them through out-of-order execution and hyper-threading. Nonetheless, first level cache misses are more penalising on the Xeon Phi. Indeed, when the L2 cache needs to be accessed, the penalty on the Xeon Phi is roughly 23 nanoseconds [38] as opposed to 15 nanoseconds for an on-chip L2 access on the Xeon CPU [39]. Furthermore, the in-order nature of the Xeon Phi prevents it from hiding this latency as well as the Xeon CPU would. Due to its architecture, the Xeon Phi can only rely on hyper-threading to hide cache latency. However, I found that there was not enough parallel work in the system to justify thread counts above a 100, as the achieved performance did not seem to improve. This limitation was introduced because larger workloads were not able to fit within the memory of the Xeon Phi.

Furthermore, there was a larger performance gap in the radix join workload, despite what seems to be an improvement in the implementation of the probing phase. This might be due to the fact that I did not correctly explore the design space of the number of partitioning passes and radix bits used. Indeed, generating larger partitions, by using fewer radix bits, enables more similar tuples to be accessed together, thus reducing the number of random accesses. However, this reduces the amount of parallelism available in the system, thus exposing the Xeon Phi's poor serial performance in non-vectorised scenarios. This factor alone does not seem to be sufficient to explain the performance discrepancies. An in-depth analysis of the impact of this would be beneficial as the L2 cache of the Xeon Phi is much larger than that of the Xeon CPU, as seen in Table 3.1. However, the L2 cache's distributed nature means that it has a higher access latency. In summary, it does not seem plausible that this factor alone is responsible for the observed performance degradation.

An important optimisation that I did not consider in my evaluation is the usage of huge pages. Jha, Saurabh and He, Bingsheng and Lu, Mian and Cheng, Xuntao and Huynh, Huynh Phung [40] claim that they experienced a 15% improvement in the run time of the parallel radix join by enabling 2 MB pages. Indeed, enabling huge pages further improves the TLB hit rate, as the same number of TLB entries span a larger address range.

3.2.3 Analysis Limitations

Unfortunately, the license for the Intel compiler and runtime libraries I was using expired before I could further tune the algorithm implementations. This is why I was unable to fully explore the possible trade-offs in the radix join implementation, nor was I able to experiment with prefetching in the non-partitioned algorithm. The license had been generously provided by the Systems Group at ETHZ in Switzerland. As they had no need for it any more, they could not justify extending it.

3.3 Summary

I found that using hardware-oblivious techniques presented two main benefits over more hardware-conscious approaches. I briefly list these below:

- Porting non-partitioned approaches to a new card is more straightforward. The Xeon Phi supports standard C programming and features a traditional x86-based architecture. There were no surprises during the porting process, as it just required a recompilation, and the performance was comparable to that of a native CPU. Furthermore, the parallelism was obvious and mapped very well to the model of static partitioning into a large number of independent threads. In contrast, hardware conscious algorithms such as the radix join require concepts such as job queues to expose the required levels of parallelism. This introduces temporal dependencies between tasks in the first pass of the partitioning phase. This makes the threads dependent on the results of earlier tasks, which would lead to huge performance degradations on devices such as GPUs.

- The algorithm does not make many assumptions about the device's memory hierarchy, just that there is a region of memory with access latencies similar to that of a L1 cache. This seems to be a valid assumption, as GPUs have intra-core L1 caches or at least, a texture cache that could be used to fulfil this purpose. By extrapolating, I suspect that this is a desirable feature in a device capable of general purpose computation. The Xeon Phi adopts a similar approach with its core-local L1 cache and the distributed L2 cache. On the other hand, approaches such as radix joins justify the additional partitioning work they perform by improving the hit rate of auxiliary caches such as the TLB, which are quite specific to the unique memory hierarchy of modern multi-core processors.

Given the above points, I claim that hardware-oblivious approaches are a better fit for most accelerators. This is because they are much easier to tune to the underlying accelerator architecture as they make fewer assumptions about the memory hierarchy. On the other hand, radix join-based strategies rely on the specific memory hierarchy that is common place on modern processors. As discussed previously, this does not map well to the Xeon Phi's hardware characteristics.

This hypothesis is confirmed by the work of Pohl, Constantin and Sattler, Kai-Uwe [41]. They found that the high-bandwidth memory available on the more recent Xeon Phi Knights Landing (KNL) accelerator led to improved performance on all join algorithms, but especially for non-partitioned joins. Their evaluation shows that only 10 cycles are necessary to process an output tuple, which demonstrates that memory latency is completely hidden. More specifically, their analysis suggests that the out-of-order nature of the individual cores, the high-bandwidth memory (431 GB/s) available on the die, and the cluster configurations, which let users schedule threads to different regions of the chip using familiar NUMA utility libraries, are the most impactful features of KNL. Their evaluation shows that these features helped speed up the many concurrent reads to the shared hashtable found in non-partitioned joins, which allowed the algorithm scale up to 256 threads (the maximum available on KNL).

Chapter 4

Task Graph Processing in Barrelfish

Throughout this chapter I use an example workload to motivate the features I introduce to my task graph processing framework. The workload I chose consists of a main memory hash join that materialises the output tuple, followed by a sorting stage that sorts the generated output tuples. Figure 4.1 shows a graphical representation of the data flow during this example. Green boxes indicate tasks that are theoretically well suited for execution on the Xeon Phi.

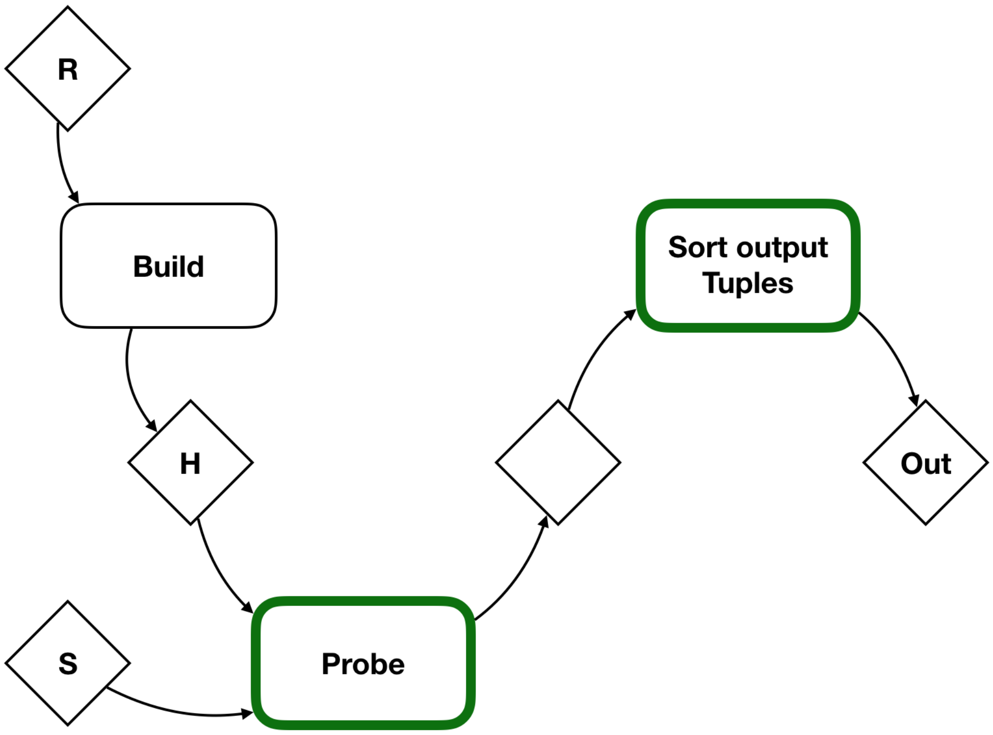


FIGURE 4.1: Hash join followed by sorting of output tuples.

4.1 Barrelfish Programming Background

I implemented my solution for offloading tasks to accelerators within the Barrelfish operating system, the design of which is documented in Chapter 2. Its architecture is akin to that of a microkernel, thus many OS services, such as drivers, run as user-space processes. Thus, Barrelfish exposes interfaces that are very different to those found in Unix-based systems. So far, I have not presented the programming abstractions it offers and the mechanisms one has to use in order to program on Barrelfish. I present here the minimal background required to understand my implementation.

4.1.1 IPC and Flounder

Barrelfish's design philosophy is to embrace the distributed nature of modern multi-core systems, and as such IPC occupies a central role within the system. Traditional IPC mechanisms relying on the availability of shared memory were deemed to be unsuitable abstractions. This is because Barrelfish aims to support a wide range of system architectures, including those where the availability of shared memory is not guaranteed. As such, it aims to expose primitives that are purely based on explicit message passing. The preferred tool for communicating between processes in Barrelfish is Flounder [42]. The tool itself has three main components:

- A DSL that lets programmers specify messages in terms of typed message stubs to be sent along a communication channel. The supported types are normal basic types, such as fixed-width integers, aggregate types specified in a syntax similar to that of a C struct, and capabilities which represent resources in the system. Capabilities are normally sent to establish shared memory between processes. Furthermore, character strings and untyped byte buffers are supported as well, but a length upper-bound needs to be specified for both of them.
- A runtime library, that implements channel creation and destruction, as well as the underlying message transfer mechanisms. Currently, the main source tree supports communication using shared memory, both intra-core and inter-core. More specifically, the inter-core variant uses a shared memory frame which is split into two buffers. Each process uses one buffer for outgoing messages and the other buffer for incoming messages. This way, each process writes its outgoing messages into the other process' reception buffer. Message reception is implemented using a combination of polling and interrupts. Nonetheless, other transport mechanisms have been implemented, but they are not of interest here [42].
- A compiler that generates C definitions and declarations for functions that implement creating and destroying a channel, as well as sending and receiving typed messages using a specified transport mechanism. The compiler operates as part of the application's compilation, which means that all message definitions need to be supplied ahead of time and channels are thus statically typed.

The process of setting up a communication channel is as follows:

1. One of the processes, qualified as the exporting process, needs to *export* the channel and provide two callbacks, the export callback and the connection callback. The runtime takes care of creating a shared memory buffer of the required size and sets up its internal state. At this point, the export callback is invoked and is responsible for registering a table of function pointers that defines handlers to be called when each message type is received. Furthermore, it also needs to publish the channel's unique identifier to the other process.

This is typically done using the *nameservice*, which is a system-wide mapping of character strings to channel identifiers.

2. Once the other process, named the connecting process, gets hold of the channel identifier, it can *connect* to the channel by providing the runtime with a callback and the channel identifier. The runtime library sets up its internal state and the callback is called. This callback is responsible for registering the reception handlers as above, as well as an optional pointer per connection that allows storage of user-defined state.

3. The exporting process's connection callback is now called, so that it can optionally register a pointer per connection to some user-defined state.

4. Both processes can now send each other messages asynchronously.

4.1.2 Existing Xeon Phi Support within Barrelfish

Reto Achermann [43] previously implemented a port of the Barrelfish kernel, runtime libraries and main userspace services to the Xeon Phi. Furthermore, he implemented a driver service that allows users to spawn processes on the Xeon Phi and to share memory capabilities with them. Also, he wrote an OpenMP implementation targeting Xeon Phi hardware.

The driver service is divided into three main components:

1. **Host and Xeon Phi Drivers**: The driver for the card is divided into two components, one running on the host processor, the other on the Xeon Phi. The host driver is more complex and has multiple roles.

   Its first responsibility is to initiate the card's boot sequence and then to transfer control to the card driver for device-specific initialisation. Secondly, the host driver needs to manage the *aperture* region mentioned in Chapter 2. This is critical as it allows the driver's clients to establish shared messaging frames with processes running on the Xeon Phi, by translating host physical addresses into Xeon Phi physical addresses within the aperture. Finally, beyond offering an API to spawn processes and share messaging frames, it exposes a virtual serial device to the card driver. This allows processes running on the card to write to standard output as the Xeon Phi does not have a Platform Controller Hub Southbridge and the associated serial ports.

   The Xeon Phi driver only offers a subset of the functionality of the host driver. More specifically, it exposes the same API to communicate and spawn processes on the system's Xeon Phi cards, and manages the aperture mapping onto the system's main memory. It does not expose a virtual serial device, nor does it allow users to boot a different Xeon Phi.

2. **Driver Manager**: This component is responsible for keeping track of the running Xeon Phi drivers. It assigns each pair of drivers a unique identifier and helps them establish a communication channel by mediating the distribution of an initial shared memory frame, in the aperture space between them.

3. **DMA Service**: The DMA service is a separate userspace driver-like process that exposes an API to perform DMA transactions between the system's main memory and the Xeon Phi's onboard memory. This provides a faster alternative to using the aperture for larger data transfers.

The control flow of the driver service can be seen in Figure 4.2.

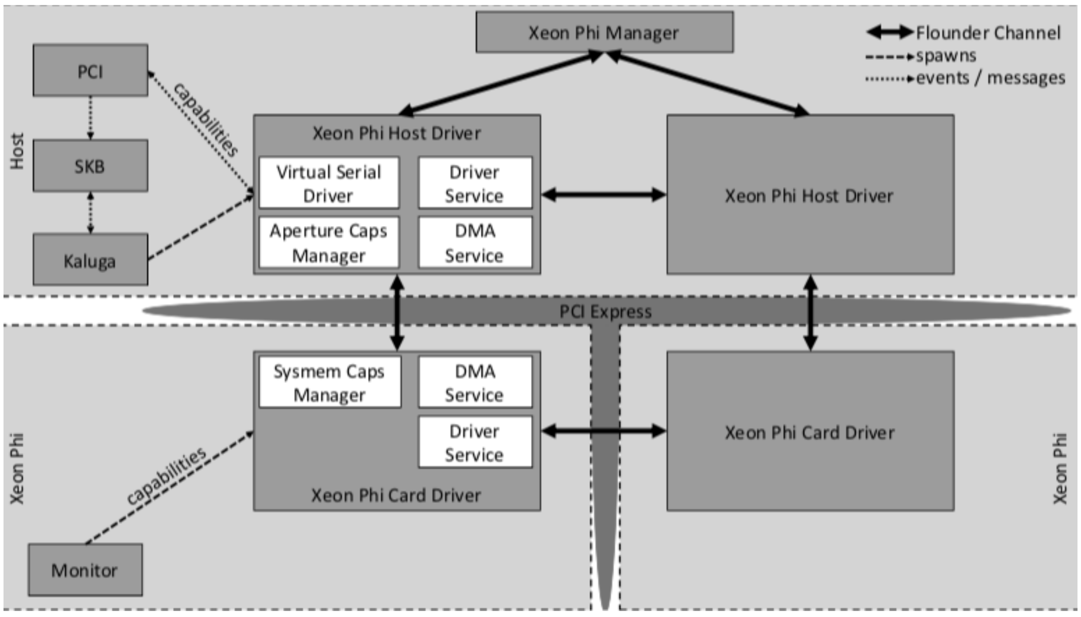


FIGURE 4.2: Barrelfish Xeon Phi Driver Architecture. Reto Achermann, "Message passing and bulk transport on heterogenous multiprocessors", Master's thesis, ETH Zurich, Zurich, Switzerland, 2014

Users can create a communication channel between a process running on the host processor and one running on the Xeon Phi. This is achieved using Flounder's shared memory communication mechanisms as follows. Firstly, a shared memory frame in the aperture needs to be agreed upon between the two processes using the driver service's capability exchange mechanism. Flounder exposes an API that allows the user to specify the frame and the layout of the send and receive buffers used as part of the exporting and connection steps. Once this required bootstrapping step is done, the usual Flounder APIs can be used to send messages, using shared memory in the aperture.

4.1.3 Differences and Similarities with MPSS

Intel provides its own software development stack (MPSS) based on Linux, for leveraging Xeon Phi hardware. It relies heavily on Intel's SCIF framework to send messages and data between Intel CPUs and Xeon Phi accelerator cards, as well as between multiple cards installed on the same system [44]. SCIF exposes to programmers a connection-based communication interface that is akin to socket-based APIs. This provides a higher level of abstraction than that of individual memory reads and writes or DMA transactions. As such, it needs to carefully select the appropriate mechanism, either writing to shared memory in the aperture or issuing a DMA transaction, to perform a data transfer operation.

Despite using approaches similar to those used in Barrelfish, SCIF adds a layer of abstraction and exposes different interfaces to those normally accessible on Unix systems. I believe that using the system's native IPC abstractions instead of a new interface has several advantages. Firstly, programmers do not have to learn and adapt to the new interface and can instead use familiar mechanisms to achieve their goals. More importantly, using native IPC abstraction helps portability immensely. Indeed, the SCIF interface is only available for Intel systems, thus host-only workloads on a system equipped with a processor not manufactured by Intel would require porting SCIF, which would add unnecessary complexity. Furthermore, using different accelerator technologies would also require porting SCIF to the new target, which presents an obvious portability concern.

Finally, SCIF needs to be implemented as a Linux driver running in kernel space, whereas the Barrelfish Xeon Phi drivers and DMA services run as user-space processes. This reduces the amount of kernel involvement, and allows users to explicitly select the used transport mechanism.

I chose to implement my framework within the Barrelfish operating system. This is mainly because programming within the Intel stack would tie my implementation to SCIF and other Intel technologies, which would have hurt the generalisability of my work to other accelerators. Moreover, the ability to implement the framework entirely in userspace made implementing the framework simpler and should lead to better reliability.

4.2 Task Graph API

The model I propose lets users define computational tasks that take up to ten immutable arguments, produce a single output and run on a specified device. Furthermore, users can connect the tasks together by using the output of a parent task as the input to a child task. Thus, users can form directed acyclic graphs of tasks that define a computation. Some components can be executed on the host processor and some can be executed on accelerator hardware. The memory management and movement operations are transparent to the user and are handled efficiently by the framework. This allows users to focus on the data flow through the graph's computation without needing to worry about optimising the data transfers between the computational devices.

The framework I propose relies on three communicating components. The *host client* implements the main logic of the application and is user-defined. It is responsible for creating the task graph, supplying any task definitions that run on the host, and initiating the start of the computation. This is achieved using the API detailed in Section 4.2.2. It can use any task definitions exposed by one or multiple *offload servers* running on accelerator hardware. These offload servers are responsible for executing the tasks they expose, when the tasks are scheduled. They are provided by users as well. Finally, the *task graph service* is the centralised component within the system that is charged with coordinating the creation and execution of a task graph. It fulfils client requests and thus is charged with memory management and data transfer operations, as well as scheduling the different tasks to run at the right time.

4.2.1 Why Offload Servers?

I chose to implement explicitly separate processes for offloaded tasks and to use a library to expose the framework's interface, as opposed to using preprocessor directives or compiler pragmas within a single code base. There were two main reasons guiding this decision, both of which are very pragmatic.

Firstly, supporting the latter paradigm would involve adding an additional preprocessing step or embedding support directly into the compiler which would have significantly complicated the framework's implementation. Furthermore, this would significantly reduce the portability of the system as it would introduce extra dependencies on the chosen compiler. This prevents users from using different compilers for their host local and offloaded components and simplifies using different programming languages.

Besides this, using separate processes allows a much more distinct separation of the code that executes on the accelerator. This simplifies distribution of optimised implementations of tasks that are known to be suitable for the chosen accelerator. For example, one can imagine highly-optimised versions of the probe phase of a hash join, as well as the sorting routines to be distributed as a pre-compiled offload server. This allows the various offload servers and the host client to only rely on each others semantics, as opposed to the alternative scheme of exposing everything as a library where ABI compatibility and other such concerns need to be taken into consideration. This is particularly important as the Xeon Phi uses an ABI that is incompatible with the usual Intel x86 64-bit one [12].

4.2.2 API Presentation

Concepts

The API exposed to users revolves around four main concepts, which are detailed below:

- **Execution Contexts**: Execution contexts are different device types on which a task can be executed. They are represented by the C enumeration `tg_execution_ctx_t`. Currently only the `HOST` and `XEON_PHI` values are supported. This enumeration should be extended if and when the framework supports new devices.
- **Registered Tasks**: Registered tasks are records indicating the availability of a named computation in a graph within an execution context. The record maps a well-known character string to a computational component. Each registered task declares up to ten arguments with a specified maximum size. It automatically produces a single output. The scheme is heavily inspired by general purpose RPC systems.
- **Sinks**: Sinks are memory buffers that can be used as a task's input, output or as both output and input. They are implicitly created when a registered task is instantiated. They are symbolically represented by the `tg_sink_handle_t` type when the user manipulates the graph's topology. When a task is executed, sinks are distributed using an untyped pointer referring to their location in the process's virtual address space, as well as a value indicating the size of the buffer.
- **Graphs**: Graphs are a collection of instantiated tasks with a directed acyclic graph topology. They are represented by the `tg_handle_t` type. Graphs are normally associated with a host client, a set of offload servers and a set of registered tasks.

```c
//Initialises the library's internal state
errval_t tg_client_init(void);

//Creates a new task graph
errval_t tg_client_create(uint64_t heap_sz, tg_handle_t *handle);

//Registers a task as available for execution in this client
errval_t tg_client_register_task(const char *task_name,
        const uint64_t *max_sizes, const uint8_t argc,
        const uint64_t out_max_sz, tg_task_exec_fn fn,
        tg_execution_ctx_t exec_ctx, tg_handle_t handle);

//Creates a task within a graph
errval_t tg_client_create_task(const char *task_name,
        tg_execution_ctx_t exec_ctx, tg_handle_t handle,
        tg_sink_handle_t *argv, size_t argc,
        tg_sink_handle_t *out_sink_handle);

//Gets information about the sink's memory buffer
errval_t tg_client_get_sink_info(tg_sink_handle_t sink_handle,
        tg_handle_t handle, tg_client_sink_t *sink);

//Allocates memory for all the sinks in the graph
errval_t tg_client_alloc_sinks(tg_handle_t handle);

//Start running the task graph
errval_t tg_client_run(tg_handle_t handle);

//Cleans up the graph
errval_t tg_client_remove(tg_handle_t handle);

//Spawns a new offload server in the provided execution context
errval_t tg_client_spawn_offload_domain(tg_execution_ctx_t ctx,
        char *path, char *argv[], uint8_t spawn_flags,
        tg_handle_t handle);
```

Listing 4.1: Task Graph Host Client API

Host Client API

The task graph client running on the host is responsible for spawning the desired offload servers, defining the task graph topology and providing data for the sinks that compose the graph's entry points. Furthermore, it needs to initiate the start of the computation.

The above responsibilities are supported with the API presented in Listing 4.1.

A typical client will first call `tg_client_init` to let the library initialise its internal state. The next call would then be to `tg_client_create` in order to create a new task graph with the desired `heap_sz` amount of memory reserved for sinks. This memory region is referred to as the sink heap. I show these calls below in the context of the join and sort example.

```

    tg_handle_t handle;
    size_t heap_sz = 2 * sizeof(struct join_rel) + 2 * sizeof (
        struct join_payloads)
        + sizeof (struct join_hashtable);
    err = tg_client_create(2 * heap_sz, &handle);
```

Next, the client can optionally spawn a number of offload servers using the `tg_client_spawn_offload_domain` function. It can then optionally register any tasks that should run on the host processor by calling `tg_client_register_task`. Again, the relevant snippet can be found below.

```
    err = tg_client_register_task("build_task", &max_size, 1,
            sizeof (struct join_hashtable), &build_task, HOST,
                handle);

    err = tg_client_spawn_offload_domain(XEON_PHI, "k1om/sbin/tests
        /join",
        NULL, 0, handle);
```

At this point the client can start creating tasks, thus defining the task graph's topology. To create a new task the client invokes `tg_client_create_task`. The input arguments to the call include `task_name`, `exec_ctx` and `handle`. These help to locate the task. The `argv` argument and the related `argc` arguments let the client provide the input sinks for this task. The semantics are that a value of `0` signifies that a new sink should be created to represent this positional argument. The newly minted sink is returned in-place within the `argv` array. The new sink is an entry point to the graph and needs to be filled in by the user before running the graph. Any other passed value is interpreted as an existing sink identifier and the data contained by the referred sink will be passed on to the new task. The call returns a sink identifier for the output sink of the new task instance in the `out_sink_handle` argument. This means that task graphs need to be built in a breadth-first fashion, so that the user has sink identifiers for all of a task's dependencies. The code that creates the tasks in the hash join and sort example is presented below.

```
    size_t in_argc = 1;
    tg_sink_handle_t out_handle;
    err = tg_client_create_task("build_task", HOST, handle, in_argv
        ,
             in_argc, &out_handle);

    tg_sink_handle_t *in_argv_2 = calloc(2, sizeof (
        tg_sink_handle_t));
    size_t in_argc_2 = 2;
    in_argv_2[0] = out_handle;
    tg_sink_handle_t out_handle_2;
    err = tg_client_create_task("probe_task", XEON_PHI, handle,
        in_argv_2,
             in_argc_2, &out_handle_2);

    tg_sink_handle_t *in_argv_3 = calloc(1, sizeof (
        tg_sink_handle_t));
    size_t in_argc_3 = 1;
    in_argv_3[0] = out_handle_2;
    tg_sink_handle_t out_handle_3;
    err = tg_client_create_task("sort_task", XEON_PHI, handle,
        in_argv_3,
             in_argc_3, &out_handle_3);
```

Once the graph is constructed, the client needs to invoke `tg_client_alloc_sinks` . This ensures that space for all the sinks is reserved within the sink heap and implicitly marks the finalisation of the graph. The client can now call `tg_client_get_sink_info` to obtain pointers to the entry sinks. The client is then required to fill in the entry sinks with the correct data, by writing to the associated buffer. This call returns a `tg_client_sink_t` struct in the third argument, which tells the user the length and location of the memory buffer. I give the definition of this struct below.

```
typedef struct tg_client_sink {
    void *mem;
    uint64_t length;
} tg_client_sink_t;
```

Once this is done, the client can finally call `tg_client_run` to initiate the beginning of the graph's execution. The task graph service then takes over and appropriately schedules the various tasks and required memory transfers in a timely manner.

In Listing 4.2, I present a simple, complete example of the API in action, where the task graph API is used to compute the sum of two integers on the host processor, using only one simple task definition. Furthermore, the control and data flow of this simple example is shown in Figure 4.3.

```c
static errval_t simple_task(const tg_client_sink_t *argv, const
    size_t argc,
        const tg_client_sink_t out)
{
    int *arg1 = (int *) argv[0].mem;
    int *arg2 = (int *) argv[1].mem;
    *((int *) out.mem) = *arg1 + *arg2;
    return SYS_ERR_OK;
}

int main(int argc, char *argv[])
{
    tg_client_init();
    tg_handle_t handle;
    tg_client_create(2 * sizeof (int), &handle);

    const uint64_t max_sizes[2] = { sizeof (int), sizeof (int) };
    tg_client_register_task("simple_task", max_sizes, 2, sizeof (
        int),
            &simple_task, HOST, handle);

    tg_sink_handle_t *in_argv= calloc(2, sizeof (tg_sink_handle_t))
        ;
    size_t in_argc = 2;
    tg_sink_handle_t out_handle;
    tg_client_create_task("simple_task", HOST, handle, in_argv,
        in_argc,
            &out_handle);

    tg_client_alloc_sinks(handle);

    tg_client_sink_t arg1;
    tg_client_get_sink_info(in_argv[0], handle, &arg1);
    *((int*) arg1.mem) = 4;

    tg_client_sink_t arg2;
    tg_client_get_sink_info(in_argv[1], handle, &arg2);
    *((int*) arg2.mem) = 5;

    tg_client_run(handle);

    tg_client_sink_t out_sink;
    tg_client_get_sink_info(out_handle, handle, &out_sink);
    int *res = (int*) out_sink.mem;
    printf("The␣result␣is␣%d!\n", *res);

    return 0;
}
```

Listing 4.2: Minimal Task Graph Example

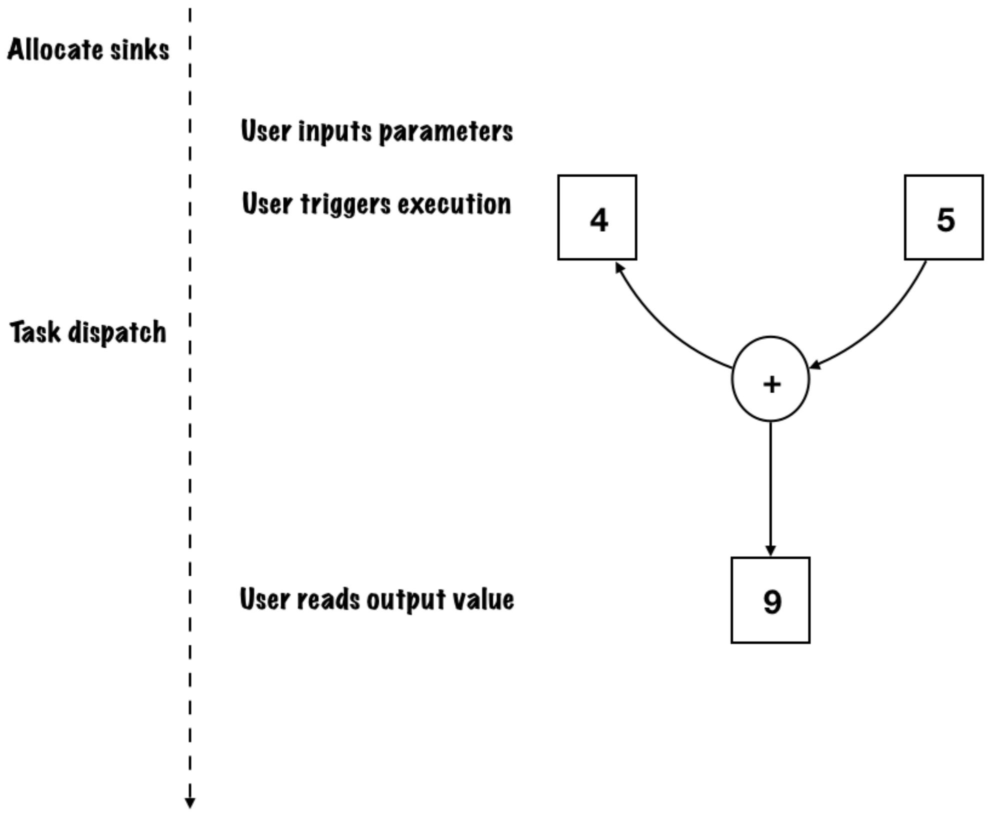


FIGURE 4.3: Minimal task graph API usage example.

Offload Server API

The offload servers support only a limited subset of the host clients's functionality. The only allowed behaviours are to initialise a connection with the task graph service, as well as to register any tasks the server wishes to expose to the task graph service and the associated client.

The handle to the associated graph is provided as the first positional command line argument to the offload server process. The offload server only has to make a single call to the revamped initialisation function shown below:

```
errval_t tg_client_init(tg_handle_t handle, tg_offload_register_fn reg_fn,
size_t heap_sz).
```

This provides the library with:

- The graph handle associated with this offload server.
- A callback that is responsible for registering all the tasks exposed by this process.
- The amount of memory to reserve for the sinks associated with this offload server's tasks.

At this point, the library sets up all its internal state, triggers the registration function, and then notifies the central task graph service that all the tasks have been correctly registered. The registration function typically just makes a series of calls to `tg_client_register_task` as would be the case in a host client. The offload server of the join and sort example registers the probe and the sort task. The relevant registration function is shown below.

```c
static errval_t reg_fn(const tg_handle_t graph_handle)
{
    errval_t err;

    uint64_t max_sizes[2] = { sizeof (struct join_hashtable),
       sizeof (struct join_rel) };
    err = tg_client_register_task("probe_task", max_sizes, 2,
            sizeof (struct join_payloads), &probe_task, XEON_PHI,
               graph_handle);
    if(err_is_fail(err)) goto error;

    uint64_t max_size = sizeof(struct join_payloads);
    err = tg_client_register_task("sort_task", &max_size, 1,
            sizeof (struct join_payloads), &sort_task, XEON_PHI,
               graph_handle);
    if(err_is_fail(err)) goto error;

    return SYS_ERR_OK;
error:
    return err;
}
```

4.3 Implementation

In order to support the API described in the previous section I had to implement four main components — the two libraries for the host clients and offload servers, the central task graph service, and a support library to marshall and unmarshall details about a task's input and output sinks. This last component was very small and was only needed as a workaround to Flounder not yet supporting the transfer of typed buffers. Thus, I won't detail the implementation of this component here.

4.3.1 Task Graph Client Libraries

Host Client Library

The host client library implements its API by delegating requests to the central task graph service through a Flounder channel. This channel is set up during the initial call to `tg_client_init`. However, I wanted the API to be synchronous so that return values, such as sink identifiers, are communicated immediately to the client. This way, the client code that describes the graph topology can be sequential, instead of the control flow being obfuscated by several levels of nested callbacks. In order to achieve this, I implemented a simple RPC mechanism on top of the existing Flounder abstractions. As part of the state associated with a connection to the task graph service, I maintain a volatile flag that represents the state of a RPC to the task graph service and two values to represent the results of such calls. The first value simply contains the error code of the task graph service's response, whilst the second is a union of structs, representing the other return values associated with each API function. I show these in the form of an independent struct, as well as the associated control functions in Listing 4.3.

The library code uses `rpc_start` in order to yield until the previous RPC is completed, and calls `rpc_wait_done` to wait for the results coming from the task graph service. The message handler associated with the task graph service's response runs on a different thread and invokes `rpc_done`, after it has populated the `rpc_err` and `rpc_data` fields. This allows the thread that is handling the current API call to wake up and return the call results to the client code, before calling `rpc_clear`, which allows another RPC cycle to begin.

Most API calls simply delegate to the task graph service, and as such are just means to synchronise accesses to the service's interface. However, `tg_client_register_task` is special in this respect. Beyond communicating the request to the task graph service, the library maintains a mapping between task names and function pointers for the correct user-defined task execution function. This allows the library to correctly dispatch task execution requests initiated by the task graph service to the appropriate function.

Furthermore, the library is responsible for establishing the shared memory region that backs the sink heap for a graph. I implemented this as part of `tg_client_create`. The user provides the required heap size to store all the sinks. Using this information, the library requests an appropriately sized physical memory frame[1]. It then transfers the capability associated with this frame to the task graph service as part of the graph creation message. Finally, the frame is installed in the host client's virtual address space, and a pointer to the virtual base address of the heap is recorded. Now, sink information is communicated by the task graph service in terms of an offset into the sink heap, which allows the library to construct pointers to the memory buffer associated with a specific sink. This is done so that client code can read from and write into the sink without the need to expose a special API. I could have performed the sink heap frame allocation within the task graph service. However, performing this step in the library allows failing early and returning to the user without the need to communicate with the task graph service. This would reduce the load on the service in an environment with many applications using the task graph API.

Offload Server Client Library

Currently, the only supported target of the offload server client library is the Xeon Phi. As explained previously, the supported API is more restricted. However, the initialisation of the library is more complex on the Xeon Phi. I detail the steps required below:

1. The host client calls `tg_client_spawn_offload_domain` to start the procedure. This call delegates to the task graph service.
2. The task graph service uses the Xeon Phi driver service to spawn the offload server process on the Xeon Phi.

[1]Barrelfish uses self-paging mechanisms instead of more common virtual memory implementations. In this scheme, “frames” do not have a fixed size but instead represent a power of two sized contiguous block of physical address space.

```c
struct rpc_state
{
    volatile uint8_t rpc_wait_reply;
    union
    {
        struct
        {
            tg_handle_t handle;
            struct capref heap;
        } create;
        struct
        {
            tg_sink_handle_t *argv;
            size_t argc;
            tg_sink_handle_t out_sink_handle;
        } create_task;
        struct
        {
            uint64_t offset;
            uint64_t length;
        } get_sink_info;
    } rpc_data;
    errval_t rpc_err;
};

void rpc_start(struct rpc_state *st)
{
    while (!st->rpc_wait_reply)
    {
        st->rpc_wait_reply = 0x1;
        messages_wait_and_handle_next();
    }
    return 0;
}

void rpc_wait_done(struct rpc_state *st)
{
    while (st->rpc_wait_reply == 0x1)
    {
        messages_wait_and_handle_next();
    }
}

void rpc_done(struct rpc_state *st)
{
    st->rpc_wait_reply = 0x2;
}

void rpc_clear(struct rpc_state *st)
{
    st->rpc_wait_reply = 0x0;
}
```

Listing 4.3: Library RPC implementation

3. The offload server parses the first positional command line argument, as it represents the graph identifier associated to this offload server.
4. It then registers itself with the Xeon Phi driver service, which lets the task graph service know that the offload server process has started.
5. The task graph service sends a capability to a frame in the aperture space. This messaging frame is now used by both sides to establish a Flounder communication channel between the offload server and the task graph service.
6. Now the task graph service registers the sink heap with the DMA service and sends the capability to the offload server, which records this information.
7. The offload server now allocates a frame the same size as the sink heap frame and installs it in its virtual address space. This frame is going to be used to mirror the sink heap. The frame is now registered with the DMA service.
8. The offload server sends the task graph service a capability to this heap mirror. The task graph service records this information.
9. At this point, the offload server invokes the user-provided task registration callback. Task registration on the offload server follows the same scheme as on the host client. Once it completes and the available tasks have been registered with the task graph service, the offload server notifies the task graph service.
10. When the task graph service receives the notification, it responds positively to the host client.

4.3.2 Task Graph Service

The task graph service is the main component in the framework. It is responsible for recording the available tasks, managing the graph data structure and the sink heap, as well as computing a schedule for graph execution. Furthermore, it needs to coordinate the data transfers to the various execution contexts as well as dispatching the tasks according to the schedule.

The task graph is composed of three modules that have the following responsibilities:

- Handling the communication with the various clients.
- Implementing and managing the task graph data structure and the associated schedule.
- Managing the sink heap.

Sink Allocator

The initial sink allocator implementation is quite simple. It first receives the heap size and the largest alignment constraints on the sink buffers (for example DMA transfers to the Xeon Phi require 64 byte alignment). The implementation follows a simple bump pointer allocation scheme, where every sink that gets allocated adds its size to the bump pointer. The bump pointer is the offset into the sink heap that represents the start of unallocated memory. I used this approach to reduce allocation overhead, as all the sinks are potentially needed throughout the lifetime of the graph

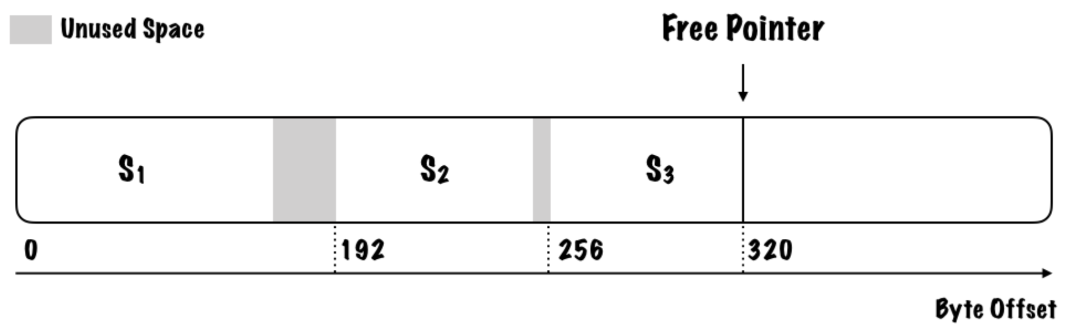


FIGURE 4.4: Example sink allocation using a simple bump pointer allocation strategy.

execution. Indeed, if all sinks are needed at all times, it is not necessary to free the associated memory during graph execution. Thus, the framework does not face the fragmentation issues inherent to bump-pointer allocation schemes.

It would be possible to reduce memory usage by freeing sinks once all the tasks that use them as inputs are completed, but this would require analysing the task graph topology in order to determine suitable points in the execution to do so. This could be a future optimisation which is further discussed in Chapter 6. The current scheme is shown in Figure 4.4.

Task Graph Data Structure

The graph module maintains the task graph data structure. Internally, no memory allocations are needed to add tasks to the graph. In order to achieve this, I have set a hard limit on 255 sinks within a graph. The graph contains a one-indexed array of 255 sinks. The index of an array element corresponds to the sink's identifier for simplicity. The sink data structure is shown in Listing 4.4.

```
struct tg_sink
{
    struct tg_sink_info info;
    struct tg_task_list *readers;
    struct tg_task writer;
    struct tg_sink_metadata metadata;
};
```

LISTING 4.4: Sink Data Structure

The fields that build up the graph topology are `readers` and `writer`. Each sink maintains the task that writes to it in the `writer` field. If the sink is an entry point to the graph, `writer` is left uninitialised, but instead the `is_entry` flag is set. An example graph topology and its relation to the sink layout is shown in Figure 4.5. The other fields maintain meta-data related to the associated memory buffer, the scheduler, and the availability of the sink in various contexts. Furthermore, the graph itself maintains a list of its entry points to avoid having to traverse the entire sink array to identify the appropriate entry sinks.

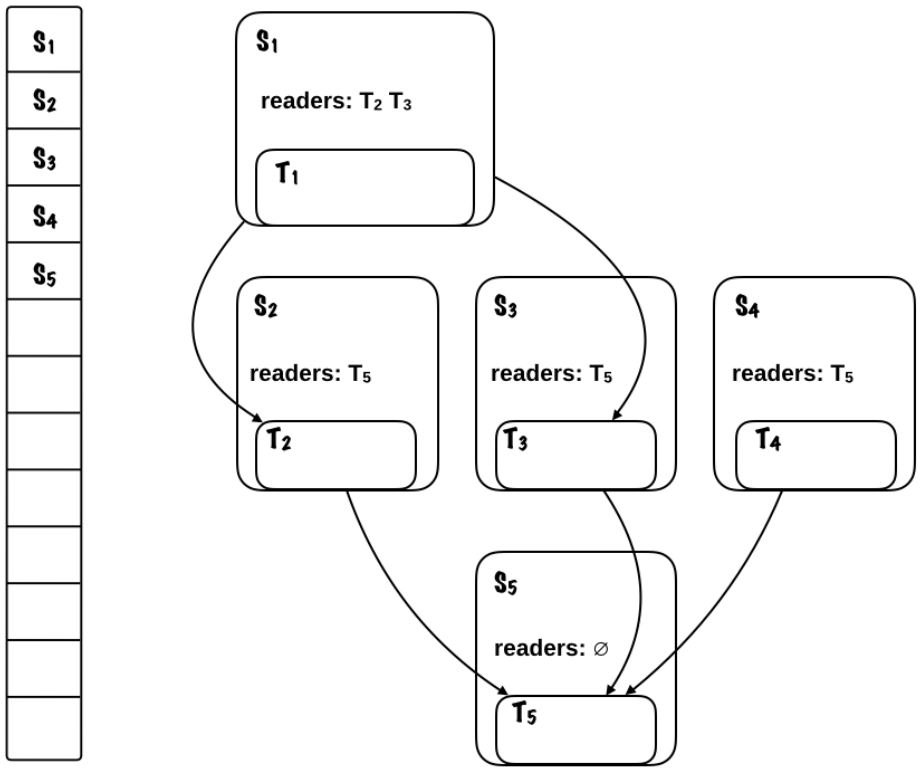


Figure 4.5: Sink layout within a task graph data structure and how it relates to graph topology.

Scheduler

Before it runs, the task graph is responsible for computing a static schedule that provides a topological ordering of the tasks. This ordering guarantees that, for any given task, its parents are scheduled before it. The algorithm for computing the schedule is presented in Algorithm 2.

I now give an informal proof of why the procedure works. Entry point sinks are added to the queue at the start the procedure in line 5. These do not depend on previous task completions, so this is allowed. Other sinks are added to the queue if and only if their writer task has been already scheduled. This is guaranteed by the sequential ordering of lines 13 and 15 respectively. Furthermore, tasks are only scheduled when all their dependencies have been seen. This is due to the fact that sinks are only added to the queue once (as seen in line 14) and that a task is only scheduled once the appropriate number of input sinks have been seen. This is shown in line 12.

The schedule is exposed through an iterator-like interface:

```
bool get_next_scheduled_task(struct tg_graph *graph, struct tg_schedule **sched
, struct tg_scheduled_task **ret, errval_t *err)
```

The call to `get_next_scheduled_task` returns `true`, as long as there are more tasks to run. If there are more tasks to run, the output parameter `ret` will contain a pointer to the next task. The `sched` parameter is an opaque pointer into the schedule queue that allows the function to maintain state across invocations.

Algorithm 2 Computation of the static schedule

```
1: procedure COMPUTE SCHEDULE(graph)
2:     $q \leftarrow$ new Queue<Sink>()
3:     $sched \leftarrow$ new Queue<Tasks>()
4:     for all $e \in$ graph.entryPoints do
5:         append $e$ to $q$
6:     end for
7:     while $q$ is not empty do
8:         $s \leftarrow q$.top()
9:         if $s$ has not been seen then
10:            for all $t \in s$.readers do
11:                increment $t$.visits
12:                if $t$.visits == $t$.numArguments then
13:                    append $t$ to $sched$
14:                    if $t$.output has not been seen then
15:                        append $t$.output to $q$
16:                    end if
17:                end if
18:            end for
19:            mark $s$ as seen
20:        end if
21:    end while
22:    return $sched$
23: end procedure
```

When it is a task's turn to run, the task graph service copies the input sinks to the appropriate device using the frame information provided by the offload server. These transfers are performed using DMA transactions. This is only necessary if the task does not run on the host. Once this optional memory transfer step is completed, the service sends a control message to the correct client, containing the task name and information about the relevant sinks' offsets within the sink heap. This way, the client library can construct pointers to the sinks' memory buffers and use its local task name mapping to dispatch the call to the appropriate user-defined function. Upon the task's completion, the client library sends a notification to the task graph service, which handles it asynchronously on a separate thread. This mechanism allows all tasks whose dependencies have been completed, to be dispatched at once, thus creating as much parallelism as possible within the system.

Chapter 5

Evaluation and Optimisations

I performed my evaluation using the same hardware set-up as in Chapter 3. As a refresher, I detail the hardware specification of the environment using one Xeon Phi accelerator card connected over PCIe to a Xeon CPU in Table 5.1.

Processor	**Cores**	**Clock Frequency**	**L1 size**	**L2 size**	**L3 size**	**Memory Size**
Xeon Phi	57	1.10 GHz	32 KB	512 KB/core (28.5 MB)	N/A	6 GB (240 GB/s)
Xeon	8	2.60 GHz	32 KB	256 KB/core (2MB)	20 MB	384 GB (51 GB/s)

Table 5.1: Hardware specifications used in benchmarking

I evaluated the framework's ability to improve existing applications using three microbenchmarks I created. These benchmarks check that the framework enables three key improvements — using additional devices to increase parallelism in the system, not introducing too much overhead and transferring memory optimally. Each measurement point in these benchmarks was the median of 15 independent runs. Furthermore, there was very little uncertainty in my experiements as the relative difference between the maximum and minimum values for each benchamrk was less than 0.1%.

5.1 Enabling Parallelism

The first microbenchmark aims to check that asynchronous task dispatch described in Section 4.3.2 enables additional parallelism.

The benchmark I propose involves sorting a large array of 650,000 random integers. Firstly, I implemented a serial host-only version of the algorithm. Here, the host first uses the selection sort algorithm to sort the first 500,000 integers. It then repeats the same procedure for the remaining 150,000 elements. At this point the two sorted runs are merged by a separate task using the well-known merging step of the mergesort algorithm.

The parallel version of the benchmark uses the available Xeon Phi card as a source of additional parallelism. Instead of sorting the two array partitions serially, the first partition is sorted on the host-processor, whilst the second is sorted on the Xeon Phi. The same merging step is then performed. I show the task graph topology for both

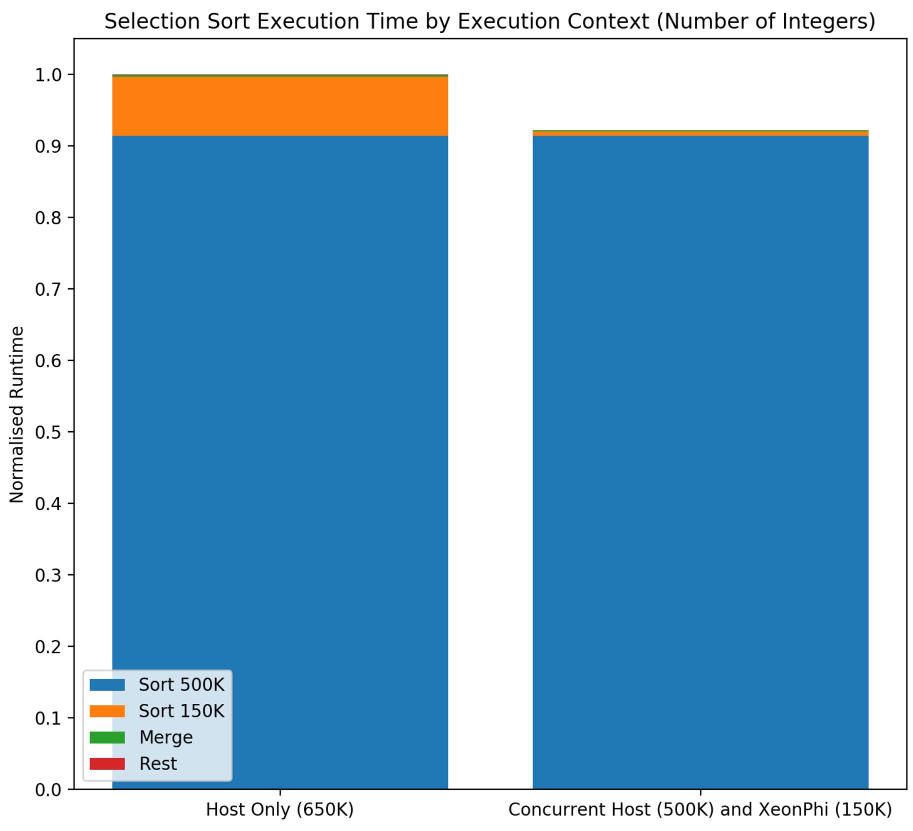


FIGURE 5.1: Selection Sort of a large array of integers both serially on the host processor and in parallel on the host and the Xeon Phi.

variants in Figure 5.2, and I show the achieved performance in Figure 5.1. I implemented the selection sort algorithm serially in order to simplify the benchmark implementation, as it only aims to show that both the host CPU and the Xeon Phi can be used concurrently to improve latency.

The results are quite promising. The parallel version runs roughly 10% faster than the serial version, as it is able to perform 23% of the sorting work concurrently on the Xeon Phi card. However, the speed-up is not perfect due to the following factors:

- As I will show later, the overhead of transferring one sink to a task running on the Xeon Phi is twice as high as that of one being communicated to a task running on the host.
- The proportion of offloaded work is low. In general, moving more work to the accelerator will result in a larger performance improvement. This is necessary because the free GCC version I used to compile the offload server does not support vector instructions. Thus the serial Xeon Phi portion workload performs poorly due to its in-order memory architecture and low clock rate.

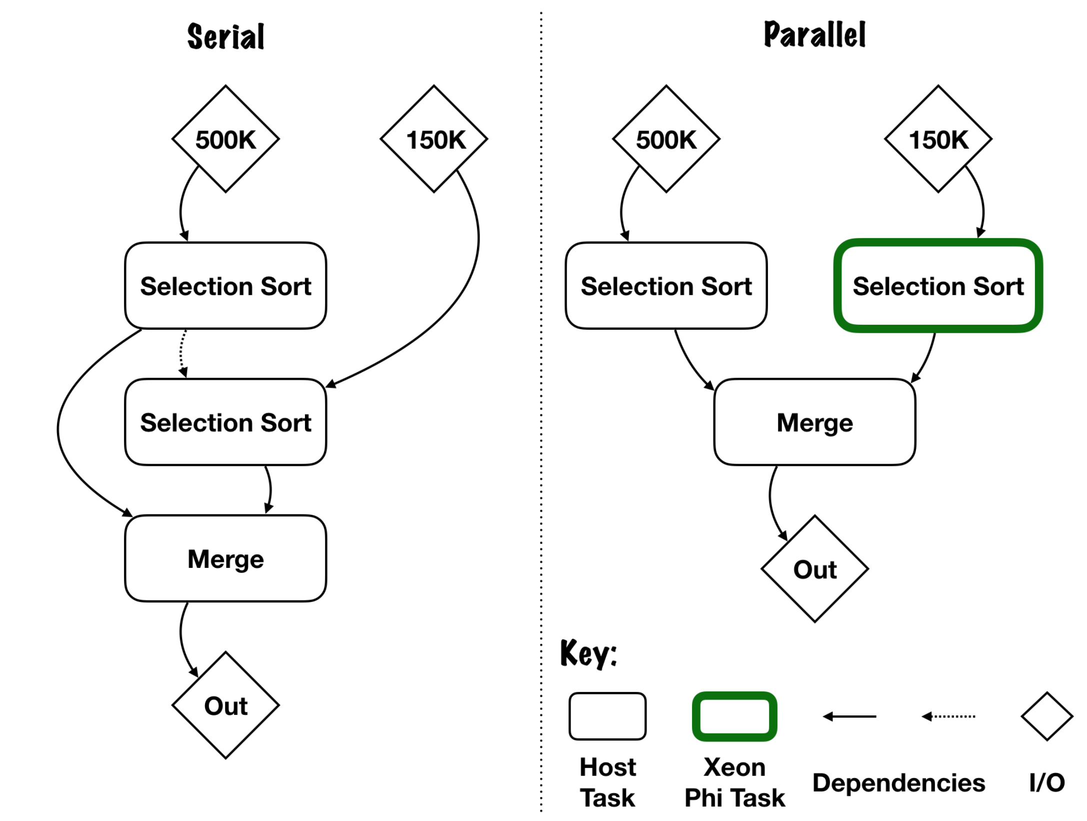


FIGURE 5.2: Task graphs representing sorting a large array serially on the host, and in parallel both on the host and the Xeon Phi.

5.2 Framework Overhead

Beyond showing the framework's capabilities, I wanted to evaluate the various overheads introduced by the framework during the execution of a task graph. In order to achieve this goal, I synthesised a pair of simple microbenchmarks, one for host-only execution and one for execution on the Xeon Phi. These benchmarks are very simple, as they only create a null task (one that immediately returns) on their respective execution context and execute it. I introduced artificial arguments in order to measure the overhead of transferring sinks to a task. My findings are detailed in Table 5.2 for host execution and Table 5.3 for execution on the Xeon Phi.

Bytes/argument	1 argument	3 arguments	6 arguments	9 arguments
10	0.183	0.218	0.280	0.341
1,000	0.183	0.218	0.279	0.341
10,000	0.183	0.218	0.280	0.341
100,000	0.183	0.218	0.279	0.341
1,000,000	0.183	0.218	0.279	0.341

TABLE 5.2: Empty task execution time (s) on the host processor by number of arguments.

There is not much to note about the results observed in host-only execution. The sinks lie in memory that is shared with the task graph service. Thus, there are no data movements in this benchmark, it just establishes the overhead of the necessary control messages in the host context.

Bytes/argument	1 argument	3 arguments	6 arguments	9 arguments
10	0.260	0.336	0.450	0.564
1,000	0.261	0.337	0.451	0.565
10,000	0.261	0.337	0.450	0.565
100,000	0.261	0.337	0.451	0.564
1,000,000	0.261	0.337	0.450	0.565

TABLE 5.3: Empty task execution time (s) on the Xeon Phi by number of arguments.

When executing this benchmark on the Xeon Phi, one would expect the overhead to be higher, as both the control messages and the sinks have to be transmitted over the PCIe boundary. Intuitively I was expecting some overhead for sending multiple arguments, as the framework performs a separate DMA transaction per argument sink. Furthermore, I imagined the size of the memory transfer would play a significant role in the observed performance drop. However, the results shown in Table 5.3 disprove this hypothesis. It seems that the observed overhead comes solely from the number of arguments, whilst their size seemingly has little effect. This is particularly surprising considering the fact that the arguments are sent independently and synchronously to the Xeon Phi before the task dispatch message is ever sent. There are two plausible explanations for these findings:

1. The observed overhead is dominated by the Flounder control message that goes through the aperture. Indeed, the control message size grows with the number of arguments as more sink offsets need to be communicated across.

2. The cost of setting up a DMA transaction far outweighs the cost of performing the actual memory copy to the Xeon Phi.

The first hypothesis appears unlikely, given the fact that the control message is, at most, 2000 bytes large.

5.2.1 Grouping Sink Transfers

Reducing the size of the control message is not feasible. However, if the input sinks of a task can be allocated contiguously, then it becomes possible to transfer them to the Xeon Phi with a single DMA transaction. In order to achieve this, I refined the sink allocator component. It now takes into account the schedule and the destinations of each sink. The scheme still uses bump pointer allocations, but instead of allocating sinks on a first come first serve basis, the allocator traverses the graph's schedule and applies Algorithm 3 to decide when to allocate a sink.

This algorithm groups together all the unallocated inputs of a task that executes on the accelerator. If an input has already been allocated, it means that it will have already been sent to the accelerator via an earlier task dispatch operation. This algorithm guarantees that all tasks that execute on the Xeon Phi only require a single DMA transaction to transfer their input sinks. The input sinks of a task that executes

Algorithm 3 Refined sink allocation algorithm

```
1:  procedure AllocateSinks(schedule)
2:      for all task ∈ schedule do
3:          if task executes on the Xeon Phi then
4:              Allocate all input sinks that have not been allocated yet.
5:          else
6:              for all sink ∈ task.inputs do
7:                  if Every task that reads sink executes on the host then
8:                      Allocate the sink.
9:                  end if
10:             end for
11:         end if
12:     end for
13: end procedure
```

on the host are scattered through the sink heap, but this is not problematic as the entire sink heap memory is shared with the host client.

Bytes/argument	1 argument	3 arguments	6 arguments	9 arguments
10	0.245	0.280	0.333	0.387
1,000	0.246	0.281	0.334	0.387
10,000	0.246	0.281	0.334	0.387
100,000	0.246	0.281	0.335	0.388
1,000,000	0.247	0.282	0.334	0.388

Table 5.4: Empty task execution time (s) on the Xeon Phi by number of arguments with transfer grouping.

I reran the null task benchmark using this refined allocation scheme. The results are shown in Table 5.4. There is still some overhead associated with having multiple arguments. This is unavoidable because of the larger control message, and because there is simply more data to transfer. However, the scalability is much improved, as is shown in Figure 5.3 for arguments of a one million bytes. The results are very similar for other argument sizes as well.

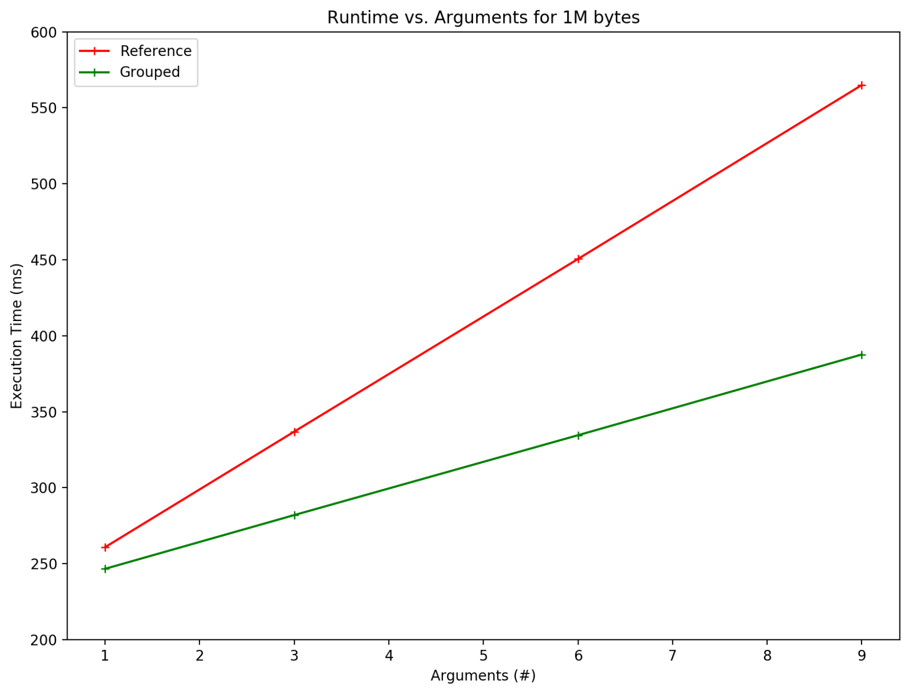


FIGURE 5.3: Comparison of the overhead introduced by the framework with sink transfer grouping both disabled and enabled.

5.3 Optimising Memory Transfers

The previous benchmark shows that even a single memory transfer to the Xeon Phi incurs significant overhead. I will now explore how my implementation of the API mitigates this.

It is important to note that, due to the data-flow nature of task graphs, sinks are implicitly immutable. Indeed, multiple copies of a sink might exist at the same time (one for each execution context where it is needed). This means that, if sinks could be modified, there would be situations where the same sinks are modified in conflicting ways at the same time. The clean resolution of such conflicts would require complex guarantees over the task graph's execution order, as well as precise precedence rules for concurrent execution. Alternatively, conflict resolution could be delegated to the user, but it is unclear how such an interface would be defined. I decided to make sinks explicitly immutable in order to side-step these difficult issues.

Furthermore, sink immutability enables a key optimisation: there needs to be only one sink transfer to an execution context where it is needed. I implemented this in the framework by augmenting the sink metadata to track the devices where a given sink is already available. This enables the framework to transfer sinks only once, the first time they are needed on a specific device.

However, preventing input sinks from being modified prevents the usage of algorithms that operate in-place. This is practically mitigated by simply copying the input data to the output sink, but this introduces and additional copying steps which negates all the performance benefits of performing modifications in-place.

5.3.1 Matrix Multiplication Benchmark

The benchmark consists of multiplying three matrices together ($A \cdot B \cdot C$). The task graph topology of the benchmark is shown in Figure 5.4. Traditional frameworks for work offloading, such as Intel's SCIF or a normal GPU driver would require the user to make a choice either to keep the matrix multiplication code simple and suffer the two additional memory transfer of the intermediate result to the host and back, or to hard-code the optimal transfer pattern and specialise the matrix multiplication routine to the three matrices case. In contrast, the task graph API uses static knowledge of the application's data-flow to provide an efficient memory transfer pattern, whilst keeping the matrix multiplication code simple. I created a microbenchmark that helps expose this improvement.

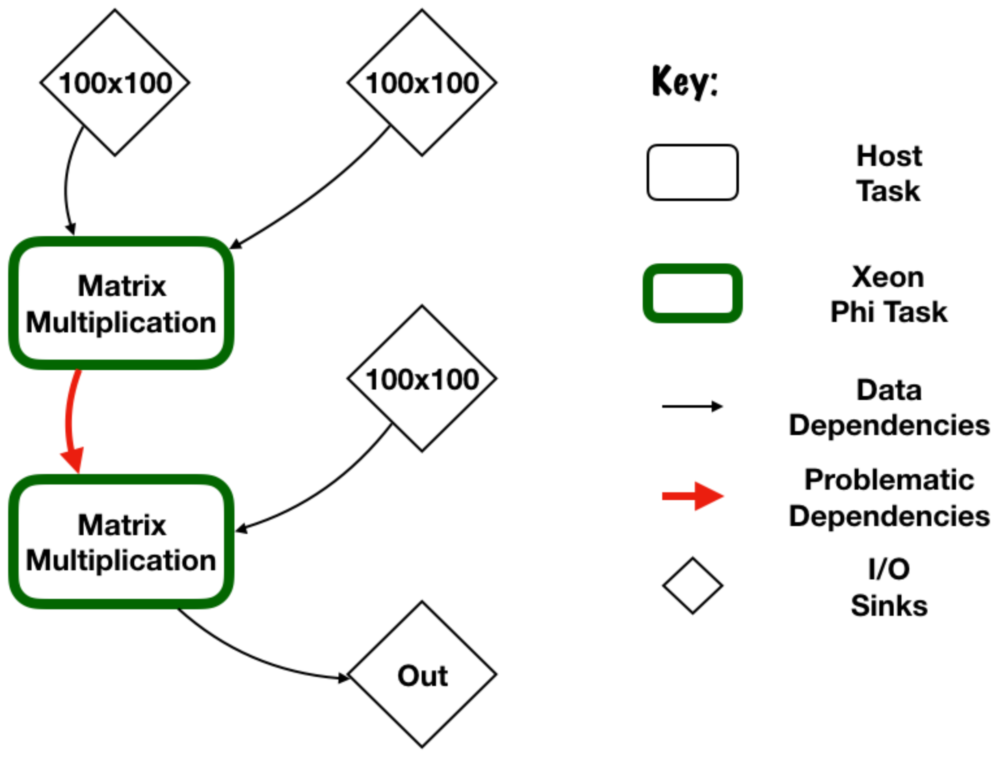


Figure 5.4: Matrix multiplication benchmark task graph topology.

I ran the benchmark with the optimised memory transfers feature enabled and disabled, with square matrices of 10,000 elements. I show the results in Figure 5.5.

Reducing the number of sink transfers is clearly worthwhile. The benchmark completes 120 milliseconds faster, which represents a 22% improvement. The computation performed by the two versions is identical, so the speed up is solely due to the better data transfer pattern.

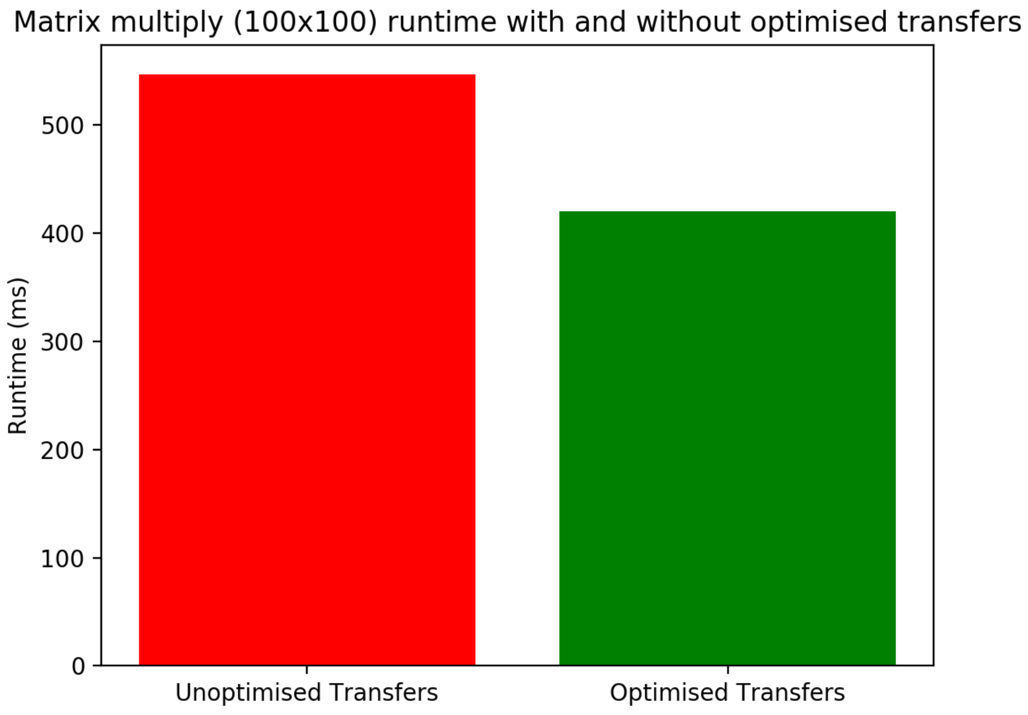


FIGURE 5.5: Matrix multiplication microbenchmark with and without optimised memory transfers.

Beyond removing redundant data transfers, an optimisation opportunity would have been to overlap the data transfer and communication steps. For example in the matrix multiplication benchmark, the third matrix could be sent to the Xeon Phi whilst the multiplication of the first two matrices is occurring. I did not implement this feature due to a lack of time, but it is a natural extension to to the other optimisations presented in this chapter.

5.4 Limitations

This study would have clearly benefited from the comparison with the Intel development stack, as well as from the study of a larger example on both development platforms. Unfortunately, the license for the Intel compiler and runtime libraries I was using expired before I could conduct these experiments. The license had been generously provided by the Systems Group at ETHZ in Switzerland, who used it as part of the work performed by Reto Achermann [43]. As they had no need for it any more, they could not justify extending it.

Chapter 6

Conclusion and Future Work

Hennessy and Patterson [45] described an exciting new future of domain-specific hardware and software co-design in their Turing Award Talk at ISCA 2018. They argue that the end of Moore's Law and the recent Spectre/Meltdown vulnerabilities are marking the end of traditional processor design and the start of a new golden age of computer architecture. In this new world of domain-specific accelerators, the question of how best to expose them as a coherent unit to the programmer remains open.

In this project, I provided an analysis of the benefits of Xeon Phi accelerator cards, in the context of main memory hash joins. Despite technical difficulties in conducting the evaluation of the main state-of-the-art algorithms, I have shown that the Xeon Phi is promising for high-performance, non-partitioned algorithms. I suspect that these hardware-oblivious algorithms will prevail in the future described by Hennessy and Patterson [45], as they do not rely on architectural quirks of the cache hierarchies typical of current multicore processors. Instead they bank on the availability of large amounts of parallelism, which seems to be a more reasonable assumption.

I then explored new programming interfaces that let programmers leverage accelerators more easily. I implemented a new API, which allows users to express their applications in terms of their data flow, instead of specifying individual memory transfers. This lets the framework handle the quirks of the increasingly complex memory hierarchies present on a machine equipped with accelerators. Thus, developers can focus on their algorithms and rest assured that their data is efficiently transferred to the various execution targets.

Barrelfish's multi-kernel design was suitable for this type of task as it allowed me to write low-level components that have knowledge of the whole system, in userspace instead of directly modifying the kernel. This greatly simplified the implementation of my framework. Furthermore, the self-paging interface and the capability system were very helpful in implementing the required shared-memory abstractions, without the need for any kernel intervention.

Future Work

The work I present in this report is by no means complete or "production quality". As a matter of fact, I believe that I have just scratched the surface of the large design space available to those creating frameworks that expose specialised hardware transparently to programmers.

My main concern lies in the memory usage of my framework, especially as very specialised hardware is likely to have only as much memory as it needs to perform its duties. More positively, I propose multiple extensions to my implementation that would help reduce memory usage and maintain it within tight upper-bounds:

- Implement sink “freeing” by analysing the task graph topology and reusing sink memory buffers when they are no longer needed. This would require a big change in the implementation of the sink heap. Garbage collection would be particularly suited for this as it would involve no programmer intervention. Furthermore, the garbage collector’s pointer fixup phase would not be required as all pointers to sink memory buffers are constructed by the framework.
- Create an overspill system similar to traditional virtual memory. In this scheme, when the memory usage becomes too high, the sinks that are going to be used last would be sent back to the host, in a similar manner to a page getting paged out. This provides the illusion of the availability of more memory in exchange for a modest performance penalty.
- Maintain multiple sink heaps, one sink per offload server, instead of a global one. This would allow offload servers to maintain a smaller heap that is only large enough for the sinks needed by the tasks they expose.
- Implement a leaner operating system image for each target. The Barrelfish port to Xeon Phi I used still contained many abstractions that are unnecessary to the task graph framework. Pruning such additional components and integrating the offload client library directly into the kernel should significantly reduce the operating system’s footprint, leaving more room for user applications.

Furthermore, the interface I provide is implemented in C and as such, can sometimes feel a bit clunky and does not elegantly expose the unique features of the Xeon Phi, or of any other potential target. In order to improve the usability of the framework, specialised compilers and domain-specific languages should be used, as proposed by Hennessy and Patterson [45]. In this scenario, the various language runtimes could interface with the task graph framework in order to provide a cohesive view of the system. This project did not explore the possibilities in compiler design for leveraging accelerators. I believe that this is crucial to the adoption of such hardware and this is a research area that I would like to explore in the future. More specifically, cost models of interesting workloads could be constructed and compilers could use this additional information to infer the optimal execution targets for that particular program.

Computer architecture is moving away from traditional system design at an ever increasing pace. It is time for the system’s software stack to follow suit.

List of Abbreviations

ABI	**A**pplication **B**inary **I**nterface
API	**A**pplication **P**rogramming **I**nterface
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
DAG	**D**irected **A**cyclic **G**raph
DMA	**D**irect **M**emory **A**ccess
DSL	**D**omain **S**pecific **L**anguage
FaaS	**F**unction **as** **a** **S**ervice
FIFO	**F**irst **I**n **F**irst **O**ut
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GPGPU	**G**eneral **P**urpose **G**raphics **P**rocessing **U**nit
GPU	**G**raphics **P**rocessing **U**nit
HPC	**H**igh **P**erformance **C**omputing
IaaS	**I**nfrastructure **as** **a** **S**ervice
ILP	**I**nstruction **L**evel **P**arallelism
IPC	**I**nter-**P**rocess **C**ommunication
ISA	**I**nstruction **S**et **A**rchitecture
MIC	**M**any **I**ntegrated **C**ore
MMU	**M**emory **M**anagement **U**nit
NIC	**N**etwork **I**nterface **C**ontroller
NUMA	**N**on **U**niform **M**emory **A**rchitecture
PaaS	**P**latform **as** **a** **S**ervice
RDMA	**R**emote **D**irect **M**emory **A**ccess
RPC	**R**emote **P**rocedure **C**all
SCIF	**S**ymmetric **C**ommunication **I**nter**F**ace
SDN	**S**oftware **D**efined **N**etwork
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SMT	**S**imultaneous **M**ulti **T**hreading
TLB	**T**ranslation **L**ookaside **B**uffer
VM	**V**irtual **M**achine
WAN	**W**ide **A**rea **N**etwork

List of Figures

Bibliography

[1] Jon Brodkin, *Who needs HP and Dell? Facebook now designs all its own servers*, https://arstechnica.com/information-technology/2013/02/who-needs-hp-and-dell-facebook-now-designs-all-its-own-servers/, [Online; accessed January 25rd 2018], 2013.

[2] Neil Brown, *Ghosts of Unix Past: a historical search for design patterns*, https://lwn.net/Articles/411845/, [Online; accessed January 25rd 2018], 2010.

[3] Baumann, Andrew and Barham, Paul and Dagand, Pierre-Evariste and Harris, Tim and Isaacs, Rebecca and Peter, Simon and Roscoe, Timothy and Schüpbach, Adrian and Singhania, Akhilesh, "The Multikernel: A New OS Architecture for Scalable Multicore Systems", in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09, Big Sky, Montana, USA: ACM, 2009, pp. 29–44, ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629579. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629579.

[4] Apple Inc., *The future is here: Iphone X*, https://www.apple.com/uk/newsroom/2017/09/the-future-is-here-iphone-x/, [Online; accessed January 23rd 2018], 2017.

[5] Simon Sharwood, *Microsoft offloads networking to FPGA-powered NICs*, https://www.theregister.co.uk/2018/01/08/azure_fpga_nics/, [Online; accessed January 23rd 2018], 2018.

[6] John Owens, *GPU Architecture Overview*, http://gpgpu.org/static/s2007/slides/02-gpu-architecture-overview-s07.pdf, [Online; accessed January 23rd 2018], 2007.

[7] DJ Greaves, *Pollack's Rule*, http://www.cl.cam.ac.uk/research/srg/han/ACS-P35/obj-5.1/zhp3123b9dbe.html, [Online; accessed January 23rd 2018], 2012.

[8] Luke Durant and Olivier Giroux and Mark Harris and Nick Stam, *Inside Volta: The World's Most Advanced Data Center GPU*, https://devblogs.nvidia.com/inside-volta/, [Online; accessed January 23rd 2018], 2017.

[9] George Chrysos, *Intel® Xeon Phi™ X100 Family Coprocessor - the Architecture*, https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner, [Online accessed January 23rd 2018], 2012.

[10] Jeffers, James and Reinders, James, *Intel Xeon Phi Coprocessor High Performance Programming*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013, ISBN: 9780124104143, 9780124104945.

[11] Intel Corp, *Intel® Xeon Phi™ Coprocessor 7120A*, https://ark.intel.com/products/80555/Intel-Xeon-Phi-Coprocessor-7120A-16GB-1_238-GHz-61-core, [Online accessed January 23rd 2018].

[12] ——, *Intel® Xeon PhiTM Coprocessor Instruction Set Architecture Reference Manual*, https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf, [Online accessed January 23rd 2018], 2012.

[13] Mogul, Jeffrey C., "TCP Offload is a Dumb Idea Whose Time Has Come", in *Proceedings of the 9th Conference on Hot Topics in Operating Systems - VOLUME 9*, ser. HOTOS'03, Lihue, Hawaii: USENIX Association, 2003, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251054.1251059.

[14] Mellanox Technologies, *NP-4 Network Processor*, http://www.mellanox.com/related-docs/prod_npu/PB_NP-4.pdf, [Online accessed January 24rd 2018], 2017.

[15] ——, *ConnectX-3 EN*, http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX3_EN_Card.pdf, [Online accessed January 24rd 2018], 2014.

[16] Albert Greenberg, *SDN for the Cloud*, https://conferences.sigcomm.org/sigcomm/2015/pdf/papers/keynote.pdf, [Online accessed January 24rd 2018], 2015.

[17] Babak Falsafi and Tim Harris and Dushyanth Narayanan and David A. Patteson, "Rack-scale Computing (Dagstuhl Seminar 15421)", *Dagstuhl Reports*, vol. 5, no. 10, B. Falsafi, T. Harris, D. Narayanan, and D. A. Patterson, Eds., pp. 35–49, 2016, ISSN: 2192-5283. DOI: 10.4230/DagRep.5.10.35. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/5697.

[18] Nightingale, Edmund B and Hodson, Orion and McIlroy, Ross and Hawblitzel, Chris and Hunt, Galen, "Helios: Heterogeneous Multiprocessing with Satellite Kernels", in *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP '09)*, Association for Computing Machinery, Inc., Oct. 2009.

[19] Beisel, Tobias and Wiersema, Tobias and Plessl, Christian and Brinkmann, André, "Programming and Scheduling Model for Supporting Heterogeneous Accelerators in Linux", in *Proc. Workshop on Computer Architecture and Operating System Co-design (CAOS)*, 2012.

[20] Intel Corporation, *Intel QuickPath Architecture*, https://www.intel.com/pressroom/archive/reference/whitepaper_QuickPath.pdf, [Online accessed January 24rd 2018], 2008.

[21] Josef Bacik, *Linux Kernel Scheduler Basics*, https://josefbacik.github.io/kernel/scheduler/2017/07/14/scheduler-basics.html, [Online accessed January 25rd 2018], 2017.

[22] Barbalace, Antonio and Sadini, Marina and Ansary, Saif and Jelesnianski, Christopher and Ravichandran, Akshay and Kendir, Cagil and Murray, Alastair and Ravindran, Binoy, "Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms", in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15, Bordeaux, France: ACM, 2015, 29:1–29:16, ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741962. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741962.

[23] Bratterud, Alfred and Walla, Alf-Andre and Haugerud, Harek and Engelstad, Paal E. and Begnum, Kyrre, "IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services", in *Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, ser. CLOUDCOM '15, Washington, DC, USA: IEEE Computer Society, 2015, pp. 250–257, ISBN: 978-1-4673-9560-1. [Online]. Available: http://dx.doi.org/10.1109/CloudCom.2015.89.

[24] Nadav Har'El, Benoît Canet, *Serverless Computing With OSv*, http://blog.osv.io/blog/2017/06/12/serverless-computing-with-OSv/, [Online accessed January 26rd 2018].

[25] Min, Changwoo and Kang, Woonhak and Kumar, Mohan and Kashyap, Sanidhya and Maass, Steffen and Jo, Heeseung and Kim, Taesoo, "Solros: A Data-centric Operating System Architecture for Heterogeneous Computing", in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18, Porto, Portugal: ACM, 2018, 36:1–36:15, ISBN: 978-1-4503-5584-1. DOI: 10.1145/3190508.3190523. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190523.

[26] Simon Gerber and Gerd Zellweger and Reto Achermann and Kornilios Kourtis and Timothy Roscoe and Dejan Milojicic, "Not Your Parents' Physical Address Space", in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/hotos15/workshop-program/presentation/gerber.

[27] Garey, M. R. and Johnson, D. S., "Tutorial: Hard real-time systems", in, J. A. Stankovic and K. Ramamritham, Eds., Los Alamitos, CA, USA: IEEE Computer Society Press, 1989, ch. Complexity Results for Multiprocessor Scheduling Under Resource Constraints, pp. 205–219, ISBN: 0-8186-0819-6. [Online]. Available: http://dl.acm.org/citation.cfm?id=76933.76948.

[28] Feitelson, Dror G. and Jette, Morris A., "Improved Utilization and Responsiveness with Gang Scheduling", in *Proceedings of the Job Scheduling Strategies for Parallel Processing*, ser. IPPS '97, London, UK, UK: Springer-Verlag, 1997, pp. 238–261, ISBN: 3-540-63574-2. [Online]. Available: http://dl.acm.org/citation.cfm?id=646378.689518.

[29] Blumofe, Robert D. and Leiserson, Charles E., "Scheduling Multithreaded Computations by Work Stealing", *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999, ISSN: 0004-5411. DOI: 10.1145/324133.324234. [Online]. Available: http://doi.acm.org/10.1145/324133.324234.

[30] Augonnet, Cédric and Thibault, Samuel and Namyst, Raymond and Wacrenier, Pierre-André, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures", *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, 2011, ISSN: 1532-0626. DOI: 10.1002/cpe.1631. [Online]. Available: http://dx.doi.org/10.1002/cpe.1631.

[31] Rossbach, Christopher J. and Currey, Jon and Silberstein, Mark and Ray, Baishakhi and Witchel, Emmett, "PTask: Operating System Abstractions to Manage GPUs As Compute Devices", in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, Cascais, Portugal: ACM, 2011, pp. 233–248, ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043579. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043579.

[32] Teubner, Jens and Alonso, Gustavo and Balkesen, Cagri and Ozsu, M. Tamer, "Main-memory Hash Joins on Multi-core CPUs: Tuning to the Underlying Hardware", in *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ser. ICDE '13, Washington, DC, USA: IEEE Computer Society, 2013, pp. 362–373, ISBN: 978-1-4673-4909-3. DOI: 10.1109/ICDE.2013.6544839. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2013.6544839.

[33] Ambuj Shatdal and Chander Kant and Jeffrey F. Naughton, "Cache Conscious Algorithms for Relational Query Processing", in *VLDB*, 1994.

[34] Boncz, Peter A. and Manegold, Stefan and Kersten, Martin L., "Database Architecture Optimized for the New Bottleneck: Memory Access", in *Proceedings of the 25th International Conference on Very Large Data Bases*, ser. VLDB '99, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 54–65, ISBN:

1-55860-615-7. [Online]. Available: http://dl.acm.org/citation.cfm?id=645925.671364.

[35] Kim, Changkyu and Kaldewey, Tim and Lee, Victor W. and Sedlar, Eric and Nguyen, Anthony D. and Satish, Nadathur and Chhugani, Jatin and Di Blas, Andrea and Dubey, Pradeep, "Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs", *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1378–1389, Aug. 2009, ISSN: 2150-8097. DOI: 10.14778/1687553.1687564. [Online]. Available: https://doi.org/10.14778/1687553.1687564.

[36] Intel Corp, *Intel Manycore Platform Software Stack (Intel MPSS)*, https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss#about, [Online accessed June 2018], 2017.

[37] Avinash Sodani, "Knights Landing (KNL): 2nd Generation Intel Xeon Phi processor", in *2015 IEEE Hot Chips 27 Symposium (HCS)*, Aug. 2015, pp. 1–24. DOI: 10.1109/HOTCHIPS.2015.7477467.

[38] Jianbin Fang and Ana Lucia Varbanescu and Henk J. Sips and Lilun Zhang and Yonggang Che and Chuanfu Xu, "An Empirical Study of Intel Xeon Phi", *CoRR*, vol. abs/1310.5842, 2013.

[39] Molka, Daniel and Hackenberg, Daniel and Schöne, Robert, "Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer", in *Proceedings of the Workshop on Memory Systems Performance and Correctness*, ser. MSPC '14, Edinburgh, United Kingdom: ACM, 2014, 4:1–4:10, ISBN: 978-1-4503-2917-0. DOI: 10.1145/2618128.2618129. [Online]. Available: http://doi.acm.org/10.1145/2618128.2618129.

[40] Jha, Saurabh and He, Bingsheng and Lu, Mian and Cheng, Xuntao and Huynh, Huynh Phung, "Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach", *Proc. VLDB Endow.*, vol. 8, no. 6, pp. 642–653, Feb. 2015, ISSN: 2150-8097. DOI: 10.14778/2735703.2735704. [Online]. Available: http://dx.doi.org/10.14778/2735703.2735704.

[41] Pohl, Constantin and Sattler, Kai-Uwe, "Joins in a Heterogeneous Memory Hierarchy: Exploiting High-bandwidth Memory", in *Proceedings of the 14th International Workshop on Data Management on New Hardware*, ser. DAMON '18, Houston, Texas: ACM, 2018, 8:1–8:10, ISBN: 978-1-4503-5853-8. DOI: 10.1145/3211922.3211929. [Online]. Available: http://doi.acm.org/10.1145/3211922.3211929.

[42] Andrew Baumann, "Inter-dispatcher communication in Barrelfish", Systems Group, Department of Computer Science ETH Zurich, CAB F.70, Universitatstrasse 6, Zurich 8092, Switzerland, Tech. Rep. TN-011, Dec. 2012. [Online]. Available: http://www.barrelfish.org/.

[43] Reto Achermann, "Message passing and bulk transport on heterogenous multiprocessors", Master's thesis, ETH Zurich, Zurich, Switzerland, 2014.

[44] Intel Corporation, Ed., *Symmetric Communications Interface (SCIF) For Intel Xeon Phi Product Family Users Guide*, 1.03, Intel Corporation, 2014.

[45] Hennessy, John L. and Patterson, David A. (Jun. 4, 2018). A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development. Turing Award Lecturei given at ISCA2018, [Online]. Available: http://iscaconf.org/isca2018/turing_lecture.html (visited on 06/07/2013).